МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

Отчёт по по дисциплине «Курсовое проектирование по управлению ресурсами суперэвм» Решение задачи нахождения алгебраических дополнений элементов матрицы

Студент,
группы 5130201/20102 _____ Салимли А. М.

Преподаватель _____ Курочкин М. А.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение

Многие задачи, возникающие на практике, требуют большого объема вычислений. Одним из вариантов решения сложных вычислительных задач является использование параллельного программирования. За последние несколько десятилетий стало очень распространено вычисление с помощью графических ускорителей — устройств с массивно-параллельной архитектурой. Производить вычисления общего назначения можно на видеокартах архитектуры Nvidia CUDA. Сегодня спроектированы и испытаны многие компьютеры, которые используют в своей архитектуре тот или иной вид параллельной обработки данных. Сложность работы программирования заключается в координации используемых ресурсов. Одним из примеров массивных вычислительных систем является суперкомпьютерный центр «Политехнический». Часть узлов этого суперкомпьютера оборудована графическими ускорителями Nvidia Tesla K40X.

1 Постановка задачи

1.1 Задачи лабораторной работы

В рамках данной работы необходимо изучить технологию параллельного программирования с использованием архитектуры Nvidia CUDA.

Также необходимо ознакомиться с принципом использования ресурсов суперкомпьютерного центра «Политехнический» для решения прикладной задачи.

Необходимо написать программу для решения поставленной задачи с использованием технологии Nvidia CUDA и провести исследование зависимости времени выполнения программы от количества используемых ресурсов.

В рамках курсовой работы необходимо написать программу для подсчета алгебраических дополнений заданной матрицы.

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;

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.

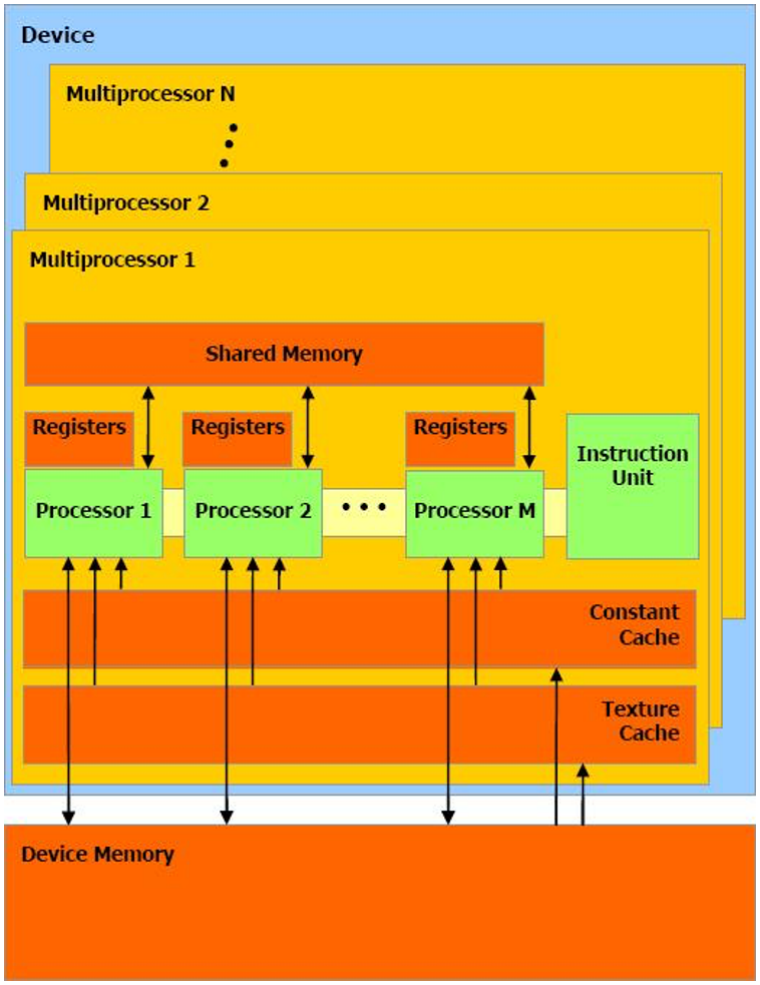

Рис. 1. Архитектура устройства

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.

