Министерство образования и науки 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 дисциплине «Курсовое проектирование по управлению ресурсами суперэвм»
Курсовая работа «Решение задачи транспонирования матрицы с использованием ресурсов СКЦ “Политехнический“»

Студент: _____ Богдан А. В.
группы 5130201/20102
Преподаватель: _____ Курочкин М. А.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

1 Постановка задачи **3**

2 Аппаратно-программная платформа Nvidia CUDA **4**
2.1 Архитектура Nvidia CUDA . . . 4
2.2 Вычислительные возможности Nvidia CUDA . . . 4
2.2.1 Микроархитектура Fermi . . . 4
2.2.2 Микроархитектура Kepler . . . 4
2.2.3 Микроархитектура Maxwell . . . 5
2.2.4 Микроархитектура Pascal . . . 5
2.2.5 Микроархитектура Turing . . . 5
2.3 Потоковая модель . . . 5
2.4 Устройство памяти . . . 6
2.5 Модели памяти . . . 7
2.5.1 Разделяемая память . . . 8
2.5.2 Глобальная память . . . 8
2.5.3 Регистровая память . . . 8
2.5.4 Локальная память . . . 8
2.5.5 Константная память . . . 8
2.5.6 Текстурная память . . . 9
2.6 Модель вычислений на GPU . . . 9
2.7 Компиляция программы . . . 10
2.8 Планировщик задач . . . 10

3 Суперкомпьютерный центр «Политехнический» **11**
3.1 Состав . . . 11
3.2 Характеристики . . . 11
3.3 Технология подключения . . . 11

4 Постановка решаемой практической задачи **13**

5 Алгоритм решения задачи **14**
5.1 Метод распараллеливания алгоритма . . . 14
5.1.1 Глобальная память . . . 14
5.1.2 Разделяемая память . . . 15

6 Описание эксперимента **16**

7 Анализ результатов **17**

Заключение **24**

1 Постановка задачи

Целью курсовой работы является изучение средств и технологий параллельного программирования на основе архитектуры NVIDIA CUDA для решения задачи транспонирования матрицы с использованием ресурсов СКЦ "Политехнический". Для решения задачи транспонирования матрицы необходимо выполнить следующее:

- Разработать программу для транспонирования матрицы.
- Выполнить измерение времени выполнения программы на узле «Торнадо» СКЦ "Политехнический" с различными значениями параметров количества потоков в блоке, количества блоков, размера исходного массива, а также с использованием различных моделей памяти.
- Провести анализ зависимости времени исполнения от этих параметров.

2 Аппаратно-программная платформа Nvidia CUDA

2.1 Архитектура Nvidia CUDA

GPU состоит из массива потоковых процессоров (Streaming Processor Array), которые организованы в кластеры текстурных процессоров (TCP). Каждый TCP включает несколько мультипроцессоров (SM), содержащих потоковые процессоры (SP) или ядра CUDA. Планировщик GigaThread Engine распределяет блоки потоков по SM. В целом, структура GPU остается схожей при переходе между микроархитектурами, меняются лишь размер и скорость L2-кэша. Однако внутренняя организация SM может значительно варьироваться, несмотря на схожий набор компонентов.

В архитектуре Nvidia CUDA используется модель SIMT (Single Instruction Multiple Thread), сочетающая подходы MIMD и SIMD. Задача делится на сетку блоков, каждый из которых состоит из потоков (threads). Планировщик запускает блоки на SM, причем один блок выполняется только на одном SM, но на одном SM может выполняться несколько блоков. Потоки внутри блока группируются в варпы по 32 потока, которые выполняют одну инструкцию одновременно, отсюда и происходит название модели SIMT.

2.2 Вычислительные возможности Nvidia CUDA

В данном разделе представлена небольшая выжимка по вычислительным возможностям микроархитектур Nvidia CUDA.

2.2.1 Микроархитектура Fermi

- SM: 32 CUDA Core (2 группы по 16 ядер), 16 LD/ST блоков, 4 SFU.
- Планировщики: 2 Warp Scheduler с dual-issue (параллельное выполнение независимых инструкций).
- Память: 128KB регистровый файл, 64KB L1/Shared Memory (конфигурации 16KB/48KB или 48KB/16KB).
- Особенности: Базовая структура SM, поддержка dual-issue для повышения производительности.

2.2.2 Микроархитектура Kepler

- SMX: 192 CUDA Core (12 групп по 16 ядер), 64 блока двойной точности (FP64), 32 LD/ST, 32 SFU.
- Планировщики: 4 Warp Scheduler с двумя блоками отправки команд каждый.
- Память: 256KB регистровый файл, 64KB L1/Shared Memory (добавлена конфигурация 32KB/32KB), 48KB Read-Only Cache.

- Особенности: Оптимизация энергопотребления, увеличение числа ядер и блоков FP64.

2.2.3 Микроархитектура Maxwell

- SMM: 4 блока по 32 CUDA Core, 8 LD/ST, 8 SFU. Общая производительность на SMM — 90% от SMX при меньшем размере.
- Память: 96KB Shared Memory (отдельный блок), 64KB регистровый файл на блок.
- Особенности: Модульная структура, улучшение производительности на ватт в 2 раза, на площадь кристалла — в 1.4 раза.

