САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО
Институт Компьютерных наук и технологий
Высшая школа искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Отчёт по курсовой работе по дисциплине «Управление ресурсами СуперЭВМ»

«Решение задачи построения кратчайшего пути движения робота по полигону с использованием аппаратно-программной платформы NVIDIA CUDA»

Группа: 3530201/00101

Студент: _____ Спасов Григорий Ефимович

Преподаватель: _____ Курочкин Михаил Александрович

«___»_____ 20___ г.

Санкт-Петербург – 2023

Содержание

Введение **3**

1 Постановка задачи **4**

2 Аппаратно-программная платформа NVIDIA CUDA **5**
2.1 Архитектура NVIDIA CUDA 5
2.2 Вычислительные возможности NVIDIA CUDA 5
2.3 Потоковая модель 7
2.4 Устройство памяти 8
2.5 Модели памяти 8
2.6 Модель вычислений на GPU 9
2.7 Интерфейс программирования CUDA C 11
2.7.1 Спецификаторы типов функций 11
2.7.2 Правила и ограничения при объявлении функций 11
2.7.3 Спецификаторы типов переменных 12
2.7.4 При объявлении переменных действуют следующие правила и ограничения: 12
2.7.5 Встроенные переменные 13
2.7.6 Конфигурирование исполнения ядер 13
2.7.7 Kernel 14
2.8 Компиляция программы 15
2.9 Планировщик задач 15

3 Суперкомпьютерный центр «Политехнический» **17**
3.1 Состав 17
3.2 Характеристики 17
3.3 Технология подключения 17

4 Постановка решаемой практической задачи **19**
4.1 Алгоритм решения задачи 19
4.2 Метод распараллеливания алгоритма 21
4.3 Результат работы программы 22

5 Описание эксперимента **26**
5.1 Результаты эксперимента 27
5.2 Анализ результатов эксперимента 29

Заключение **30**

Список источников **31**

Введение

Суперкомпьютер – это высокопроизводительная компьютерная система, способная выполнять огромное количество вычислений за короткий промежуток времени. Он состоит из сотен и тысяч процессоров, объединенных вместе для параллельной обработки данных.

Суперкомпьютеры используются в различных областях науки и промышленности. Они позволяют проводить более точные и сложные моделирования, исследования климата и погоды, расчеты в физике и астрономии, анализ геномов, разработку новых материалов, аэродинамические и гидродинамические расчеты, машинное обучение и многое другое. Суперкомпьютеры являются неотъемлемым инструментом для научных и инженерных исследований, где требуется высокая вычислительная мощность.

Одним из примеров суперкомпьютерного центра в Санкт-Петербурге является суперкомпьютерный центр "Политехнический". Он базируется на технологиях NVIDIA и предоставляет вычислительные ресурсы для решения различных научных и инженерных задач. В частности, для решения практических задач на графическом процессоре суперкомпьютера NVIDIA используется платформа CUDA. CUDA (Compute Unified Device Architecture) - это архитектура, разработанная NVIDIA, которая позволяет разработчикам программ использовать вычислительные возможности графических процессоров (GPU) для выполнения широкого спектра вычислительных задач. CUDA обеспечивает эффективную параллельную обработку данных и значительное ускорение вычислений на суперкомпьютерах, использующих графические процессоры NVIDIA.

В данной курсовой работе представлен обзор аппаратно-программной платформы NVIDIA CUDA, её архитектурные особенности, модель вычислений, модель памяти и особенности работы с графическими процессорами NVIDIA. Также представлен обзор вычислительных систем и средств суперкомпьютерного центра «Политехнический», его состав и характеристики вычислительных узлов.

В качестве практической части курсовой работы был описан и реализован алгоритм построения пути движения робота по полигону с использованием аппаратно-программной платформы NVIDIA CUDA. Полученная программа была запущена на вычислительном узле суперкомпьютерного центра «Политехнический».

Был поставлен эксперимент, в ходе которого выяснялась зависимость времени исполнения программы от параметров распараллеливания на GPU.

1 Постановка задачи

В рамках выполнения курсовой работы были поставлены следующие задачи:

1. Изучить аппаратно-программную платформу NVIDIA CUDA: её архитектуру, вычислительные возможности, устройство памяти, модель вычислений;
2. Изучить и применить на практике технологию программирования CUDA C;
3. Изучить состав и характеристики вычислительных средств суперкомпьютерного центра «Политехнический», изучить и применить технологию подключения к СКЦ;
4. Реализовать с использованием технологии CUDA C алгоритм построения кратчайшего пути движения робота по полигону;
5. Поставить эксперимент для выяснения зависимости времени выполнения программы от выбора модели памяти и размера исходного массива;
6. Проанализировать результаты эксперимента.

2 Аппаратно-программная платформа NVIDIA CUDA

2.1 Архитектура NVIDIA CUDA

Архитектура NVIDIA CUDA (Compute Unified Device Architecture) - это программная и аппаратная платформа, разработанная компанией NVIDIA, которая позволяет использовать графические процессоры (GPU) для параллельных вычислений и ускорения обработки данных [1].

Основной принцип архитектуры CUDA заключается в использовании мощности вычислений графического процессора для выполнения параллельных задач, работающих вместе с центральным процессором (CPU) компьютера. Графический процессор состоит из множества вычислительных ядер, которые могут выполнять однотипные задачи одновременно, независимо друг от друга. Это позволяет обрабатывать большие объемы данных и решать сложные вычислительные задачи с высокой производительностью.

Основные характеристики архитектуры NVIDIA CUDA:

- Язык программирования: CUDA поддерживает язык программирования CUDA C/C++, который предоставляет расширения для работы с параллельными вычислениями на GPU.

- Модель исполнения: CUDA использует модель исполнения, основанную на понятии ядер (kernels). Ядро (kernel) - это функция, которая выполняется параллельно на множестве потоков на графическом процессоре.

- Модель памяти: CUDA предоставляет шесть типов памяти, включая глобальную память, разделяемую память (shared memory) и константную память (constant memory). Каждый тип памяти имеет свои особенности и используется для различных целей, таких как обмен данными между потоками и сохранение постоянных данных.

2.2 Вычислительные возможности NVIDIA CUDA

Параметр "compute capability" (вычислительная способность) в NVIDIA CUDA используется для описания возможностей и характеристик конкретной архитектуры графического процессора [2]. Он определяет, какие функции и возможности доступны для программ, компилируемых с использованием CUDA.

Вычислительная способность обозначается числовым значением, состоящим из двух цифр, разделенных точкой (например, 6.1 или 7.0). Первая цифра обозначает основную версию архитектуры, а вторая цифра указывает на подверсию.

Вычислительная способность зависит от конкретной архитектуры GPU и определяет следующие характеристики:

- Количество и тип вычислительных ядер: Вычислительная способность определяет количество ядер SM (Streaming Multiprocessors) на графическом процессоре и тип архитектуры ядра, такой как NVIDIA's Fermi, Kepler, Maxwell, Pascal, Volta, Turing или Ampere. Каждая архитектура имеет свои особенности и функции.

- Размер разделяемой памяти и регистров: Вычислительная способность также определяет доступный объем разделяемой памяти (shared memory) и количество доступных регистров для каждого потока выполнения на графическом процессоре. Эти ресурсы могут использоваться для ускорения вычислений и обмена данными между потоками.

- Поддержка определенных функций и инструкций: Вычислительная способность определяет, какие функции и инструкции доступны для использования в программе CUDA. Новые архитектуры могут включать новые инструкции и возможности, которые не доступны в старых версиях.

- Производительность и энергоэффективность: Вычислительная способность также связана с производительностью и энергоэффективностью архитектуры GPU. Более новые версии архитектур часто имеют улучшенную производительность и более эффективное использование энергии.

В СК «Политехник - РСК Торнадо», на котором проводились вычисления, установлены графические процессоры Nvidia Tesla K40. Они имеют compute capability версии 3.5, ниже приведены их основные характеристики.