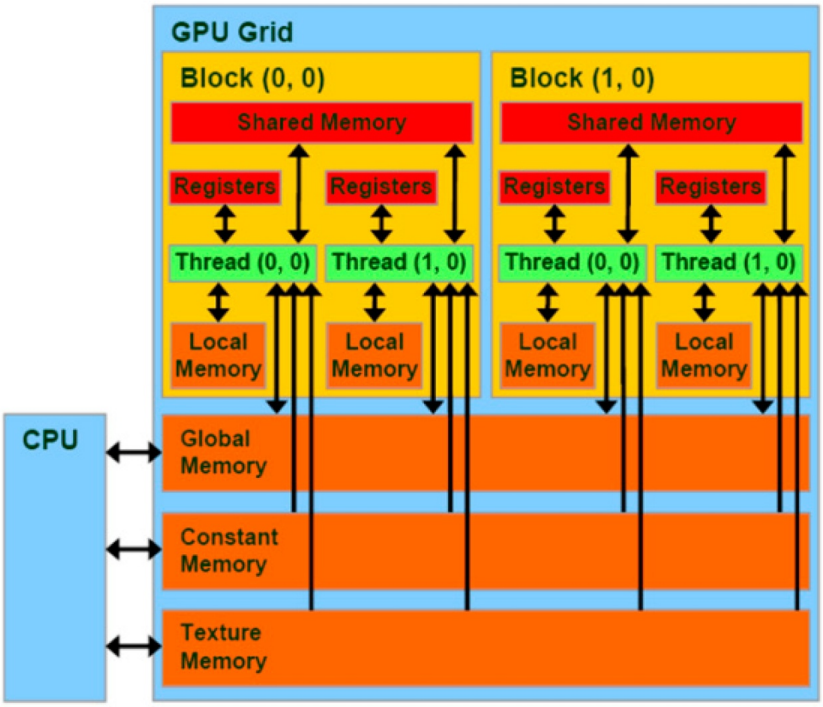

Рис. 2. Виды памяти

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).