2.2.4 Микроархитектура Pascal

- SM: 2 блока с удвоенным регистровым файлом и увеличенным числом блоков двойной точности.
- Память: 64KB Shared Memory, Texture/L1 кэш.
- Особенности: Переход на 16нм FinFET, улучшение производительности на ватт.

2.2.5 Микроархитектура Turing

- SM: 4 блока, каждый содержит 32 FP32 ядра, 32 INT ядра, 8 FP64 ядер, 2 Tensor Core, 8 LD/ST, 1 SFU.
- Tensor Core: Обработка матриц 4x4 (FP16/FP32 вход, FP32/FP64 выход), прирост до 9.3x в задачах глубокого обучения.
- Память: 128KB L1/Shared Memory (до 96KB для Shared Memory).
- Особенности: Independent Thread Scheduling (у каждого потока свой счетчик команд), 12нм FinFET, Tensor Core для ИИ.

2.3 Потоковая модель

Архитектура Nvidia CUDA основана на понятии мультипроцессора и концепции SIMT (Single Instuction 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. Група 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.

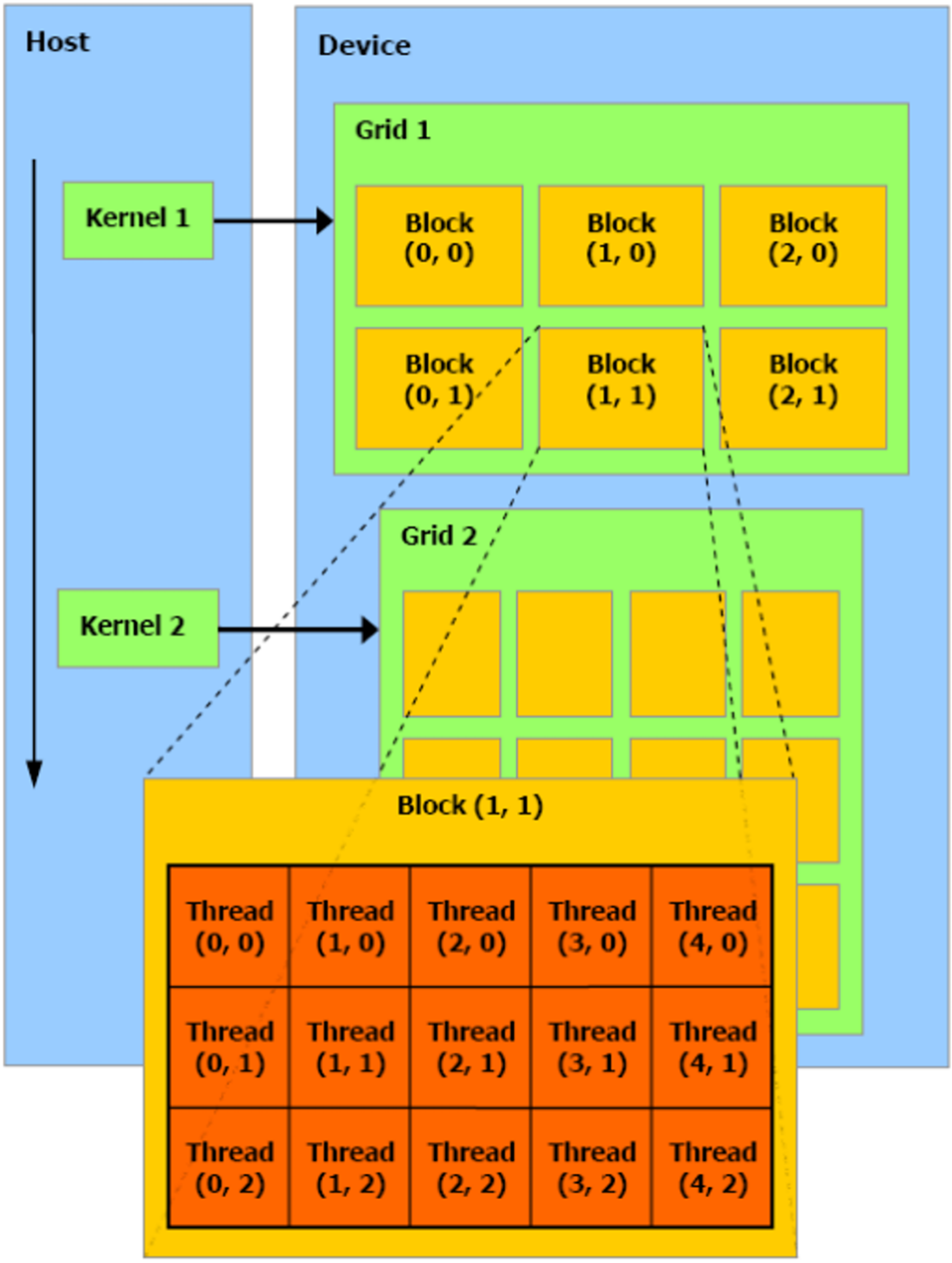

Рис. 1. Организация потоков.

2.4 Устройство памяти

Видеокарта имеет собственную глобальную память, отделённую от оперативной памяти CPU (на хосте). При выполнении кода на видеокарте (на устройстве), обращение может происходить только к собственной глобальной памяти видеокарты,

поэтому для выполнения вычислений и получения результата требуется производить перенос данных из оперативной памяти хоста в глобальную память видеокарты и обратно.