- Количество одновременно выполняющихся ядер: 32;
- Максимальная размерность сетки блоков: 3-х мерная;
- Максимальная x-составляющая сетки блоков: 2^{31-1};
- Максимальная y- и z-составляющая сетки блоков: 65535;
- Максимальная размерность блока: 3-х мерный;
- Максимальная x- и y-составляющая блока: 1024;
- Максимальная z-составляющая блока: 64;
- Максимальное количество нитей в блоке: 1024;
- Размер варпа: 32;
- Максимальное количество одновременно выполняющихся блоков на SM: 16;
- Максимальное количество варпов на SM: 64;
- Максимальное количество нитей на SM: 2048;
- Количество 32-битных регистров на SM: 64K;
- Максимальное количество 32-битных регистров на исполняемый блок: 64K;
- Максимальное количество 32-битных регистров на исполняемую нить: 255;
- Максимальный объем разделяемой памяти: 48 KB;
- Максимальное объем разделяемой памяти на блок: 48 KB;
- Объем константной памяти: 64 KB;
- Рабочий объем кэша константой памяти на SM: 8 KB;

2.3 Потоковая модель

Вычислительная архитектура CUDA основана на концепции одна команда на множество данных (Single Instruction Multiple Data, SIMD) и понятии мультипроцессора.

Концепция **SIMD** подразумевает, что одна инструкция позволяет одновременно обработать множество данных.

Мультипроцессор — это многоядерный SIMD процессор, позволяющий в каждый определенный момент времени выполнять на всех ядрах только одну инструкцию. Каждое ядро мультипроцессора скалярное, т.е. оно не поддерживает векторные операции в чистом виде.

Также в вычислительной архитектуре CUDA важны понятия устройство и хост.

Устройство (device) - видеоадаптер, поддерживающий драйвер CUDA, или другое специализированное устройство, предназначенное для исполнения программ, использующих CUDA.

Хост (host) - программа в обычной оперативной памяти компьютера, использующую CPU и выполняющую управляющие функции по работе с устройством.

Логически устройство можно представить как набор мультипроцессоров (см. Рис. 1) плюс драйвер CUDA.

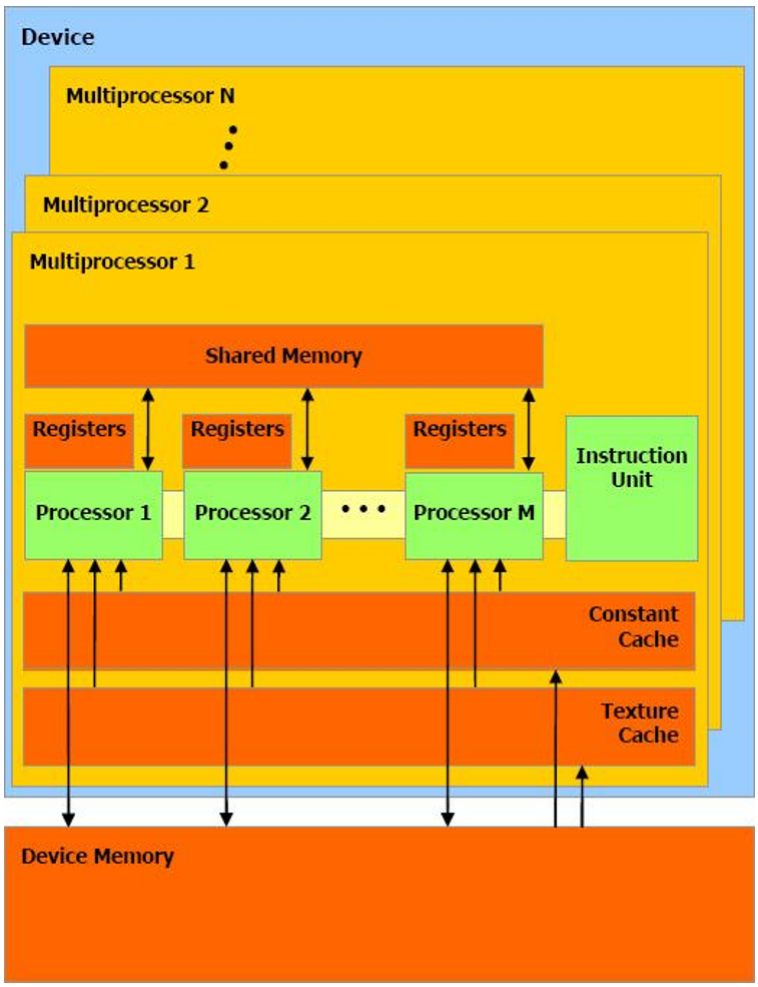

Рис. 1: Архитектура устройства

2.4 Устройство памяти

В CUDA выделяют шесть видов памяти (рис. 2). Это регистры, локальная, глобальная, разделяемая, константная и текстурная память. Такое обилие обусловлено спецификой видеокарты и первичным ее предназначением, а также стремлением разработчиков сделать систему как можно дешевле, жертвуя в различных случаях либо универсальностью, либо скоростью. Подробно каждый вид памяти будет рассмотрен в следующем разделе.

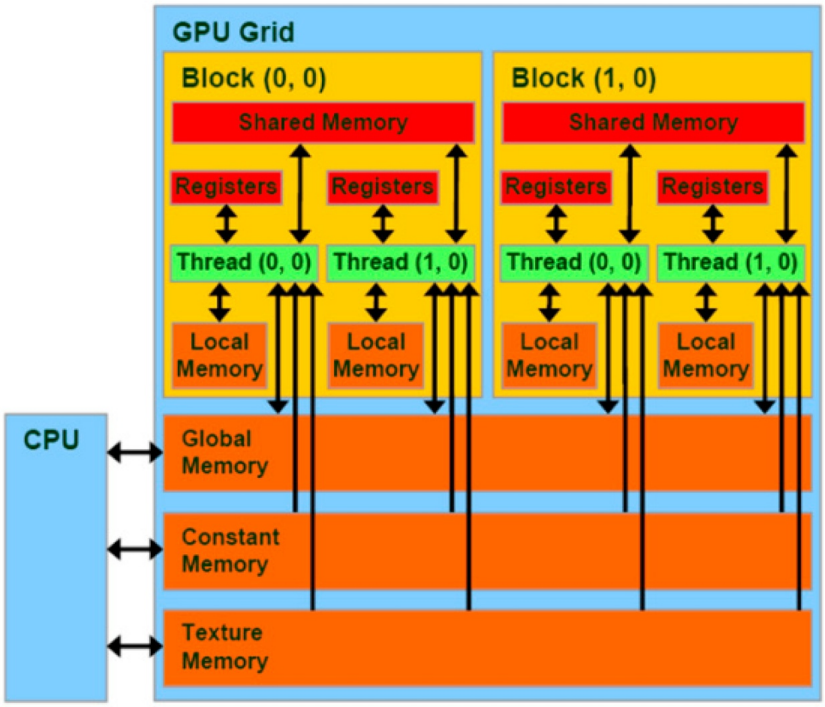

Рис. 2: Виды памяти

2.5 Модели памяти

В CUDA выделяют шесть видов памяти:

1. **Регистры** – память, в которой по возможности компилятор старается размещать все локальные переменные функций. Доступ к таким переменным осуществляется с максимальной скоростью. В текущей архитектуре на один мультипроцессор доступно 8192 32-разрядных регистра.

2. **Локальная память** – в случаях, когда локальные данные процедур занимают слишком большой размер, или компилятор не может вычислить для них некоторый постоянный шаг при обращении, он может поместить их в локальную память. Этому может способствовать, например, приведение указателей для типов разных размеров. Физически локальная память является аналогом глобальной памяти, и работает с той же скоростью.