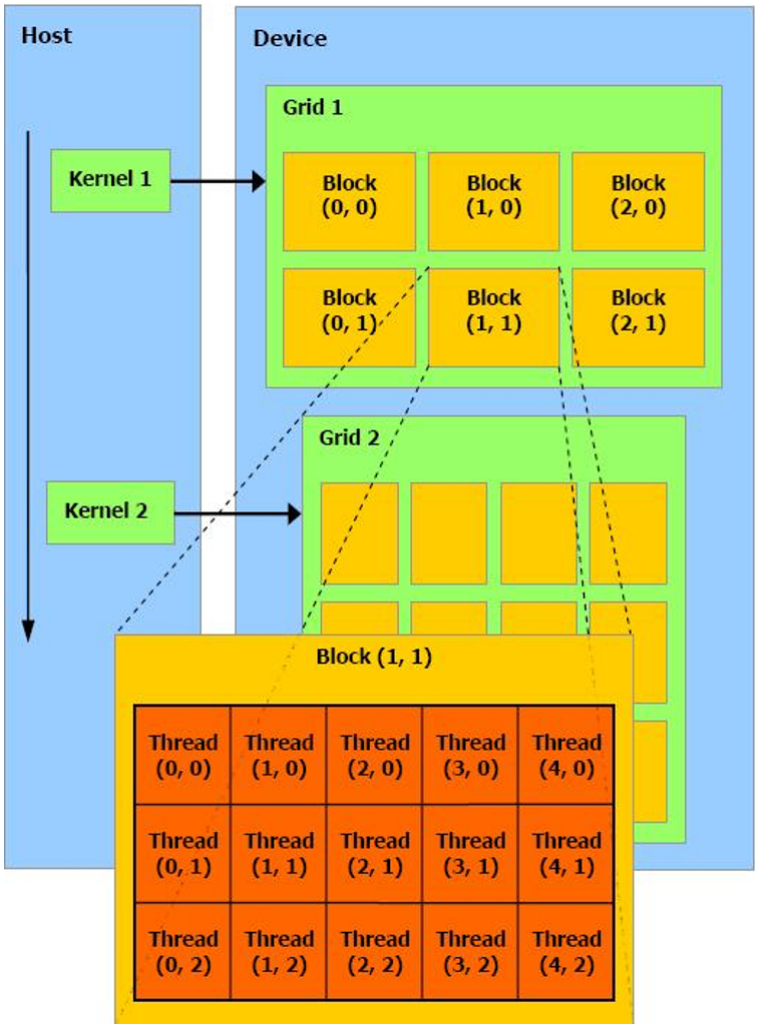

Рис. 3. Организация потоков

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

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

обращаются к памяти в произвольном порядке или часто совершают прыжки в адресном пространстве, эффективность кэширования варпа может снизиться.

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.

3 Суперкомпьютерный центр «Политехнический»

3.1 Состав

Суперкомпьютерный центр «Политехнический» состоит из узлов трех типов:

- 668 узлов кластера «Политехник - РСК Торнадо»;
- 288 узлов вычислителя с ультравысокой многопоточностью «Политехник - РСК ПетаСтрим».
- 64 узла кластера «Политехник - NUMA».

3.2 Характеристики

Политехник - РСК Торнадо

Кластер содержит узлы двух типов:

- 612 узлов с прямым жидкостным охлаждением серии «Торнадо»(производитель РСК Технологии РФ), имеющие каждый два CPU Intel Xeon E5-2697 v3 (14 ядер, 2.6 ГГц) и 64 ГБ оперативной памяти DDR4;
- 56 узлов с прямым жидкостным охлаждением серии Tornado содержащие каждый два CPU Intel Xeon E5-2697 v3 и два ускорителя вычислений NVIDIA Tesla K40X, 64 ГБ оперативной памяти DDR4.

Политехник - РСК ПетаСтрим

Кластер содержит узлы двух типов:

- 288 однопроцессорных узлов с пиковой производительностью 1 ТФлопс каждый;
- 17280 многопоточных ядер общего назначения (69120) потоков, поддерживающих векторную обработку данных посредством аппаратно реализованных инструкций FMA (Fused Multiply-Accumulate);
- оперативная память узла - 8 ГБ, GDDR5; суммарный объём оперативной памяти системы 2304 ГБ;
- пропускная способность между двумя узлами модуля системы на тесте MPI OSU или Intel MPI Benchmarks не менее 6 ГБ/с.

Политехник - NUMA

Кластер содержит узлы двух типов:

- 64 вычислительных узла, каждый из которых включает:
  - 3 CPU AMD Opteron 638;
  - Адаптер NumaConnect N313-48;

– 192 ГБ оперативной памяти;

- 192 процессора;
- 3072 ядер x86

3.3 Технология подключения

Для подключения зарегистрированного пользователя к СКЦ необходимо использовать SSH клиент. С помощью него получается доступ к удаленному терминалу для работы с ресурсами СКЦ.

В рамках работы была использована следующая технология подключения:

- Были получен приватный ключ от администрации СКЦ в виде файла.
- При помощи команды ssh был произведен вход: ssh -v tm3u12@login1.hpc.spbstu.ru -i /.ssh/id_rsa, где tm3u12 - логин, login1.hpc.spbstu.ru - адрес, id_rsa - приватный ключ.
- Чтобы переслать файлы, использовалась команды: scp -r "kernel.cu"`tm3u12 @login1.hpc.spbstu.ru:home/kernel.cu`, где "kernel.cu путь до файла на локальном компьютере, tm3u12 - логин, login1.hpc.spbstu.ru - адрес, home/kernel.cu - пусть сохранения файла на СКЦ.