Помимо глобально оперативной памяти на каждом мультипроцессоре есть текстурная, константная, глобальная, локальная, регистровая, разделяемая память, а также кэш.

2.5 Модели памяти

На рисунке 2 изображена схема организации памяти в CUDA-программы. На ней выделено 6 видов памяти:

1. Разделяемая память.
2. Регистровая память.
3. Локальная память.
4. Глобальная память
5. Константная память.
6. Текстурная память.

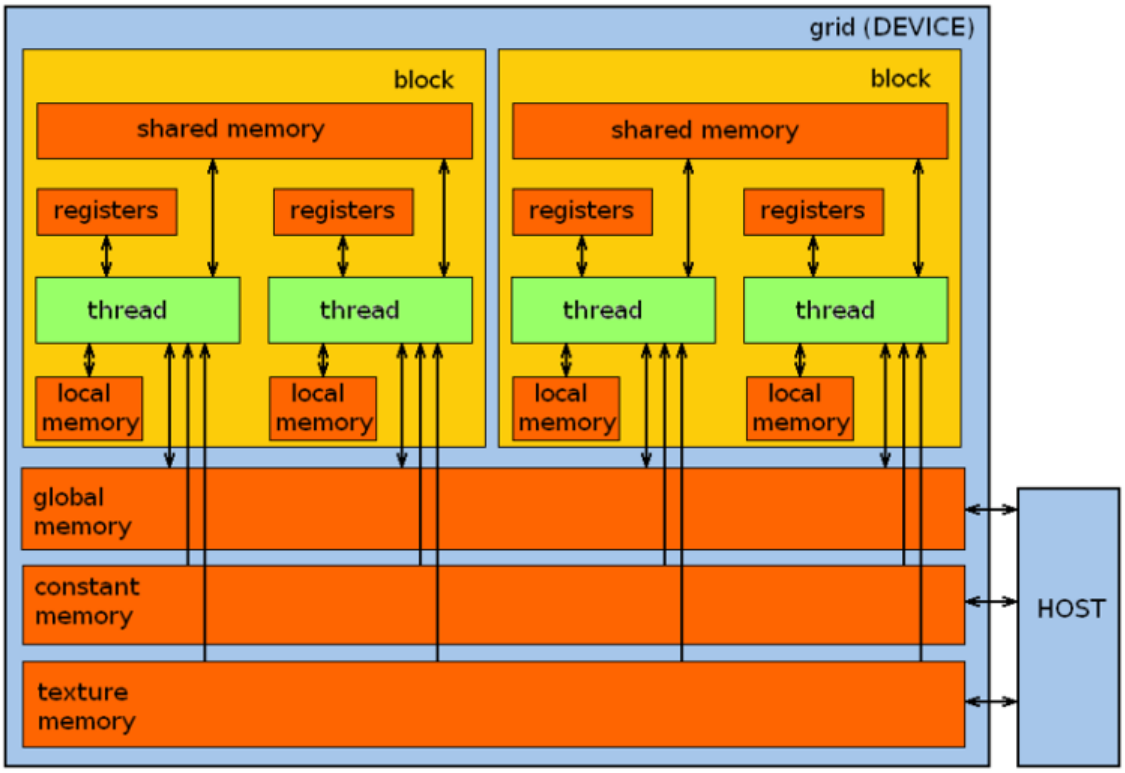

Рис. 2. Схема организации памяти CUDA-программы.

2.5.1 Разделяемая память

Язык CUDA C предоставляет в распоряжение программы так называемую разделяемую память. Если в объявление любой переменной добавить ключевое слово __shared__, то эта переменная будет размещена в разделяемой памяти. Компилятор CUDA C обрабатывает переменные в разделяемой памяти иначе, чем обычные переменные. Он создает копию такой переменной в каждом блоке, запускаемом на GPU. Все нити, работающие в одном блоке, разделяют эту переменную, но не могут ни увидеть, ни модифицировать ее копии, видимые в других блоках. Это создает прекрасный механизм взаимодействия и кооперации нитей, находящихся водном блоке. Кроме того, буферы разделяемой памяти физически находятся в самом GPU, а не в DRAM (динамическое запоминающее устройство с произвольной выборкой) вне кристалла. Это означает, что время задержки при доступе к разделяемой памяти существенно меньше, чем при доступе к обычным буферам, то есть разделяемая память играет роль внутриблочного программно управляемого кэша.

2.5.2 Глобальная память

Глобальная память обладает большим объёмом - до 4GB. 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. Также следует обратить внимание, что глобальная память расположена на хосте, когда как разделяемая память расположена на плате, в связи с чем накладные расходы на передачу данных по каналам связи значительно ниже, чем у глобальной памяти, хотя сама по себе память ничем не отличается.

2.5.3 Регистровая память

Каждому мультипроцессору доступно 8192 32-разрядных регистра. Они распределяются между нитями в этом потоке. Скорость регистровой памяти является наиболее быстрой среди всех видов памяти.

2.5.4 Локальная память

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

2.5.5 Константная память

Как следует из самого названия, константная память служит для хранения данных, которые не изменяются в процессе исполнения ядра. Оборудование NVIDIA предоставляет 64 Кб константной памяти, работа с которой организована иначе, чем со стандартной глобальной памятью. В некоторых случаях использование константной памяти вместо глобальной позволяет уменьшить трафик между памятью и процессором.

