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 миноров матрицы

Студент,
группы 5130201/20102 _____ Губковский Д. В.

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 миноров заданной матрицы.

2 Аппаратно-программная платформа Nvidia CUDA

CUDA (Compute Unified Device Architecture) — это архитектура параллельных вычислений от Nvidia, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров) фирмы Nvidia.

2.1 Архитектура Nvidia CUDA

Видеокарты Nvidia CUDA имеют иерархическую архитектуру:

- Процессор GPU представляет собой массив потоковых процессоров (Streaming Processor Array);
- Потоковый процессор состоит из кластеров текстурных процессоров (Texture Processor Clusters);
- Текстурные процессоры состоят из набора мультипроцессоров (Streaming Multiprocessor);
- Мультипроцессоры содержат несколько потоковых процессоров (Streaming Processors) или ядер (CUDA cores).

Планирование выполнения команд происходит при помощи GigaThread Engine. Он распределяет блоки потоков по мультипроцессорам. Общий вид GPU кардинально не меняется при переходе от одной микроархитектуры к другой, изменяется размер и скорость L2 кэша.

В архитектуре Nvidia CUDA применяется SIMT (Single Instruction Multiple Thread) модель исполнения. Это модель является комбинацией из MIMD (Multiple Instruction Multiple Data) и SIMD (Single Instruction Multiple Data). Вычисляемая задача состоит из сетки блоков, а блок состоит из нитей (thread), при исполнении нити разбиваются на варпы — группы по 32 нити. Все нити варпы выполняют в одно время одну и ту же инструкцию.

2.2 Вычислительные возможности Nvidia CUDA

Видеокарты Nvidia CUDA разных микроархитектур обладают разным количеством ядер разного назначения. В этом разделе приведены краткие сведения о вычислительных возможностях различных поколений Nvidia CUDA.

Микроархитектура Fermi

В каждом потоковом процессоре:

- 32 ядра CUDA для выполнения операций с целыми числами и с числами с плавающей точкой;
- 16 ядер загрузки/выгрузки данных;

- 4 блока специального назначения (для вычисления сложных арифметических функций).

Для распределения задач используется два планировщика варпов. Для хранения используется:

- Регистровый файл, 128KB;
- L1-cache, 16KB/48KB;
- Разделяемая память, 48KB/16KB.

Микроархитектура Kepler

В каждом потоковом процессоре:

- 192 ядра CUDA для выполнения операций с целыми числами и с числами с плавающей точкой;
- 64 блока для обработки чисел с двойной точностью;
- 32 ядер загрузки/выгрузки данных;
- 32 блока специального назначения (для вычисления сложных арифметических функций).

Для распределения задач используется четыре планировщика варпов. Для хранения используется:

- Регистровый файл, 256KB;
- L1-cache, 16KB/48KB/32KB;
- Разделяемая память, 48KB/16KB/32KB;
- Константная память (Read-Only Data Cache), 48KB.

Микроархитектура Maxwell

Каждый потоковый процессор в этой архитектуре состоит из четырех блоков. В каждом блоке:

- Свой планировщик варпов;
- Свой регистровый файл, 64KB;
- 32 ядра CUDA;
- 8 ядер загрузки/выгрузки данных;
- 8 блока специального назначения (для вычисления сложных арифметических функций).

Общими для четырех блоков являются L1/Texture кэш и разделенная память. В отличие от предыдущих микроархитектур shared memory теперь является отдельным блоком размером 96KB.

Микроархитектура Pascal

В плане архитектуры Pascal отличается от Maxwell тем, что каждый потоковый процессов содержит 2 блока.

Микроархитектура Tesla

В Volta SM отказались от CUDA ядер и расщепили их на отдельные блоки (INT + FP32), что позволило использовать их одновременно и, соответственно, увеличило общую производительность. Также в состав включили абсолютно новый компонент под названием Tensor Core, нацеленный на увеличение производительности для глубокого обучения. По аналогии с Maxwell в SM есть 4 одинаковых блока, каждый из которых содержит планировщик варпов, один модуль отправки команд, 8 ядер двойной точности, 32 целочисленных ядра, 32 ядра одинарной точности, 2 Tensor Core, 8 LD/ST и 1 SFU. Регистровый файл для каждого блока равен 64KB. Для Volta L1 кэш и разделяемая память общие для всех блоков и снова объединены общим объемом в 128KB, где под разделяемую память можно отвести до 96 KB.