3. **Глобальная память** – самый большой объём памяти, доступный для всех МП на видеочипе (размер от 256Мбайт до 4Гбайт). Основная особенность - возможность произвольной адресации. Однако глобальная память работает очень медленно, не кэшируется. Поэтому количество обращений к глобальной памяти следует минимизировать. Глобальная память необходима в основном для сохранения результатов работы программы перед отправкой их на хост (в обычную память DRAM).

4. **Разделяемая память** – некэшируемая, но быстрая память. Ее и рекомендуется использовать как управляемый кэш. На один мультипроцессор доступно всего 16KB разделяемой памяти. Отличительной чертой разделяемой памяти является то, что она адресуется одинаково для всех задач внутри блока. Отсюда следует, что ее можно использовать для обмена данными между потоками только одного блока.

5. **Константная память** – размер составляет всего 64 Кбайт (на все устройство). Константная память кэшируется, поэтому доступ в общем случае достаточно быстрый. Кэш существует в единственном экземпляре для одного мультипроцессора, а значит, общий для всех задач внутри блока. Константная память очень удобна в использовании. Можно размещать в ней данные любого типа и читать их при помощи простого присваивания. Однако из-за её небольшого объема имеет смысл хранить лишь небольшое количество часто используемых данных.

6. **Текстурная память** – блок памяти, доступный для чтения всеми МП. Кэшируется. Медленная, как глобальная - сотни тактов задержки при отсутствии данных в кэше. Имеет очень важное свойство пространственной локальности. При вычислении на модели в виде матрицы, где соседние элементы взаимодействуют друг с другом, часто возникает необходимость обращаться к элементам окрестности элемента матрицы. С точки зрения адресной арифметики элементы окрестности какого-либо элемента матрицы не расположены в памяти рядом друг с другом. Для ускорения вычисления вышеописанного вида применяется текстурная память, которая позволяет кэшировать данные по свойству их пространственной, а не адресной локальности.

2.6 Модель вычислений на GPU

В основе модели вычислений на GPU лежит понятие сетки блоков. На Рис. 3 ядро обозначено как Kernel. Все потоки, выполняющие это ядро, объединяются в блоки (Block), а блоки, в свою очередь, объединяются в сетку (Grid).

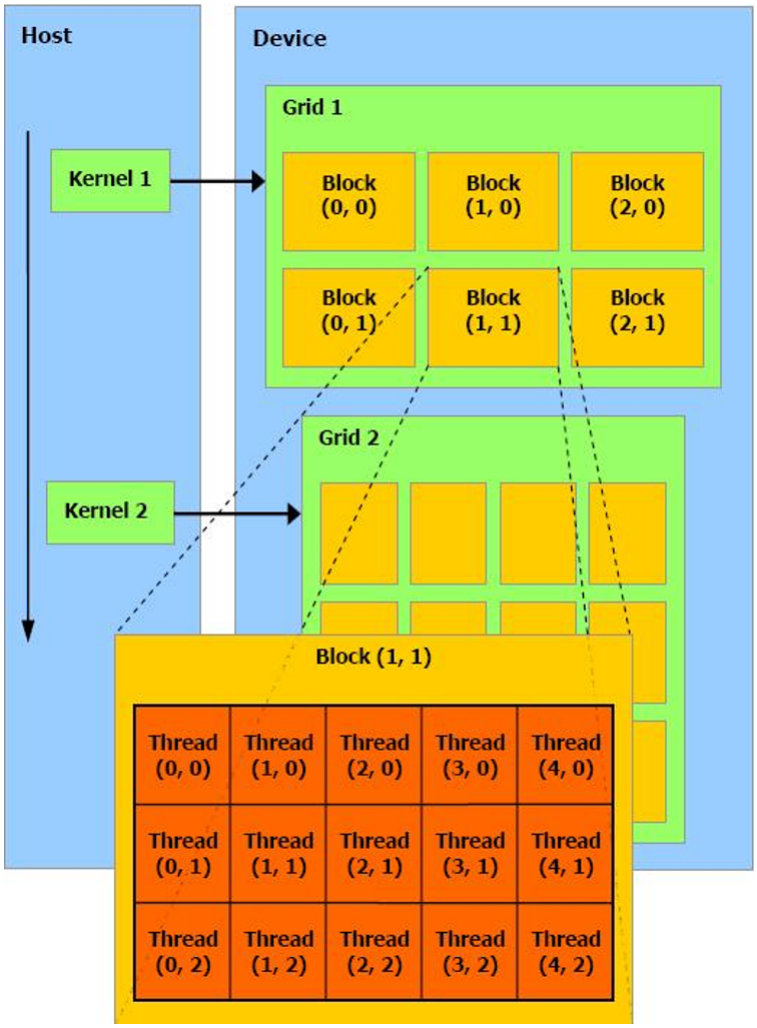

Рис. 3: Организация потоков

Вычисление шейдерной программы (ядра) на матрице данных распределяется по блокам, ответственным за отдельно взятый участок матрицы данных. Блоки образуют одно-двух- трёх мерную сетку.

В свою очередь выполнение программы на участке матрицы данных блока выполняется параллельно в нитях, из которых состоит блок. Нить — это экземпляр выполняемого ядра на отдельно взятом наборе входных данных.

Не все нити блока запускаются на выполнение одновременно: они разбиваются на варпы[1] по 32 нити.

Когда потоки внутри варпа выполняют доступы к памяти, например, чтение или запись в глобальную память, они часто обращаются к смежным адресам памяти. Кэширование варпов используется для ускорения этих доступов к памяти – когда один поток в варпе выполняет доступ к памяти, данные загружаются из глобальной памяти в специальный кэш варпа. Если другие потоки в том же варпе позже обратятся к тем же адресам памяти, они смогут получить данные непосредственно из кэша варпа, избегая доступа к глобальной памяти. Это позволяет существенно сократить задержку на чтение и запись в память. Если потоки в варпе обращаются к памяти в произвольном порядке или часто совершают прыжки в адресном пространстве, эффективность кэширования варпа может снизиться.

[1]Варп (warp) - группа одинаковых потоков, исполняющих одну и ту же инструкцию одновременно на графическом процессоре NVIDIA CUDA.

При разбиении задачи на блоки стоит учитывать ограничения аппаратной архитектуры CUDA, которые заключаются в том, что на одном SM может выполнятся несколько блоков, но один блок может выполнятся только на одном SM.

Очевидно, что одновременное выполнение одинаковых инструкций на нитях в варпе возможно только если в шейдерной программе не используются операторы ветвления.В случае, если операторы ветвления используются, то варп разделяется на две группы нитей, которые поочерёдно выполняют инструкции одной и другой ветки условного оператора.

2.7 Интерфейс программирования CUDA C

2.7.1 Спецификаторы типов функций

Программисту доступны следующие спецификаторы функций:

- __device__ объявляет функцию, которая: выполняется на GPU; может быть вызвана только с GPU.
- __global__ объявляет функцию (ядро), которая: выполняется на GPU; может быть вызвана только с ЦПУ.
- __host__ объявляет функцию, которая: выполняется на ЦПУ; может быть вызвана только с ЦПУ. Объявление функции со спецификатором __host__ эквивалентно ее объявлению без спецификаторов __device__, __global__ или __host__; в обоих случаях функция компилируется только для ЦПУ.

2.7.2 Правила и ограничения при объявлении функций

- Спецификатор __host__ может использоваться совместно со спецификатором __device_ В данном случае функция компилируется как для ЦПУ, так и для GPU.
- Функции со спецификатором __device__ и __global__ не могут содержать объявления статических переменных, не поддерживают переменное число аргументов.
- Функции со спецификатором __global__ не поддерживают рекурсию. Ранние версии CUDA и устройств также не поддерживают рекурсию для функций со спецификатором __device__.
- Спецификаторы __global__ и __host__ не могут использоваться совместно в объявлении функции.
- Функции со спецификатором __global__ (ядра) должны возвращать тип void.
- Вызов функции со спецификатором __global__ является асинхронным. Это означает, что возврат управления осуществляется до того, как выполнение функции на GPU закончится.
- Параметры функции со спецификатором __global__ передаются через разделяемую память.