2.5.6 Текстурная память

Как и константная память, текстурная память кэшируется на кристалле, поэтому в некоторых случаях позволяет уменьшить количество образений к внешнему DRAM. Если быть точным, текстурные кэши предназначены для графических приложений, в которых доступ к памяти характеризуется высокой простанственной локальностью. В вычислительном приложении общего назначения это означает, что нить с большей вероятностью будет обращаться к адресам, расположенным «рядом» с адресами, к которым обращаются близкие нити.

С точки зрения арифметики, четыре показанных на рисунке адреса не являются соседними, поэтому не будут кэшироваться вместе при использовании стандартной схемы кэширования, применяемой в CPU. Но текстурные кэши GPU специально разработаны для ускорения доступа в таких ситуациях, поэтому применение текстурной памяти вместо глобальной может дать выигрыш.

2.6 Модель вычислений на GPU

Ядро (kernel) - эта функция, которая будет запущена на GPU и исполнена N раз с помощью N потоков. Ядро вызывается только с хоста. В CUDA C существует такая вещь как потоки (stream).

Они 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 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.

Таким образом, 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. Например, нужно выполнить в двух разных потоках аналогичные операции: копирование на девайс, запуск ядра, копирование на хост. 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 Компиляция программы

Программа для видеокарт Nvidia Cuda использует расширения языка C++, которое называется Cuda C.

Для того чтобы скомпилировать программу используется компилятор nvcc, который входит в пакет инструментов разработчика CUDA Toolkit. Это пакет можно скачать с сайта Nvidia.

2.8 Планировщик задач

В CUDA C существует такая вещь как потоки (stream). 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 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.

Планировщик задач в CUDA — это механизм, который управляет выполнением потоков (threads) на GPU. Он отвечает за распределение ресурсов, таких как вычислительные ядра и память, между потоками и блоками (blocks). На рисунке 3 представлен планировщик задач для архитектуры Turing.

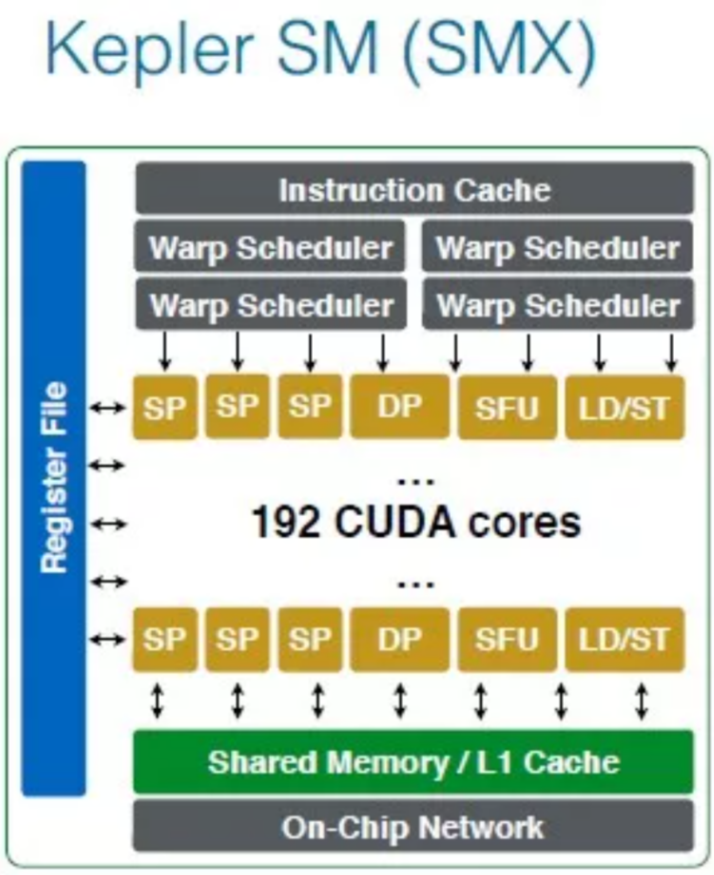

Рис. 3. Планировщик варпов в архитектуре Kepler.

3 Суперкомпьютерный центр «Политехнический»

3.1 Состав

Пользователям СКЦ «Политехнический» в настоящее время доступны следующие вычислительные ресурсы:

- 612 узлов кластера "Политехник - РСК Торнадо".
- 56 узлов кластера "Политехник - РСК Торнадо"с ускорителями NVIDIA K-40.
- 288 узлов вычислителя с ультравысокой многопоточностью "Политехник - РСК Петастрим".

3.2 Характеристики

- один узел кластера "Политехник - РСК Торнадо"содержит:
  - 2 x Intel Xeon CPU E5-2697 v3 @ 2.60GHz
  - 64G RAM
- один узел кластера "Политехник - РСК Торнадо"с ускорителями NVIDIA K-40 содержит:
  - 2 x Intel Xeon CPU E5-2697 v3 @ 2.60GHz
  - 64G RAM
  - 2 x Nvidia Tesla K40x 12G GDDR
- один узел кластера "Политехник - РСК Петастрим"содержит:
  - 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 пользователю предоставляется доступ к узлам tornado. Доступ к остальным типам узлов предоставляется по запросу.

3.3 Технология подключения

Для подключения к СКЦ "Политехнический" были выполнены следующие шаги:

1. От администрации СКЦ были получены публичный и приватный ключи в виде файлов, а также логин пользователя для доступа к системе.