4 Постановка решаемой практической задачи

Дано:

- Матрица A размером $n \times n$ вещественных чисел;

Требуется:

- Найти B — все алгебраические дополнения матрицы A;

Ограничения:

- Числа в матрице A не должны превышать 10^5;
- Числа в матрице A не должны быть меньше -10^5.
- n не больше 100

5 Алгоритм решения задачи

Для решения задачи выполняются следующие действия:

1. Из исходной матрицы A удаляются строка i и столбец j.
2. Далее происходит подсчет определителя для полученной матрицы d_{ij} размером $(n-1) \times (n-1)$.
3. Число $(-1)^{i+j} d_{ij}$ записывается в матрицу B в строку i в столбец j.
4. Данные действия повторяются до тех пор, пока все алгебраические дополнения не будут найдены.

На Рис. 4 приведен пример вычисления алгебраического дополнения для элемента матрицы:

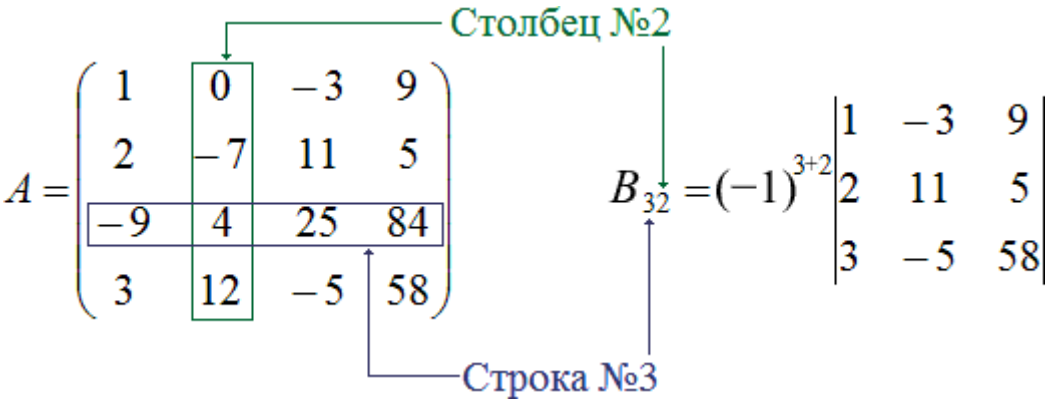

Рис. 4. Вычисления алгебраического дополнения для произвольной матрицы

5.1 Метод распараллеливания алгоритма

При распараллеливании алгоритма каждое алгебраическое дополнение для строки i и столбца j считается на отдельном потоке. Для этого в функции запуска ядра каждому потоку выдаются свои значения i и j в соответствии с его номером. Номер потока вычисляется следующим образом:

$$threadId = threadIdx.x + blockIdx.x * blockDim.x$$

где threadIdx.x - номер потока в блоке по оси x, blockIdx.x - номер блока по оси x, blockDim.x - количество потоков в блоке по оси x.

Чтобы упростить работу с матрицами, они были представлены в виде векторов. Чтобы получить доступ к определенному элементу этого вектора, происходит преобразование столбцов и строк матрицы в линейный индекс:

$$index = row * cols + col$$

где index - конечный индекс элемента в векторе, row - номер строки в матрице, col - номер столбца в матрице, cols - количество столбцов в матрице.

Чтобы перейти от индексации вектора к индексации матрицы, производятся следующие преобразования:

$$row = \frac{index}{cols}$$

$$col = index \ \ mod \ \ cols,$$

где index - конечный индекс элемента в векторе, row - номер строки в матрице, col - номер столбца в матрице, cols - количество столбцов в матрице.

По этим двум формула происходит выделение индексов i и j, по которым будут убраны строка и столбец.

Чтобы поток мог произвести повторные вычисления, если потоков недостаточно для полного покрытия матрицы, то происходит смещение по следующий формуле:

$$threadId = threadId + blockDim.x * gridDim.x$$

где threadId - номер потока в блоке по оси x, blockDim.x - количество потоков в блоке по оси x, gridDim.x - количество потоков по оси x.

После вычисления алгебраического дополнения поток записывает его в матрицу $n \times n$ в строку i и в столбец j.