2.7.3 Спецификаторы типов переменных

Программисту доступны следующие спецификаторы типов переменных:

- __device__ объявляет переменную, которая: - размещается на GPU в глобальном пространстве памяти; - имеет время жизни, равное времени жизни приложения; - доступна из всех потоков, выполняемых на GPU; - доступна из основной программы через библиотеки времени выполнения.

- __constant__ объявляет переменную, которая: - размещается на GPU в константном пространстве памяти; - имеет время жизни, равное времени жизни приложения; - доступна из всех потоков, выполняемых на GPU; - доступна из основной программы через библиотеки времени выполнения.

- __shared__ объявляет переменную, которая: - размещается на GPU в разделяемой памяти блока потоков (для каждого блока потоков будет создан свой экземпляр переменной); - имеет время жизни, равное времени жизни блока потоков; - доступна из потоков, принадлежащих блоку потоков.

2.7.4 При объявлении переменных действуют следующие правила и ограничения:

- Указанные спецификаторы неприменимы к формальным параметрам функций, а также к локальным переменным функций, исполняемых на ЦПУ.

- Переменные со спецификаторами __shared__ и __constant__ подразумевают статическое хранение.

- Переменные со спецификаторами __device__, __shared__ и __constant__ не могут быть объявлены с помощью ключевого слова extern. Исключение составляет так называемая динамически распределяемая разделяемая память.

- Переменные со спецификаторами __device__ и __constant__ должны быть объявлены в глобальном пространстве имен.

- Переменные со спецификатором __constant__ могут быть инициализированы только с ЦПУ через функции времени выполнения.

- Переменные со спецификатором __shared__ не могут инициализироваться при объявлении.

- Автоматическая переменная, объявленная в выполняемой на GPU функции, без использования вышеперечисленных спецификаторов обычно размещается в регистрах. Однако в некоторых случаях компилятор может размещать ее в локальной памяти. Обычно это происходит, когда объявляются большие структуры или массивы, которые могут потребовать слишком большого количества пространства памяти регистров, либо объявляются массивы, для которых компилятор не может определить, являются ли они индексированными с использованием константных величин.

- Указатели в коде, который выполняется на GPU, поддерживаются до тех пор, пока компилятор способен определить: указывают ли они на пространство общей памяти или на глобальное пространство памяти. В противном случае могут использоваться лишь указатели на память, выделенную в глобальном пространстве памяти GPU.

Необходимо отметить, что на всех GPU с поддержкой CUDA скорость работы разделяемой памяти существенно (на 2 порядка) превосходит скорость работы глобальной памяти 1, поэтому одной из основных техник оптимизации является размещение интенсивно используемых данных в разделяемой памяти. Кроме того, данный вид памяти открывает возможности для эффективной кооперации потоков одного блока.

2.7.5 Встроенные переменные

В приложении на языке CUDA C (в функциях, исполняемых на GPU) доступны следующие встроенные переменные:

- gridDim Переменная типа dim3, содержит текущую размерность решетки;
- blockIdx Переменная типа uint3, содержит индекс блока внутри решетки;
- blockDim Переменная типа dim3, содержит размерность блока потоков;
- threadIdx Переменная типа uint3, содержит индекс потока внутри блока;
- warpSize Переменная типа int, содержит размер варпа в количестве потоков.

Указанные встроенные переменные предназначены только для чтения и не могут быть модифицированы вызывающей программой.

2.7.6 Конфигурирование исполнения ядер

Любой вызов функции со спецификатором __global__ (ядра) должен определять конфигурацию исполнения для данного вызова. Конфигурация выполнения задает размерность решетки и блоков, которые будут использоваться для исполнения функции на GPU. - 1 На устройствах архитектуры Fermi появились L1/L2 кэши для глобальной памяти. В случае попадания в кэш скорость доступа примерно равна скорости доступа к разделяемой памяти. Конфигурация определяется с помощью выражения специального вида «» между именем функции и списком ее аргументов, где:

- grid Переменная типа dim3, которая определяет размерность и размер сетки, так что grid.x × grid.y × grid.z равно числу блоков потоков, которые будут запущены (в ранних версиях CUDA требовалось, чтобы grid.z всегда было равно 1).
- block Переменная типа dim3, которая определяет размерность и размер каждого блока потоков, block.x × block.y × block.z равно числу потоков на блок.
- size Переменная типа size_t, определяет число байт в разделяемой памяти, которое динамически выделяется на блок для этого вызова в добавление к статически выделенной памяти. Данная динамически выделяемая память используется переменными, объявленными как внешние массивы. Аргумент
- size является необязательным, значение по умолчанию равно 0.
- stream Переменная типа cudaStream_t, определяет CUDA-поток (в смысле потоков на ЦПУ), ассоциированный с выполнением ядра. Механизм CUDA- потоков применяется для обеспечения асинхронной работы и использования нескольких GPU.

Данный аргумент является необязательным, для приложений, не использующих CUDA-потоки в явном виде, используется значение по умолчанию. Таким образом, обязательными частями конфигурации исполнения являются лишь первые две: количество блоков и размер блока.

2.7.7 Kernel

Kernel является самым важным элементом расширения CUDA C, которое позволяет запустить код, написанный в ядре, параллельно. С точки зрения синтаксиса C/C++ ядро вызывается довольно нетипичной конструкцией: kernel«<gridSize, blockSize, sharedMemSize, cudaStream»>().

- gridSize - размер сетки блоков, задается типом dim3, который задает количество блоков по каждой из осей OX, OY, OZ.
- blockSize - размер блока в потоках, также задается типом dim3. sharedMemSize - размер разделяемой памяти для каждого блока.
- cudaStream - переменная cudaStream_t, задающая поток, в котором будет произведен вызов.

2.8 Компиляция программы

Компиляция кода с использованием CUDA происходит под управлением программы nvcc.

Компилятор nvcc производит отделение кода, выполняющегося на GPU, от кода, выполняющегося на CPU, и компиляцию кода, выполняющегося на GPU, в бинарное представление (объекты cubin). Для кода, выполняющегося на CPU, полностью поддерживаются все возможности языка C++. Для кода, выполняющего на GPU, поддерживается подмножество языка C++ (с расширениями языка CUDA C). Как исключение из правил использования C++, указатель на void не может быть присвоен указателю на данные другого типа без явного приведения типа.

Для более точной настройки компиляции и оптимизации программы CUDA под конкретные требования и характеристики аппаратного обеспечения используются директивы компилятора nvcc. Некоторые из них:

- __noinline__ - данный спецификатор указывает компилятору не встраивать код функции (если это возможно). Тело функции должно содержаться с том же файле, откуда происходит вызов функции. Компилятор может не учитывать спецификатор __noinline__ для функций, среди параметров которых встречаются указатели, и для функций с большим списком параметров.

- #pragma unroll N - данная директива используется для того, чтобы явно управлять развертыванием того или иного цикла. Директива должна быть помещена в коде непосредственно перед циклом. Дополнительный параметр директивы (N) показывает, сколько раз цикл должен быть развернут.

- #pragma unroll 1 - данная директива указывает компилятору на то, что цикл не должен быть развернут. Если величина N не определена, то цикл полностью разворачивается, если его число повторений равно константе; в противном случае цикл не разворачивается вообще.

2.9 Планировщик задач

В NVIDIA CUDA планировщик задач, также известный как планировщик варпов (warp scheduler), играет важную роль в управлении выполнением инструкций на графическом процессоре (GPU).

Основные особенности планировщика задач варпов в NVIDIA CUDA:

- Варп: Варп представляет собой группу потоков (threads), которые исполняют одинаковую инструкцию одновременно на разных ядрах CUDA. Варпы в CUDA имеют фиксированный размер, который зависит от конкретной архитектуры GPU, например, в архитектуре NVIDIA Turing варпы состоят из 32 потоков.