2. Установка программного обеспечения:

   - Для организации подключения к терминалу СКЦ был установлен инструмент OpenSSH.
   - Для удобства переноса файлов между персональным компьютером и узлом СКЦ был установлен клиент WinSCP.

3. Подключение к узлу СКЦ через SSH:

   - В PowerShell-терминале персонального компьютера была выполнена команда:

```
ssh tm3u2@login1.hpc.spbstu.ru -i .\01
```

   - В данной команде:
     - tm3u2 - логин пользователя
     - login1.hpc.spbstu.ru - адрес узла СКЦ.
     - -i .\01 - указание на файл приватного ключа, находящегося в текущей директории.

4. Настройка клиента WinSCP для передачи файлов:

   - В клиенте WinSCP были настроены следующие параметры (рис. 4):
     - **Имя хоста:** login1.hpc.spbstu.ru
     - **Порт:** 22
     - **Имя пользователя:** tm3u2
   - Для аутентификации:
     - В разделе «Ещё... -> SSH -> Аутентификация» был выбран файл закрытого ключа, полученный от администрации СКЦ.

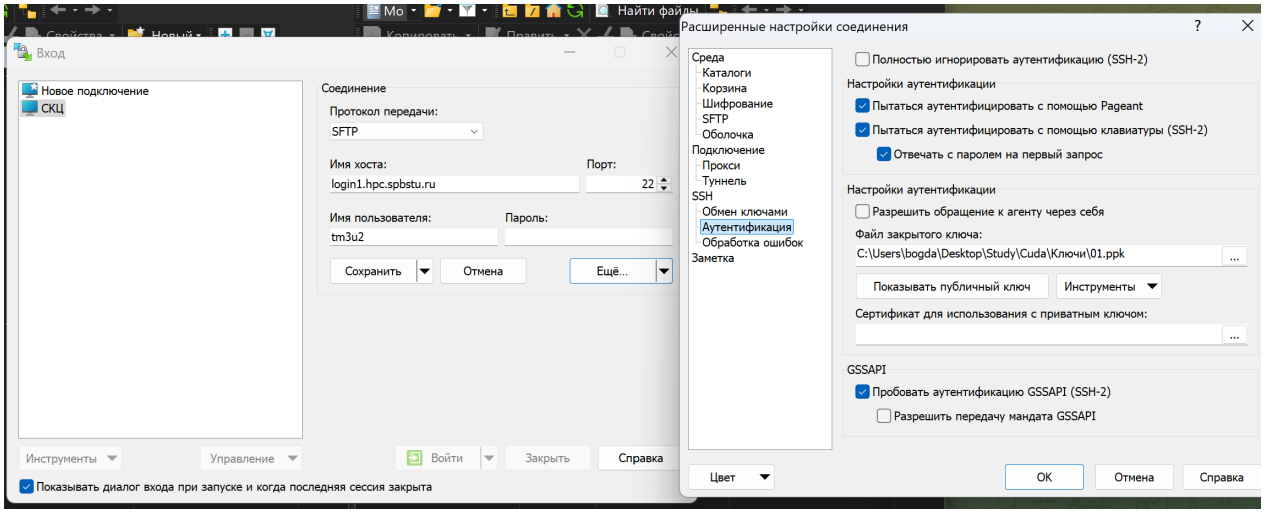

Рис. 4. Параметры WinSCP.

4 Постановка решаемой практической задачи

Дано:

- Массив M - матрица $n \times k$ вещественных чисел.

Требуется: Вычислить M^T.
Ограничения:

1. $1 \leq$ n, k ≤ 10000;
2. Вещественные числа принимают значения от 0 до 10000.

5 Алгоритм решения задачи

Пусть дана прямоугольная матрица M размерности $n \times k$:

$$M = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{pmatrix}$$

Тогда матрица M^T размерности $k \times n$:

$$M = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k1} & c_{k2} & \cdots & c_{kn} \end{pmatrix}$$

Где:

$$c_{ij} = a_{ji}, i = 1, 2, ..., k, j = 1, 2, ..., n.$$

При транспонировании матрицы M её строки становятся столбцами и наоборот. Полученная матрица называется транспонированной и обозначается M^T.

5.1 Метод распараллеливания алгоритма

Рассмотрим два подхода к распараллеливанию этой задачи на CUDA: с использованием глобальной памяти и с использованием разделяемой (shared) памяти.

5.1.1 Глобальная память

1. **Распределение потоков:**

   - Запускается 1D-сетка потоков, где каждому потоку назначается h-ое количество элементов результирующей матрицы, где

     $$h = ceil(elementsInMatrixCount/(gridDim.x * blockDim.x))$$

   - Индекс потока (threadId), а также индексы элементов(idx), обрабатываемые потоком с этим индексов, вычисляются следующим образом:

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

     $$idx = threadId + i * (gridDim.x * blockDim.x),$$

     $i = 1, 2, ..., h.\ threadId + i*(gridDim.x*blockDim.x) < elementsInMatrixCount.$

   - Каждый поток производит копирование элемента из исходной матрицы в результирующую используя только глобальную память.