6 Описание эксперимента

В этом разделе выполняется исследование времени решения задачи при изменении следующих параметров:

- Размеры матрицы: 32×32, 75×75, 100×100;
- Количество блоков: 1, 10, 100, 1000, 10000;
- Количество потоков: 1, 10, 100, 1000;
- Используемая память: глобальная, глобальная и константная.

Для измерения времени вычислений использовался модуль time.h.
Измерение происходило следующим образом:

1. Происходила генерация матрицы;
2. Выделение данных для host;
3. Запись времени начала эксперимента;
4. Инициализация памяти на device;
5. Проведение вычислений;
6. Очистка памяти;
7. Запись времени конца эксперимента;
8. Вывод разницы между временем начала эксперимента и его конца.

7 Анализ результатов

В таблицах 1, 2, 3 приведены результаты измерения времени в милисекундах для глобальной памяти для размеров матрицы 32×32, 75×75, 100×100 соответственно. Знаком — отмечены те случаи, при которых произошла нехватка памяти. В таблицах 4, 5, 6 приведены результаты измерения времени в милисекундах для глобальная и константная памяти для размеров матрицы 32×32, 75×75, 100×100 соответственно.

Таблица 1. Результаты измерения времени исполнения программы для матрицы 32×32 и глобальной памяти

Числов блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	4440	450	50	20	20
10	900	100	40	10	20
100	210	60	30	50	40
1000	120	100	120	110	100

Таблица 2. Результаты измерения времени исполнения программы для матрицы 75×75 и глобальнной памяти

Числов блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	217020	22110	2380	1080	940
10	44810	4610	—	—	—
100	10140	—	—	—	—
1000	4630	—	—	—	—

Таблица 3. Результаты измерения времени исполнения программы для матрицы 100×100 и глобальнной памяти

Числов блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	1187990	119440	11950	5860	5060
10	245640	24780	—	—	—
100	62360	—	—	—	—
1000	—	—	—	—	—

Таблица 4. Результаты измерения времени исполнения программы для матрицы 32×32 и глобальная и константная памяти

Число блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	4030	390	50	20	20
10	840	100	20	20	20
100	220	50	50	40	50
1000	110	110	110	100	110

Таблица 5. Результаты измерения времени исполнения программы для матрицы 75×75 и глобальная и константная памяти

Число блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	200490	20310	2070	990	860
10	43150	4420	—	—	—
100	10380	—	—	—	—
1000	4560	—	—	—	—

Таблица 6. Результаты измерения времени исполнения программы для матрицы 100×100 и глобальная и константная памяти

Число блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	1101650	111320	11190	5500	4730
10	237750	24120	—	—	—
100	56750	—	—	—	—
1000	—	—	—	—	—

Как можно увидеть, что при максимальном количестве блоков вычисление алгебраических дополнений матрицы показываются минимальный показатель затраченного времени, но увеличении потоков время выполнения возрастает. Это происходит из-за того, что выделено слишком большое число потоков, которые не выполняют никаких вычислений.

Из роста количества потоков видно, что при их большом количестве время возрастает даже для одного блока. Таким образом, можно сделать вывод, что произведение числа блоков на число потоков должно быть приблизительно равно

количеству элементов в матрице. При этом мы получим самое минимальное время, если количество блоков будет больше, чем количество потоков.

Были выделены лучшие конфигурации для матриц с разными размерами:

- Для матрицы с размером 32×32 - 1000 блоков, 100 потоков;
- Для матрицы с размером 75×75 - 10000 блоков, 1 поток;
- Для матрицы с размером 100×100 - 10000 блоков, 1 поток;

Для константной памяти время вычисления меньше, чем время вычисления без нее, но на небольшую долю. В среднем время выполнения при использовании глобальной памяти больше время выполнения при использовании глобальной и константной памяти на 5%.

Также появляется одна проблема. При больших размерах исходной матрицы(75×75 и 100×100) и потоков на одном блоке происходит нехватка памяти. Дело в том, что каждая такая матрица копируется внутри каждого потока, и при большом количестве потоков в блоке не хватает памяти.