- Планирование: Планировщик задач варпов определяет, какие варпы и в каком порядке будут исполняться на графическом процессоре. Он осуществляет динамическое переключение между варпами для использования ресурсов GPU максимально эффективным образом. Планировщик может выбирать варпы на основе различных факторов, таких как доступность данных, зависимости инструкций и т.д.

- Параллельность варпа: Инструкции внутри одного варпа исполняются параллельно на разных ядрах CUDA. Это позволяет достичь массовой параллельности и высокой производительности при выполнении вычислительных задач на GPU.

- Управление зависимостями: Планировщик задач варпов обрабатывает зависимости между инструкциями, чтобы обеспечить правильный порядок выполнения. Если инструкции в одном варпе имеют зависимости по данным или по управлению, планировщик может переключиться на другой варп и исполнять его, чтобы избежать простоя выполнения и максимизировать использование ресурсов.

- Многопоточность и дивергенция: Планировщик задач варпов обрабатывает множество варпов параллельно, чтобы обеспечить высокую многопоточность и эффективное использование ресурсов GPU. Однако, если инструкции внутри варпа различаются (например, из-за условных операторов), происходит дивергенция, и некоторые потоки в варпе могут ожидать, пока другие завершат выполнение.

В целом, планировщик задач варпов в NVIDIA CUDA играет ключевую роль в организации параллельного выполнения инструкций на графическом процессоре, обеспечивая высокую производительность и эффективное использование вычислительных ресурсов.

3 Суперкомпьютерный центр «Политехнический»

3.1 Состав

СКЦ располагает следующими высокопроизводительными системами [3]:

- "Политехник - РСК Торнадо";
- "Политехник - РСК Торнадо"с ускорителями NVIDIA K-40;
- "Политехник - РСК Петастрим";
- Файловая система Lustre с системой хранения данных Xyratex ClusterStor6000.

3.2 Характеристики

- 612 узлов кластера "Политехник - РСК Торнадо"(далее tornado)
  - 2 x Intel Xeon CPU E5-2697 v3 @ 2.60GHz
  - 64G RAM
- 56 узлов кластера "Политехник - РСК Торнадо"с ускорителями NVIDIA K-40
  - 2 x Intel Xeon CPU E5-2697 v3 @ 2.60GHz
  - 64G RAM
  - 2 x Nvidia Tesla K40x 12G GDDR
- 288 узлов вычислителя с ультравысокой многопоточностью "Политехник - РСК Петастрим"
  - 1 x Intel Xeon Phi 5120D @ 1.10GHz
  - 8G RAM

Все доступные узлы используют сеть 56Gbps FDR Infiniband в качестве интерконнекта. Также, на всех узлах доступна параллельная файловая система Lustre объёмом около 1 ПБ.

По умолчанию пользователям предоставляется доступ к узлам tornado. Доступ к остальным типам узлов предоставляется по запросу.

3.3 Технология подключения

Подключение к суперкомпьютеру возможно при наличии логина и ключа доступа, которые выдаются сотрудниками суперкомпьютерного центра. Ключ заменяет использование пароля для доступа к суперкомпьютеру.

Вход в SSH по публичному ключу (без пароля) очень удобен и безопасен. Сотрудниками СКЦ при регистрации создаётся пара «публичный ключ — приватный ключ». Публичный ключ копируется на компьютер с сервером SSH, то есть на компьютер, к которому будет осуществляться подключение и на котором будут выполнятся команды. Пользователю отдается приватный ключ.

Удаленное подключение к суперкомпьютеру (вычислительному кластеру) осуществляется через головной узел по протоколу SSH по адресу login1.hpc.spbstu.ru, порт 22. Для подключения необходимо воспользоваться утилитой MobaXterm.

Алгоритм действий при выполнений данной курсовой работы для подключения к СКЦ:

- От сотрудников СКЦ были получены учетные данные: логин tm4u8 и файл "spasov_ge" - ключ в формате OpenSSH;
- Была создана новая сессия в MobaXterm, полученные данные были введены в настройках сессии (Рис. 4);

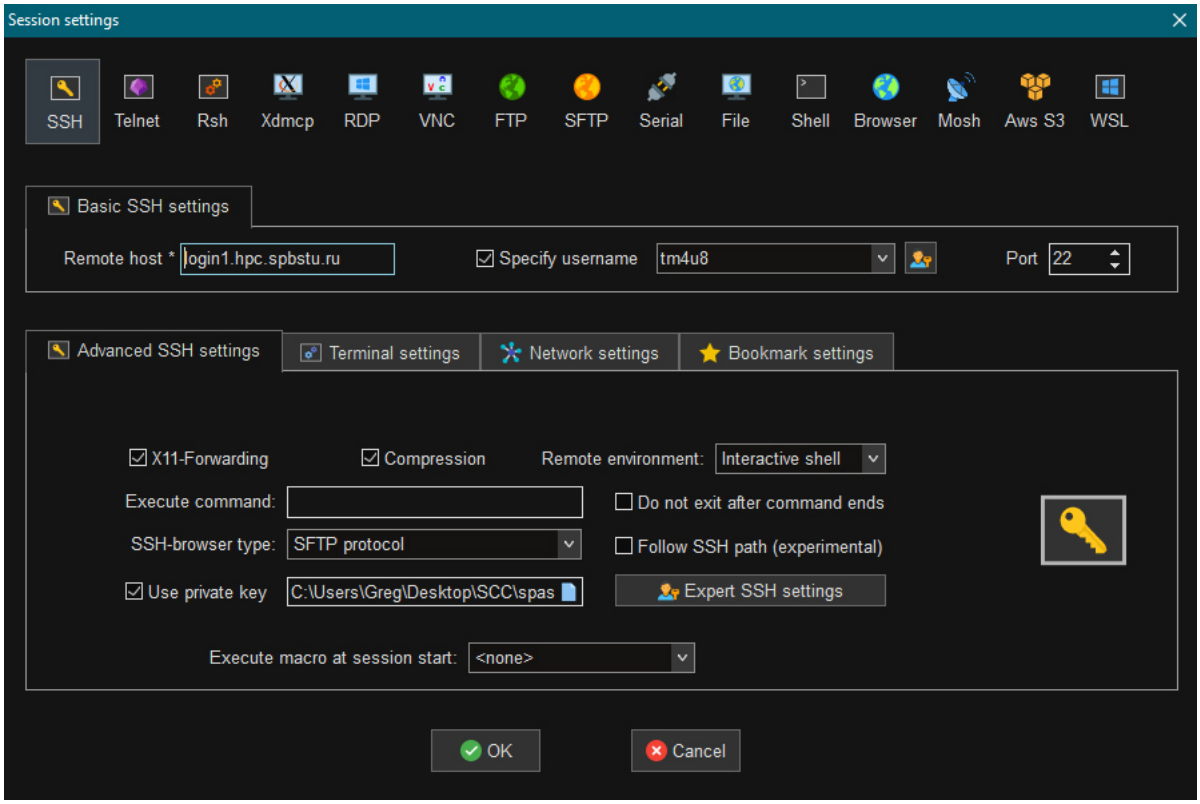

Рис. 4: Настройка сессии MobaXterm

В результате было осуществлено подключение к СКЦ по протоколу SSH (Рис. 5).