2. **Копирование данных:**

   - Каждый поток копирует элементы с индексами y и x, которые вычисляются следующим образом:

$$y = idx/cols;$$

$$x = idx\%cols;$$

   в результирующую матрицу из исходной: M^T[x][y] = M[y][x]

5.1.2 Разделяемая память

1. **Разбиение матрицы на блоки данных:**

   - Запускается 1D-сетка потоков, которая делится на 1D блоки потоков.
   - Каждый блок отвечает за вычисление свей подматрицы subMatrix размером $blockDim.x * h \times blockDim.x$.
   - Каждый поток отвечает за вычисление h элементов матрицы subMatrix.

2. **Копирование данных в разделяемую память:**

   - Каждый поток производит загрузку своей партии элементов h параллельно.
   - После загрузки данных в разделяемую память производится их синхронизация, которая гарантирует, что все элементы матрицы subMatrix будут помещены в разделяемую память, до того как будут производиться дальнейшие вычисления.

3. **Копирование данных в глобальную память:**

   - Также как и в случае с копированием данных в разделяемую память, потоки будут производить параллельное копирование данных из разделяемой памяти в глобальную.

6 Описание эксперимента

Для исследования изменения времени решения задачи в зависимости от параметров были выбраны следующие значения:

- Используемая память: глобальная, разделяемая.
- Количество элементов в исходном массиве: 1000, 5000, 10000.
- Количество блоков: 1, 10, 100, 1000, 10000.
- Количество потоков в блоке: 1, 10, 100, 1000.

Для каждой комбинации параметров было проведено 100 измерений при помощи событий CUDA, и взято среднее значение.

Также было проведено исследование изменения времени решения задачи в зависимости от числовых параметров кратных степени двойки:

- Используемая память: глобальная, разделяемая.
- Количество элементов в исходном массиве: 1024, 4096, 16384.
- Количество блоков: 1, 16, 128, 1024, 16384.
- Количество потоков в блоке: 1, 16, 128, 1024.

В этом случае измерение времени также проводилось 100 раз при помощи событий CUDA, после чего было взято среднее значение.

7 Анализ результатов

В таблицах 1-3 приведены усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти.

Таблица 1. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 1000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	0.5894	0.0647	0.0117	0.0112	0.0373
10	0.0711	0.0127	**0.0071**	0.0090	0.0357
100	0.0142	0.0072	0.0075	0.0092	0.0359
1000	0.0080	0.0078	0.0087	0.0330	0.2710

Таблица 2. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 5000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	2.9204	0.2986	0.0354	0.0232	0.0438
10	0.3309	0.0389	0.0097	0.0103	0.0367
100	0.0464	0.0095	**0.0070**	0.0095	0.0359
1000	0.0136	0.0078	0.0087	0.0327	0.2715

Таблица 3. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 10000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	5.8351	0.5916	0.0656	0.0372	0.0506
10	0.6566	0.0713	0.0128	0.0115	0.0380
100	0.0857	0.0145	**0.0083**	0.0102	0.0366
1000	0.0208	0.0084	0.0100	0.0336	0.2720

В таблицах 4-6 приведены усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти.

Таблица 4. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 1000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	0.4460	0.0507	0.0116	0.0129	0.0529
10	0.0568	0.0126	0.0076	0.0109	0.0512
100	0.0128	**0.0075**	0.0077	0.0130	0.0736
1000	0.0090	0.0090	0.0120	0.0652	0.5951

Таблица 5. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 5000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	2.2061	0.2278	0.0305	0.0231	0.0609
10	0.2599	0.0329	0.0099	0.0124	0.0520
100	0.0392	0.0098	**0.0080**	0.0140	0.0739
1000	0.0151	0.0092	0.0126	0.0658	0.5954

Таблица 6. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 10000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	4.4020	0.4491	0.0525	0.0356	0.0712
10	0.5135	0.0573	0.0125	0.0137	0.0535
100	0.0708	0.0142	**0.0085**	0.0142	0.0737
1000	0.0238	0.0093	0.0129	0.0660	0.5957

В таблицах 7-9 приведены усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти и параметров, кратных степени двойки.

Таблица 7. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 1024 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	0.6026	0.0435	0.0105	0.0115	0.0792
16	0.0471	0.0080	**0.0064**	0.0114	0.0840
128	0.0124	0.0066	0.0076	0.0145	0.1074
1024	0.0080	0.0106	0.1303	0.0330	0.8735

Таблица 8. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 4096 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	2.3920	0.1560	0.0258	0.0194	0.0792
16	0.1716	0.0159	**0.0075**	0.0115	0.0841
128	0.0307	0.0078	0.0081	0.0146	0.1292
1024	0.0143	0.0108	0.0230	0.1309	1.9693

Таблица 9. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 16384 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	9.5560	0.0647	0.0831	0.0552	0.0792
16	0.6688	0.0127	0.0115	0.0114	0.0838
128	0.1055	0.0120	**0.0075**	0.0145	0.1290
1024	0.0361	0.0108	0.0233	0.1341	2.0272

В таблицах 10-12 приведены усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти и параметров, кратных степени двойки.