Вычислительные возможности Nvidia Tesla K40

Программа в рамках данной работы исполнялась на видеокарте Nvidia Tesla K40. Ниже приведены вычислительные возможности этой видеокарты:

- Размерность сетки блоков. Максимальное число измерений: 3; максимальное число блоков по каждому измерению: $2^{31} - 1$, 65535, 65535;
- Размерность блока. Максимальное число измерений: 3; максимальное число нитей по каждому измерению: 1024, 1024, 64; максимальное число нитей в блоке: 1024;
- Размер варпа: 64;
- Размер регистрового файла. На блок: 64000 регистров; на нить: 255 регистров;
- Размер разделяемой памяти на мультипроцессор: 48K;
- Размер константной памяти на блок: 64KB.

2.3 Потоковая модель

Вычислительная архитектура CUDA основана на понятии мультипроцессора и концепции SIMT (Single Instruction Multiple Threads). При выполнении многопоточной программы на видеокарте CUDA, все потоки разделяются на блоки, а внутри блоков на варпы, где все потоки выполняют одну и ту же инструкцию. Группа блоков выполняется на потоковом процессоре, распределением задач занимается планировщик. Программа, выполняющаяся на нескольких блоках одновременно называется ядром (kernel).

Особенностью архитектуры CUDA является блочно-сеточная организация, необычная для многопоточных приложений (Рис. 1). При этом драйвер CUDA самостоятельно распределяет ресурсы устройства между потоками.

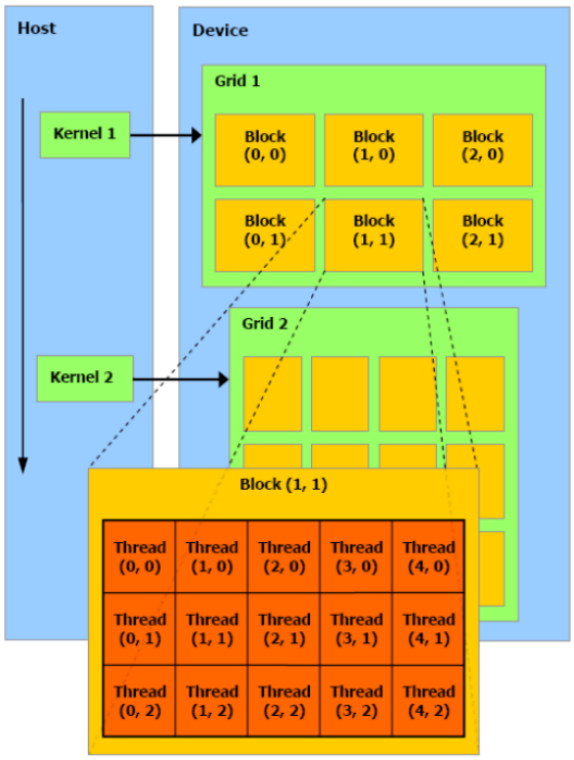

Рис. 1. Организация потоков

2.4 Устройство памяти

Видеокарта имеет собственную оперативную глобальную память, отдельную от оперативной памяти CPU (на хосте). При выполнении кода на видеокарте (на устройстве) обращение может происходить только к памяти на видеокарте. Для перемещения данных с хоста на устройство и обратно используются служебные функции, вызываемые с хоста. Помимо глобальной оперативной памяти на каждом мультипроцессоре есть свой кэш, своя текстурная, константная, разделяемая память и тд.

2.5 Модели памяти

В CUDA выделяют шесть видов памяти:

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

На Рис. 2 представлено устройство памяти.

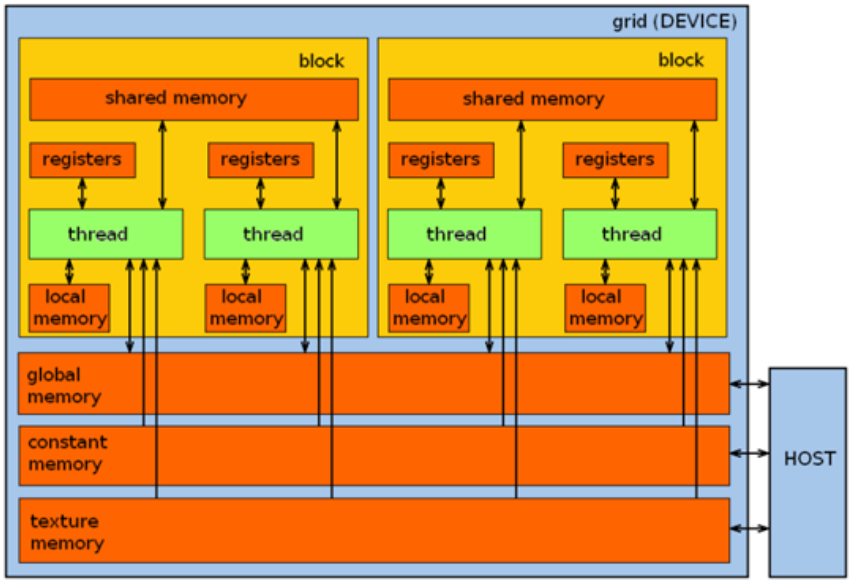

Рис. 2. Устройство памяти

Регистры

На один мультипроцессор доступно 8192 32-разрядных регистров. Они распределяются между нитями в этом потоке. Обращение к этой памяти самое быстрое.