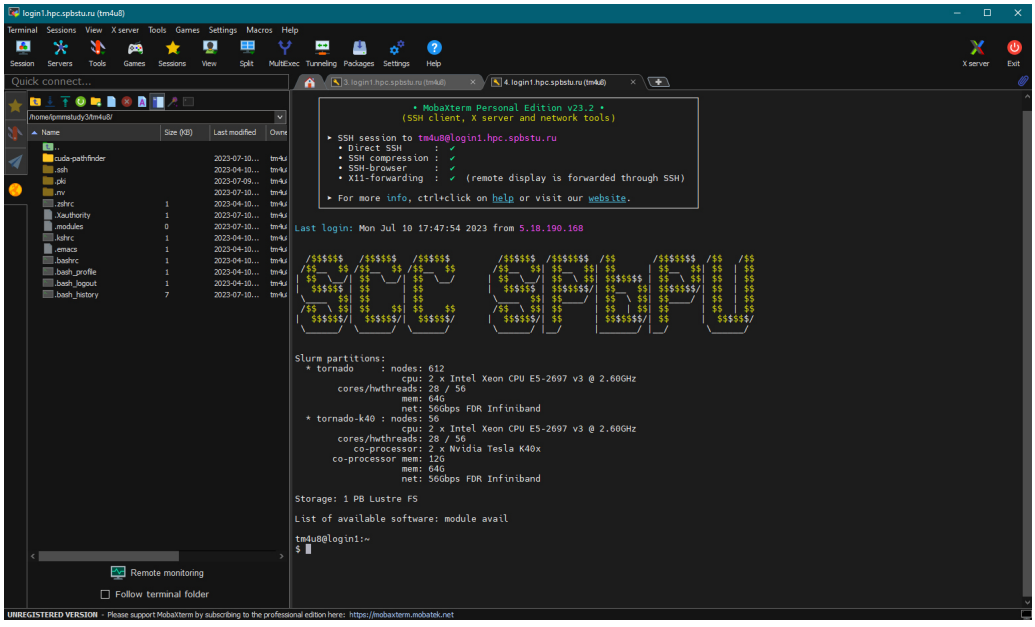

Рис. 5: Использование утилиты MobaXterm

4 Постановка решаемой практической задачи

В рамках курсовой работы была поставлена следующая практическая вычислительная задача, которую надо решить с помощью ресурсов СКЦ:

Вариант 5.1

Дано:

- массив P - полигон (nxn) целых чисел;
- P1(x1,y1), P2(x2,y2) - точки на полигоне;

Необходимо:

- построить L - траекторию минимальной длины, соединяющую P1 и P2.

Ограничения:

- L не включает точки полигона со значениями P(i,j) < 0;
- Значения P(i,j) заданы по случайному распределению в диапазоне -1, 10.

4.1 Алгоритм решения задачи

Алгоритм построения L-траектории

В качестве алгоритма построения L-траектории, соединяющую точки P1 и P2, был выбран алгоритм Ли.

Алгоритм Ли, также известный как волновой алгоритм, является алгоритмом поиска кратчайшего пути между двумя точками на планарном графе. Он основан на методе распространения волны из начальной точки к конечной точке, пройдя через все доступные пути.

Его использование хорошо подходит для параллельных вычислений по нескольким причинами:

1. Каждая ячейка в графе зависит только от своих соседей. Поэтому можно легко разделить граф на различные регионы и распределить их между несколькими вычислительными узлами.
2. Алгоритм Ли имеет хорошо определенный порядок обновления ячеек - на каждом шаге каждая ячейка обновляется только один раз. Благодаря этому легко осуществлять синхронизацию между вычислительными узлами.

Входные данные:

1. Планарный граф, представленный в виде матрицы. Каждая ячейка матрицы может быть либо свободной, либо заблокированной - принимать значение -1.
2. Начальная точка P1(x1, y1) - координаты точки, с которой начинается поиск.
3. Конечная точка P2(x2, y2) - координаты точки, которую необходимо достичь.

Выходные данные:

L-траектория между начальной P1 и конечной P2 точками, представленный в виде последовательности координат ячеек.

Ограничения:

1. Граф должен быть планарным, то есть представлять собой сетку без пересекающихся ребер.

2. Допускается только движение вверх, вниз, влево и вправо, без диагональных перемещений.

3. Граф должен быть ориентированным или невзвешенным, то есть каждая ячейка может быть либо свободной, либо заблокированной - имеет значение -1.

Шаги алгоритма:

1. Создать пустую матрицу того же размера, что и исходная матрица полигона. Эта матрица будет использоваться для хранения информации о расстоянии от начальной точки до каждой доступной ячейки.

2. Установить значение начальной точки P1 в 1 в созданной матрице.

3. Создать список, который будет хранить координаты ячеек, через которые проходит волна распространения.

4. Добавить начальную точку P1 в список.

5. Пока список не пуст или пока не достигнута конечная точка P2, выполнить следующие шаги:

   - Извлечь текущую ячейку из списка.
   - Получить список соседних ячеек (вверх, вниз, влево, вправо) текущей ячейки.
   - Для каждой соседней ячейки проверить ее значение - ячейка свободная (имеет значение 0) или заблокированная(имеет значение -1). Также проверить, что в ячейке отсутствует значение в матрице.
   - Если соседняя ячейка является свободной и не имеет значения расстояния, установить значение расстояния от начальной точки до этой ячейки, увеличивая значение расстояния текущей ячейки на 1, и добавить эту ячейку в список.
   - Повторить предыдущий шаг для каждой соседней ячейки.

6. Если значение конечной точки P2 в матрице равно 0, значит, путь не найден. В противном случае, начиная с конечной точки P2, восстановить путь, перемещаясь к соседним ячейкам с уменьшающимся значением расстояния до начальной точки P1.

7. Вернуть найденный путь - это и будет L-траектория.

На Рис.6 изображен пример работы алгоритма Ли построения L-траектории, соединяющей две точки. Красным цветом отмечены узлы графа, запрещенные для движения. Зеленым цветом - начальная точка, синим цветом - конечная точка. Фиолетовым цветом обозначена получившаяся траектория после действия алгоритма.