Таблица 10. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 1024 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	0.4566	0.0342	0.0109	0.0138	0.1123
16	0.0385	0.0081	**0.0078**	0.0135	0.1171
128	0.0117	0.0079	0.0083	0.0145	0.1599
1024	0.0091	0.0114	0.0308	0.1981	1.4166

Таблица 11. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 4096 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	1.8089	0.1194	0.0223	0.0192	0.1122
16	0.1352	0.0149	0.0084	0.0138	0.1168
128	0.0265	**0.0081**	0.0085	0.0180	0.1759
1024	0.0149	0.0115	0.0309	0.1983	3.0216

Таблица 12. Усреднённые результаты измерения времени транспонирования матрицы с использованием разделённой памяти для массива из 16384 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	16	128	1024	16384
1	7.0015	0.4586	0.0689	0.0477	0.1122
16	0.0019	0.0389	0.0105	0.0135	0.1169
128	0.0019	0.0115	**0.0085**	0.0179	0.1759
1024	0.0018	0.0112	0.0308	0.1999	3.0471

Зависимость времени выполнения от числа блоков и потоков

- **Малое число блоков и потоков**: при малом числе блоков и потоков (например, 1 блок и 1 поток) время выполнения максимально, так как нагрузка на один поток слишком велика.

- Например, для матрицы с 1000 элементов (Таблица 1):

  - 1 блок, 1 поток: 0.5894 мс.

  - 100 блоков, 10 потоков: 0.0071 мс.

- **Оптимальное число блоков и потоков**: оптимальное время выполнения достигается в ситуации, когда один поток обрабатывает один элемент матрицы. Например:

  - Для матрицы с 1000 элементов (Таблица 1): 100 блоков, 10 потоков: 0.0071 мс.

  - Для матрицы с 10000 элементов (Таблица 3): 100 блоков, 100 потоков: 0.0083 мс.

- **Большое число блоков и потоков**: при слишком большом числе блоков и потоков время выполнения увеличивается из-за накладных расходов на управление потоками и конкуренции за ресурсы.

- Например, для матрицы с 1000 элементов (Таблица 1):

  - Например, для матрицы с 1000 элементов (Таблица 1): 10000 блоков, 1000 потоков: 0.2710 мс.

Эту зависимость можно пронаблюдать на рисунке 5.

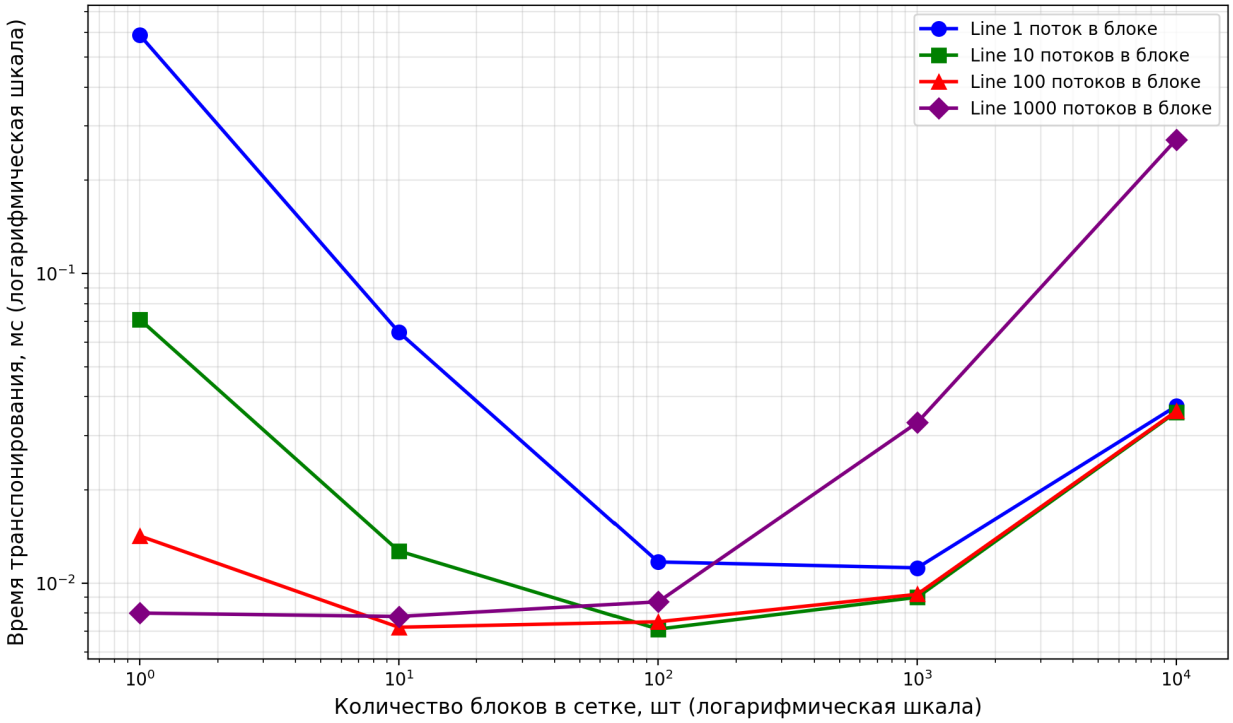

Рис. 5. Усреднённые результаты измерения времени транспонирования матрицы с использованием глобальной памяти для массива из 1000 элементов.