Глобальная память

Глобальная память имеет большой объем. Она поддерживает произвольный доступ для всех мультипроцессоров, а также запись и чтение с хоста. Однако, эта память очень медленная и не кэшируется, поэтому рекомендуется сократить количество обращений к этой памяти.

Локальная память

Это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная — такая же, как и глобальная.

Разделяемая память

Разделяемая память — это некэшируемая, но быстрая память. Ее и рекомендуется использовать как управляемый кэш. На один мультипроцессор доступно всего 16KB разделяемой памяти. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Разделив это число на количество задач в блоке, получим максимальное количество разделяемой памяти, доступной на один поток.

Константная память

Константная память — это тип памяти, который хранит неизменяемые данные, доступные на уровне сетки. Ее размер 64KB, физически она не отделена от

глобальной памяти, но, используя системы кэшей и механизму широкого вещания (broadcast), она может обеспечить прирост производительности за счет сокращения трафика между процессором и памятью.

Текстурная память

Текстурная память — это тип памяти, который похож на константную память, поскольку через текстурный блок разрешены запросы только для чтения. Физически текстурная память не отделена от глобальной. Как и константная память, позволяет увеличить производительность за счет системы кэшей. Отличительной особенностью является оптимизация текстурного кэша для двумерной пространственной локальности (данные расположены рядом в двумерном пространстве).

2.6 Модель вычислений на GPU

Программа, запускаемая на GPU с Nvidia CUDA, называется ядром (kernel). Ядро запускается одновременно на сетке из блоков. Каждый из блоков, как было сказано раньше, состоит из нескольких нитей. Количество блоков и потоков задается при вызове ядра. В каждой нити выполняется один и тот же код. Добиться разного поведения в нитях возможно при помощи информации о номере блока и нити, которая доступна для каждой нити.

Все команды, пришедшие на GPU, будут исполняться в порядке общей для всех потоков очереди. Последовательность исполнения может сгладить планировщик, который может запустить одновременно копирование с хоста, на хост и исполнение ядра. А если ядро использует меньше 50% мощности GPU, то запустить параллельно следующее ядро из

другого потока, если оно готово к запуску. Так, для оптимизации времени выполнения задач в разных потоках нужно учитывать, что на GPU команды будут исполнятся в том порядке, в котором их вызвали в хост коде. Это значит, что не всегда лучше заполнить задачами один поток, затем второй и т.д. Скорее более оптимальным подходом будет равномерный запуск задач по всем потокам. Для этого можно сначала заполнить один поток командами, а затем другой. Тогда команды второго потока будут ожидать окончания выполнения команд первого потока, или, что более вероятно, начала копирования в память хоста из первого потока. Другой вариант, когда команды будут распределяться по потокам поочередно, т.е. в первый поток отправляется первая команда, затем во второй отправляется также первая, затем в первый вторая, и т.д. В результате такого равномерного распределения можно добиться улучшения производительности за счет умения планировщика одновременно запускать операции копирования и ядра.

2.7 Планировщик задач

Поток (stream) в CUDA — логическая последовательность зависимых асинхронных операций, независимая от операций в других потоках. Потоки позволяют запускать CUDA команды на GPU в порядке, определенном в контексте одного потока. С точки зрения GPU потоков не существует, и все команды, пришедшие на GPU, будут исполняться в порядке общей для всех потоков очереди, знание этой особенности может помочь при оптимизации. Последовательность исполнения может сгла-

дить планировщик, который может запустить одновременно копирование с хоста, на хост и исполнение ядра. При вызове ядра можно указать поток, в который будет добавлено это ядро. По умолчанию все ядра добавляются в поток 0, который является синхронным с хостом.

2.8 Компиляция программы

Программа для видеокарт Nvidia CUDA пишется на основе других языков, в частности используется расширение языка C. Оно называется CUDA C. Для сборки программы используется компилятор nvcc, который входит в пакет инструментов разработчика. Этот пакет, а также библиотеку CUDA можно скачать с сайта Nvidia.