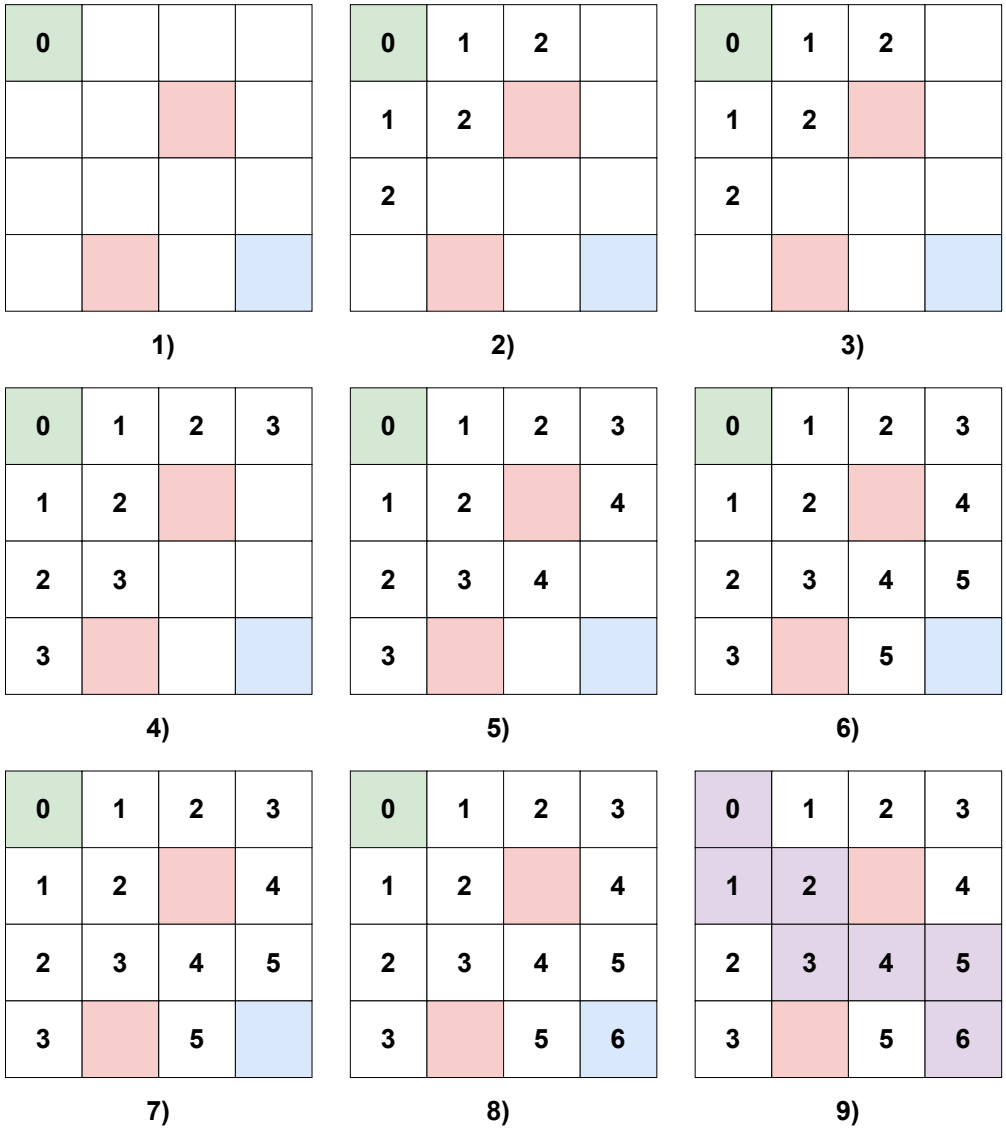

Рис. 6: Пример работы алгоритма построения L-траектории

4.2 Метод распараллеливания алгоритма

Для распараллеливания алгоритма Ли на графическом процессоре (GPU) можно использовать методы, которые позволяют одновременно вычислять значения в нескольких клетках на каждом шаге распространения волны.

Одним из таких методов является параллельное выполнение алгоритма на уровне волн (wavefront level parallelism). Этот метод основан на том, что волна распространяется по графу последовательно, но значения в клетках на фронте волны можно вычислять параллельно.

Основные шаги для распараллеливания алгоритма Ли на GPU:

1. Граф необходимо разделить на подграфы или регионы, каждый из которых будет обрабатываться отдельным блоком (thread block) на GPU. Каждый блок будет отвечать за вычисление значений в своем регионе.

2. Исходные данные графа, включая начальную и конечную точки, загружаются на требуемую (глобальную и локальную/разделяемую) память GPU.

3. Параллельное выполнение шагов алгоритма:

   (a) Выполняется первый шаг алгоритма, где вычисляются значения для клеток на расстоянии 1 от начальной точки. Каждый блок будет отвечать за вычисление значений в своем регионе.

   (b) После завершения шага блоки синхронизируются, чтобы убедиться, что значения для клеток на расстоянии 1 уже доступны.

   (c) Последовательно выполняются оставшиеся шаги алгоритма (2, 3, 4 и так далее), где каждый шаг вычисляет значения для клеток на определенном расстоянии от начальной точки. Каждый блок будет отвечать за вычисление значений в своем регионе на текущем шаге.

   (d) Блоки синхронизируются после завершения каждого шага, чтобы убедиться, что значения для клеток на текущем расстоянии уже доступны.

4. Когда алгоритм завершен и все значения расстояний вычислены, можно восстановить кратчайший путь, перемещаясь от конечной точки к начальной по ячейкам с уменьшающимся значением расстояния.

Распараллеливание алгоритма Ли на GPU позволяет эффективно использовать вычислительные ресурсы графического процессора, особенно при работе с большими графами или при необходимости обработки множества запросов на поиск пути одновременно.

4.3 Результат работы программы

На Рис. 7 - Рис. 8 представлены *примеры* графических выводов результата работы программы на полигоне размером 16x16. Буквой S обозначена начальная точка P1, буквой D обозначена конечная точка P2. Символом # обозначены контуры точки, запрещенные для движения. Символом @ обозначена получившаяся в результате работы алгоритма L-траектория.

На Рис.7 представлен пример ситуации, в которой невозможно построить L-траекторию. На Рис. 8 предоставлен пример успешного построения минимальной L-траектории.

На Рис. 9 представлен пример вывода программой результатов измерения с варьирующимися значениями размера полигона, количества блоков в сетке и количества нитей в блоке. Вывод трассировки пути был отключен.

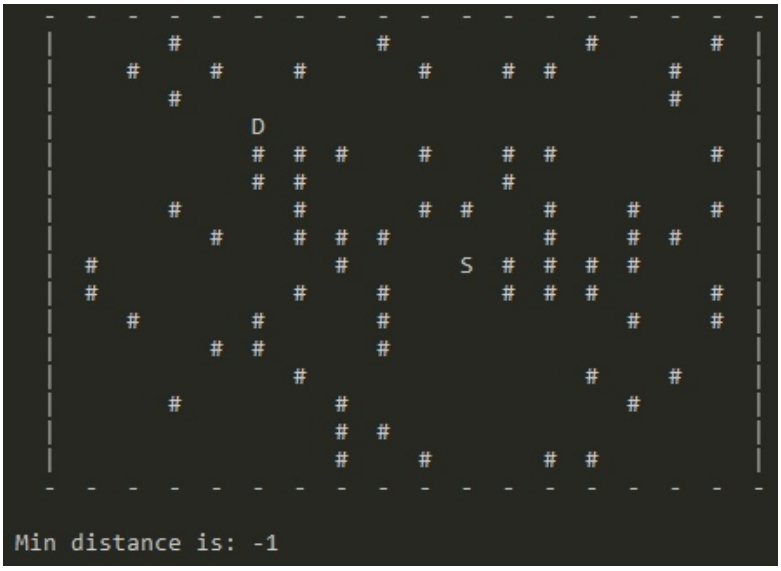

Рис. 7: Результат работы программы на поле 16x16 при отсутствии возможности построить L-траекторию

Рис. 8: Результат работы программы на поле 16x16 при успешном построении L-траектории

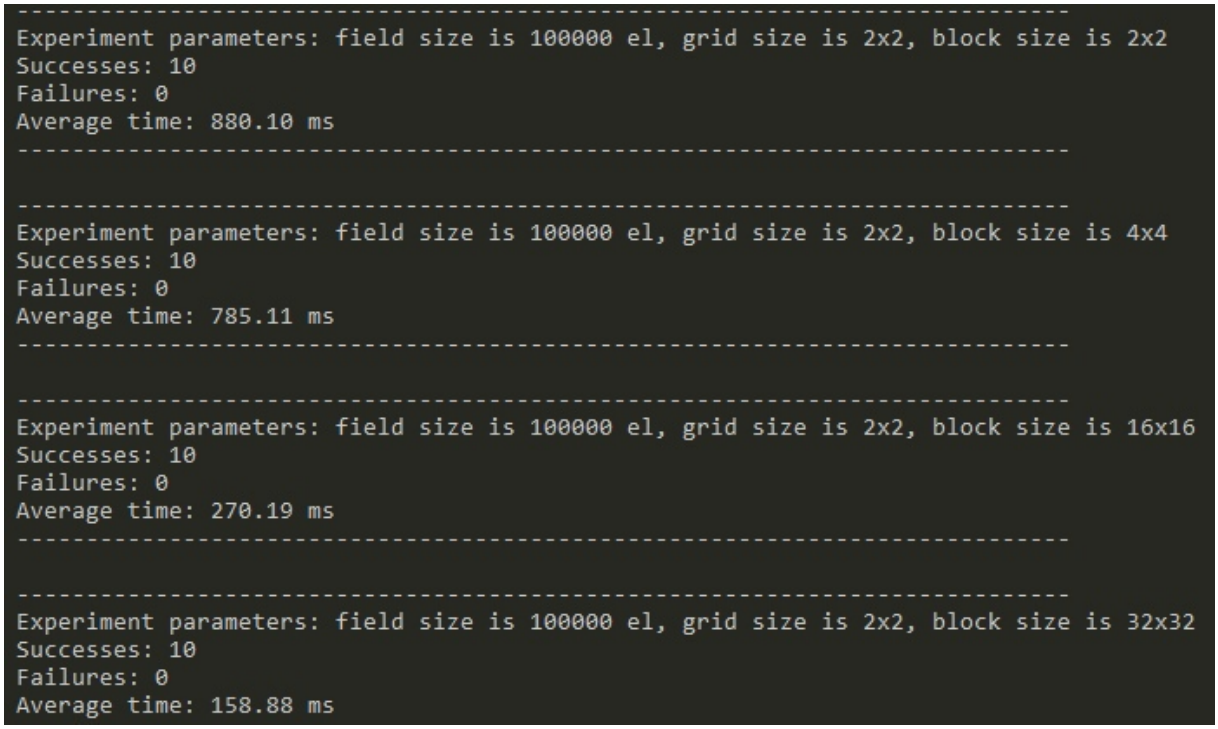

Рис. 9: Вывод программой измеренных значений

5 Описание эксперимента

Цель эксперимента: оценка зависимости времени решения задачи от объема обрабатываемых данных и параметров запуска ядер GPU.