Сравнение глобальной и разделённой памяти

- Время выполнения транспонирования матрицы:
  - Для матрицы с 1000 элементов с использованием глобальной памяти (размеры 10x100) (Таблица 1):
    - Минимальное время выполнения: **0.0071** мс (100 блоков, 10 потоков).
    - Максимальное время выполнения: 0.5894 мс (1 блок, 1 поток).
  - Для матрицы с 1000 элементов с использованием разделённой памяти (размеры 10x100) (Таблица 4):
    - Минимальное время выполнения: 0.0075 мс (10 блоков, 100 потоков).
    - Максимальное время выполнения: 0.5951 мс (10000 блок, 1000 поток).
  - Для матрицы с 5000 элементов с использованием глобальной памяти (размеры 10x100) (Таблица 2):
    - Минимальное время выполнения: **0.0070** мс (100 блоков, 100 потоков).
    - Максимальное время выполнения: 2.9204 мс (1 блок, 1 поток).
  - Для матрицы с 5000 элементов с использованием разделённой памяти (размеры 10x500) (Таблица 5):

    - Минимальное время выполнения: 0.0080 мс (100 блоков, 100 потоков).
    - Максимальное время выполнения: 2.2061 мс (1 блок, 1 поток).
- Для матрицы с 10000 элементов с использованием глобальной памяти (размеры 100x100) (Таблица 3):
    - Минимальное время выполнения: **0.0083** мс (100 блоков, 100 потоков).
    - Максимальное время выполнения: 5.8351 мс (1 блок, 1 поток).
- Для матрицы с 10000 элементов с использованием разделённой памяти (размеры 10x1000) (Таблица 6):
    - Минимальное время выполнения: 0.0085 мс (100 блоков, 100 потоков).
    - Максимальное время выполнения: 4.4020 мс (1 блок, 1 поток).

В данном случае разделённая память не приносит никаких преимуществ в задаче по транспонированию матрицы, так как при сразу же после считывания элемента матрицы из глобальной памяти производится его обратное считывание в глобальную память по новым индексам. В таком случае не получается воспользоваться преимуществами разделяемой памяти, как, например, в случае с умножением матриц, когда между процессом считывания элемента из глобальной памяти и записью обратно происходят операции по вычислению. В таком случае разделённая память позволяет не обращаться несколько раз к глобальной памяти, за счёт этого и происходит значительное ускорение вычислений.

Сравнение параметров, не равных степени двойки, с параметрами, равными степени двойки

Параметры, равные степени двойки (например, 1024, 4096), показывают лучшее время выполнения, так как они лучше соответствуют архитектуре GPU. Например для матрицы с 1024 элементами (размеры 32x32) (Таблица 10):

- Минимальное время выполнения: 0.0064 мс (128 блоков, 16 потоков).
- Максимальное время выполнения: 0.8735 мс (16384 блоков, 1024 потоков).

Параметры, не равные степени двойки, могут показывать худшее время выполнения из-за неэффективного использования ресурсов GPU.. Например для матрицы с 1000 элементами (размеры 10x100) (Таблица 1):

- Минимальное время выполнения: 0.0071 мс (100 блоков, 10 потоков).
- Максимальное время выполнения: 0.5894 мс (1 блок, 1 поток).

Заключение

В результате выполнения курсовой работы была изучена технология параллельного программирования с использованием архитектуры Nvidia CUDA.

Для задачи по транспонированию матрицы была разработана программа на языке CUDA C. Далее была проведена отладка на персональном компьютере с видеокартой Nvidia. После чего программа была перемещена на суперкомпь ютер «Политехнический» и запущена на вычислительном узле "Политехник - РСК Торнадо" на одном графическом ускорителе Nvidia K-40.

В ходе эксперимента были получены 240 усреднённых временных измерений транспонирования матрицы при различных конфигурациях параметров распараллеливания. По полученным данным был проведён сравнительный анализ.

Использование разделяемой памяти не принесло значительного прироста производительности в задаче транспонирования матрицы. Это связано с тем, что после считывания элемента матрицы из глобальной памяти происходит его запись обратно в глобальную память по новым индексам, что не позволяет эффективно использовать преимущества разделяемой памяти. В задачах, где между считыванием и записью происходят дополнительные вычисления (например, умножение матриц), разделяемая память может значительно ускорить выполнение.

Измерение времени работы показало, что производительность зависит от количества потоков в блоке и числа блоков, а также от типа используемой памяти. Оптимальное время выполнения достигается, когда один поток обрабатывает один элемент матрицы. Например, для матрицы с 1000 элементов минимальное время выполнения составило 0.0071 мс при использовании 100 блоков по 10 потоков. При увеличении числа блоков и потоков время выполнения увеличивается из-за накладных расходов на управление потоками и конкуренции за ресурсы.

Также было установлено, что параметры, кратные степени двойки, показывают лучшее время выполнения, так как они лучше соответствуют архитектуре GPU. Например, для матрицы с 1024 элементами минимальное время выполнения составило 0.0064 мс при использовании 128 блоков и 16 потоков, когда как для матрицы с 1000 элементами минимальное время выполнения составляет 0.0071 мс при использовании 100 блоков по 10 потоков.

Список литературы

[1] Суперкомпьютерный центр - Ресурсы СКЦ. // Суперкомпьютерный цент «Политехнический». [Электронный ресурс]. – Режим доступа: https:/scc.spbstu.ru/resources (дата обращения: 23.03.2025).

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

[3] CUDA Toolkit 12.8 [Электронный ресурс]. – Режим доступа: https://developer.nvidia.com/cuda-downloads? (дата обращения: 23.03.2025).

[4] Download WinSCP. // WinSCP/ [Электронный ресурс]. – Режим доступа: https:winscp.net/eng/download.php (дата обращения: 23.03.2025).