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 tm3u8@login1.hpc.spbstu.ru -i /.ssh/id_rsa, где tm3u8 - логин, login1.hpc.spbstu.ru - адрес, id_rsa - приватный ключ.
- На самом СКЦ был изменен приватный ключ, чтобы был доступ к перессылке файлов.
- Чтобы переслать файлы, использовалась команды: scp -r "E:/documents/university/6_sem/C
  где "E:/documents/university/6_sem/СуперЭВМ/Cuda/Lab/Lab/kernel.cu путь до файла на локальном компьютере, tm3u8 - логин, login1.hpc.spbstu.ru - адрес, home/kernel.cu - пусть сохранения файла на СКЦ.

4 Постановка решаемой практической задачи

Дано:

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

Требуется:

- Найти все миноры матрицы;

Ограничения:

- Числа в матрице не должны превышать 100000;
- Числа в матрице не должны быть меньше -100000.

5 Алгоритм решения задачи

Для решения задачи выполняются следующие действия:

1. Из исходной матрицы удаляются строка x и столбец y.
2. Далее происходит подсчет определителя для полученной матрицы размером $(n-1) \times (n-1)$.
3. Результат предыдущего шага записывается в матрицу $n \times n$ под в строку x в столбец y.
4. Данные действия повторяются до тех пор, пока все миноры не будут найдены.

На Рис. 3 приведен пример вычисления минора для произвольной матрицы:

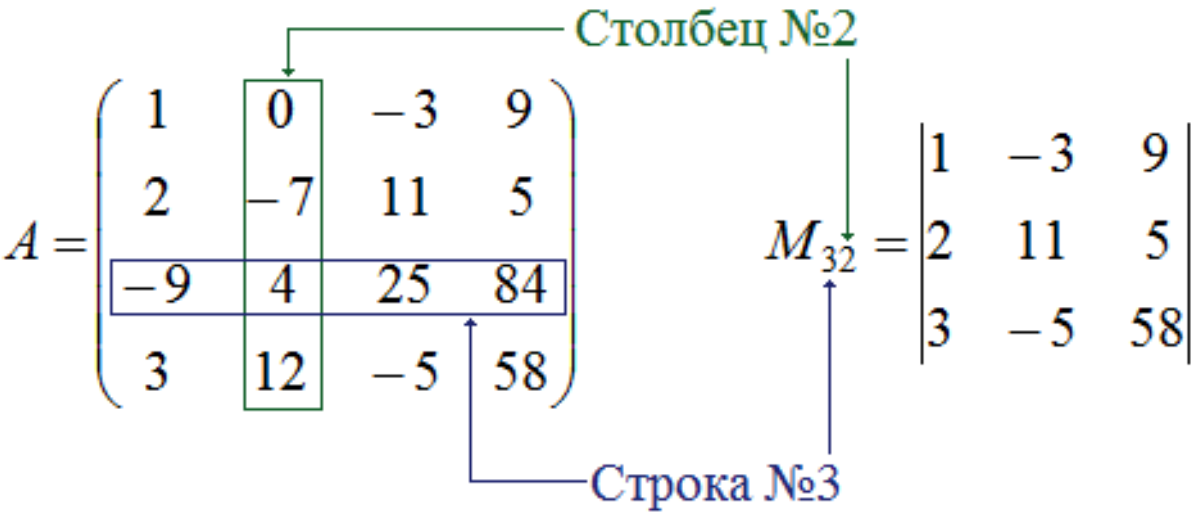

Рис. 3. Вычисления минора для произвольной матрицы

5.1 Метод распараллеливания алгоритма

При распараллеивании алгоритма каждый минор для строки x и столбца y считается на отдельном потоке. Для этого в функции запуска ядра каждому потоку выдаются свои значения x и y в соответсвии 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 допступ 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 x и y, по которым будут убраны строка и столбец.

Чтобы поток мог произвести повторные вычисления, если потоков недостаточно для полного прокрытия матрицы, то происходи 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 минора поток записывает его в матрицу $n \times n$ в строку x и в столбец y.

6 Описание эксперимента

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

- Размеры матрицы: 32×32, 71×71, 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 началы 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, 71×71, 100×100 соответственно. Знаком Н/П 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, 71×71, 100×100 соответственно.