Изменяемые параметры:

- используемая модель памяти: глобальная, /*разделяемая*/;
- размер исходного массива (размер полигона): 10^4, $5 \cdot 10^4$, 10^5;
- число блоков: 2, 4, 16, 32, 64, 128;
- число потоков в блоке: 2, 4, 16, 32, 64, 128.

Методы измерения, системные средства и инструментальные средства:
Для измерения времени исполнения программы на GPU используется механизм, предоставляемый NVIDIA CUDA для работы с событиями - cudaEvent.

Для измерения времени исполнения программы на GPU с использованием cudaEvent потребуются следующие шаги:

1. Необходимо создать два объекта cudaEvent: один для события начала (startEvent) и один для события окончания (endEvent). Это можно сделать с помощью функций cudaEventCreate().
2. Поместить вызов cudaEventRecord() с объектом startEvent перед кодом, время выполнения которого необходимо измерить. В данном эксперименте измеряется время выполнения трех функций: генерации поля, построения всех возможных траекторий пути на поле, обратной трассировки пути.
3. Разместить вызов cudaEventRecord() с объектом endEvent после выполнения кода, время выполнения которого необходимо измерить.
4. Важно добавить вызов cudaEventSynchronize() перед извлечением времени, чтобы убедиться, что все операции на GPU завершены.
5. Необходимо извлечь время между двумя событиями с помощью функции cudaEventElapsedTime() посредством передачи указателя на переменную типа float, в которую будет записан результат измерения.
6. Важно не забыть освободить память, занятую объектами cudaEvent, с помощью функций cudaEventDestroy().

После выполнения данного алгоритма переменная elapsedTime содержит время выполнения кода на GPU в миллисекундах. Время решения задачи вычислялось как сумма времени выполнения трех функций: генерации поля, построения всех возможных траекторий пути между двумя точками на поле, обратной трассировки пути.

Число испытаний:
Было проведено по 10 испытаний для каждого уникального набора данных. Суммарное количество испытаний – 960.

5.1 Результаты эксперимента

Были проведены испытания для разных параметров: используемой модели памяти, размера полигона (10^4, $5 \cdot 10^4$), 10^5), числа блоков (2, 4, 8, 32) и числа потоков в блоках (2, 4, 8, 32). При использовании числа блоков и числа потоков в блоке больших 32, программа выполнялась более 2 часов и завершалась с ошибкой. На Табл.1 - Табл.6 представлены результаты времени выполнения программы.

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	30.04	36.64	9.48	6.11
4	14.77	8.76	3.14	1.38
16	1.78	1.01	0.76	0.94
32	1.45	0.65	0.47	1.00

Таблица 1: Время работы алгоритма на графе 10^4 клеток, глобальная память (мс)

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	391.70	371.25	136.75	64.72
4	89.16	55.00	30.67	87.32
16	25.34	9.39	3.84	6.74
32	6.21	7.88	2.28	2.57

Таблица 2: Время работы алгоритма на графе $5 \cdot 10^4$ клеток, глобальная память (мс)

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	327.71	444.24	182.91	194.31
4	204.39	202.93	69.53	150.67
16	63.46	26.75	14.29	30.91
32	11.32	13.71	9.03	6.20

Таблица 3: Время работы алгоритма на графе 10^5 клеток, глобальная память (мс)

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	21.76	39.22	10.01	6.66
4	6.19	4.47	3.68	1.63
16	2.17	1.12	0.69	0.86
32	1.27	0.66	0.45	1.95

Таблица 4: Время работы алгоритма на графе 10^4 клеток, разделяемая память (мс)

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	571.13	375.41	120.64	59.97
4	98.67	54.19	30.73	56.98
16	17.69	8.36	4.25	5.89
32	7.44	6.18	1.54	2.59

Таблица 5: Время работы алгоритма на графе $5 \cdot 10^4$ клеток, разделяемая память (мс)

	Размер блоков, MxM			
Размер гридов, NxN	2	4	16	32
2	880.10	785.11	270.19	158.88
4	212.56	87.48	69.53	151.37
16	65.75	37.45	17.05	42.50
32	14.80	17.93	7.95	8.86

Таблица 6: Время работы алгоритма на графе 10^5 клеток, разделяемая память (мс)

5.2 Анализ результатов эксперимента

При увеличении размерности сеток, среднее время выполнения испытания уменьшается, что соответствует теоретическим обоснованиям.

При увеличении количества нитей в блоке с 2 до 16, как с использованием глобальной памяти, так и разделяемой, время выполнения испытания уменьшается до минимального; но при последующем увеличении нитей до 32 время снова увеличивается – это объясняется накладными расходами на создание и вызов нитей, не используемых программой (выходящих за полигон). Таким образом, можно сделать вывод о том, что увеличение количества нитей в блоке не всегда приводит к увеличению производительности; существует оптимальное количество нитей в блоке, зависящее от размера полигона.

При использовании разделяемой памяти вместо глобальной, среднее время выполнения программы почти не изменяется. Это можно объяснить особенностью реализации программы: количество используемых локальных переменных в функциях, исполняемых на графическом ускорителе, мало; компиляция программы с данными переменными, отмеченными модификатором разделяемой памяти, не увеличивает производительность этих функций.

Заключение

В рамках курсовой работы была изучена технология параллельного программирования на основе архитектуры Nvidia CUDA.

Для задачи нахождения минимальной траектории на полигоне между двумя точками был разработан алгоритм, реализованный программно на языке CUDA C. Программа была запущена на ресурсах суперкомпьютерного центра «Политехнический». Для запуска использовался узел типа «Торнадо» с видеокартой NVIDIA Tesla K40X. Запуск программы проводился на одном узле с использованием одной видеокарты.

Количество комбинаций данных размера полигона, количества блоков в сетке и количества нитей в блоке равняется 96 (48 для экспериментов с использованием глобальной памяти, 48 – с использованием разделяемой памяти). Каждый эксперимент был запущен 10 раз для получения среднего времени эксперимента. Таким образом, всего было запущено 960 экспериментов.

Результаты экспериментов подтверждают теоретические данные: при увеличении размерности сеток, среднее время выполнения испытания уменьшается. Увеличение количества нитей в блоке не всегда приводит к увеличению производительности; существует оптимальное количество нитей в блоке, зависящее от размера полигона. При использовании разделяемой памяти вместо глобальной, среднее время выполнения реализованной программы почти не изменяется. При использовании числа блоков и числа потоков в блоке больших 32, программа выполнялась более 2 часов и завершалась с ошибкой, что может говорить либо об особенностях используемого графического ускорителя, либо о неоптимальной программной реализации алгоритма для работы с такими характеристиками сеток и блоков.

Программа компилировалась на сервере СКЦ «Политехнический» с использованием компилятора NVCC 10.1.

Список источников

[1] Эдвард Кэндрот, Джейсон Сандерс «Технология CUDA в примерах. Введение в программирование графических процессоров» – ДМК-Пресс, 2018 г. – 232 с.

[2] CUDA Programming Guide 12.1 (`http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`).

[3] Краткое руководство пользователя вычислителей «Политехник - РСК Торнадо» и «Политехник - РСК Петастрим».