На Рис. 5 отображена зависимость времени от разного числа потоков в определенной выборке блоков.

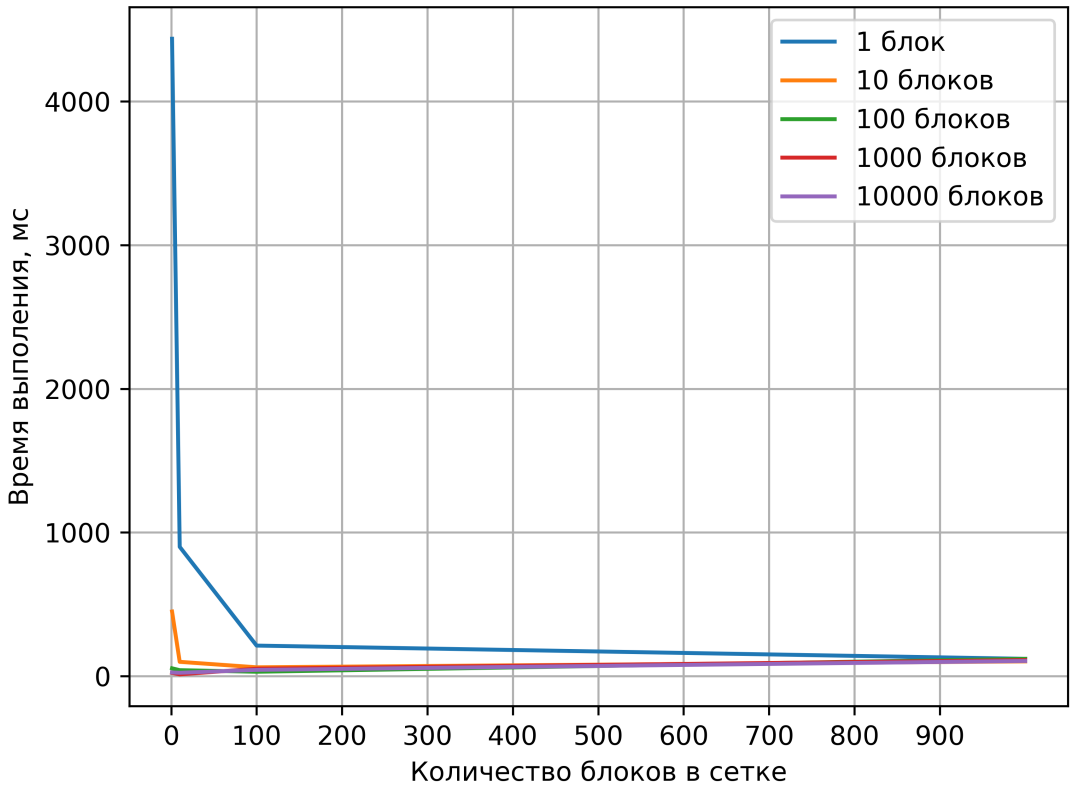

Рис. 5. Зависимость времени от разного числа потоков в определенной выборке блоков

Заключение

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

Для задачи подсчета фигур на изображении был разработан параллельный асинхронный алгоритм, алгоритм был реализован 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.

Для программы было измерено время работы при различной степени распараллеливания. Полученные результаты согласовались с теоретической оценкой максимального количества потоков. Использование оптимальной конфигурации позволило уменьшить время выполнения в 400 раз относительно наихудшей конфигурации для матрицы 32×32, в 230 раз для 75×75 и в 235 раз для 100×100.

Реализация алгоритма с использованием константной памяти увеличило время выполнение в среднем на 5%.

Измерение времени исполнения в зависимости от размера матрицы показало линейную зависимость.

В рамках курсовой работы была написана программа размером около 300 строк. Работа на СКЦ «Политехнический» шла три недели, за это время было сделано примерно 50 авторизаций и порядка 100 запусков задач на исполнение.

Для сборки использовался компилятор NVCC версии 11.6u2.

Список литературы

[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».