Таблица 1. Результаты измерения времени исполнения программы для матрицы 32×32 и глобальнной 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 71×71 и глобальнной памяти

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

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	Н/П	Н/П	Н/П
100	62360	Н/П	Н/П	Н/П	Н/П
1000	Н/П	Н/П	Н/П	Н/П	Н/П

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

Числов 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 71×71 и глобальная и константная памяти

Числов блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	200490	20310	2070	990	860
10	43150	4420	Н/П	Н/П	Н/П
100	10380	Н/П	Н/П	Н/П	Н/П
1000	4560	Н/П	Н/П	Н/П	Н/П

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

Числов блоков в потоке	**Число блоков**				
	1	10	100	1000	10000
1	1101650	111320	11190	5500	4730
10	237710	24120	Н/П	Н/П	Н/П
100	56710	Н/П	Н/П	Н/П	Н/П
1000	Н/П	Н/П	Н/П	Н/П	Н/П

Как можно увидеть, что при максимальном количестве блоков вычисление миноров 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 блоко, 100 потоков;
- Для матрицы с размером 71×71 - 10000 блоко, 1 поток;
- Для матрицы с размером 100×100 - 10000 блоко, 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(71×71 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 просто не хватает памяти.

На Рис. 4 отображена зависимость времения от разного числа потоков в определенной выборке блоков.

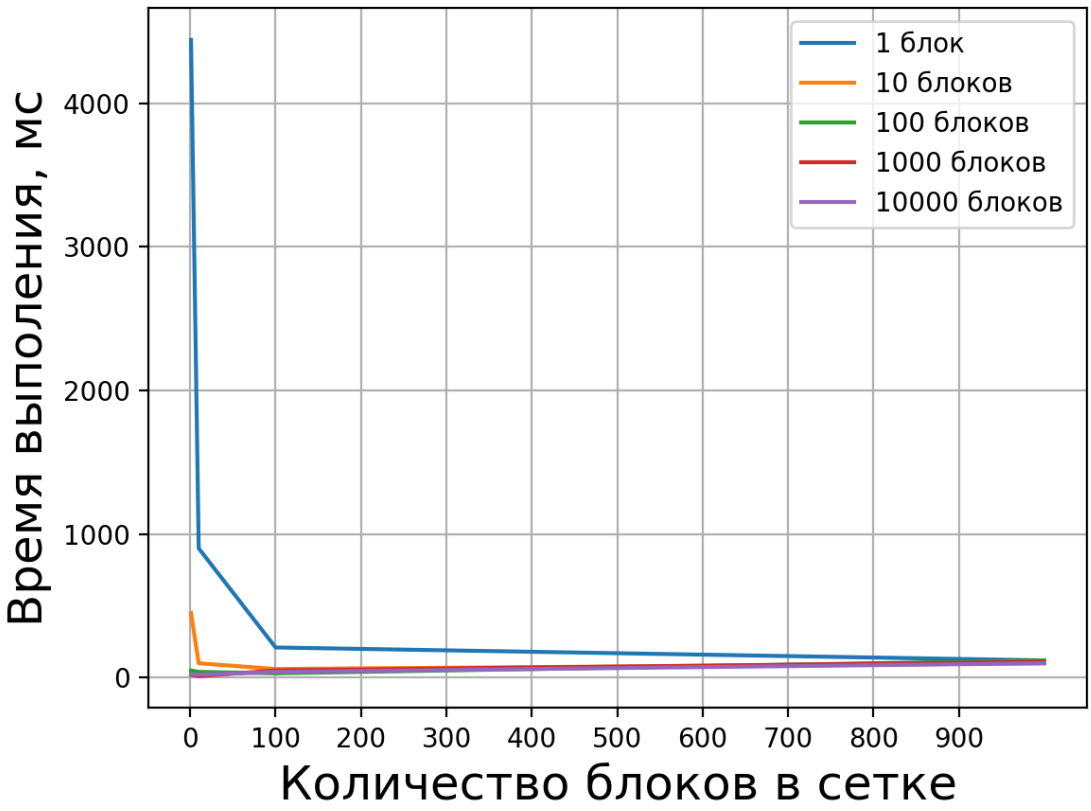

Рис. 4. Зависимость времения от разного числа потоков в определенной выборке блоков

8 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 444 раза относительно наихудшей конфигурации для матрциы 32×32, в 230 раз для 71×71 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 300 строк. Работа на СКЦ «Политехнический» шла две недели, за это время было сделано примерно 30 авторизаций и порядка 50-100 запусков задач на исполнение.

Для сборки использовался компилятор NVCC версии 11.6u2.

Список литературы

[1] Сандерс Дж., Кэндорт Э. Технология CUDA в примерах: введение в программирование графических процессоров. — М.: ДМК Пресс, 2013. — 232 с.: ил.