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 построения полигона с использованием ресурсов СКЦ “Политехнический“»

Студент: _____ Яшнова Д.М.
группы 5130201/20102
Преподаватель: _____ Курочкин М. А.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

1 Постановка задачи

Целью курсовой работы является исследование технологий параллельного программирования на основе архитектуры NVIDIA CUDA для решения задачи построения полигона с использованием вычислительных ресурсов СКЦ «Политехнический».

Для достижения поставленной цели необходимо решить следующие **задачи**:

- разработать программную реализацию алгоритма построения полигона с использованием CUDA;
- провести серию вычислительных экспериментов на узле «Торнадо» СКЦ «Политехнический», варьируя параметры:
  - количество потоков в блоке,
  - количество блоков,
  - размер исходной матрицы,
  - используемые модели памяти;
- выполнить сравнительный анализ времени выполнения реализации при различных параметрах и определить оптимальные конфигурации.

2 Аппаратно-программная платформа Nvidia CUDA

2.1 Архитектура GPU

Архитектура GPU включает:

- Массив потоковых процессоров (Streaming Processor Array), организованных в кластеры (TCP)
- Каждый TCP содержит несколько мультипроцессоров (SM)
- Каждый SM включает ядра CUDA (SP)
- Планировщик GigaThread Engine распределяет блоки потоков по SM

Основные особенности:

- Общая структура сохраняется между поколениями GPU
- Основные изменения касаются:
  - Размера и скорости L2-кэша
  - Внутренней организации SM
- Набор компонентов SM остается схожим

В архитектуре Nvidia CUDA применяется модель SIMT (Single Instruction Multiple Thread), которая объединяет в себе принципы MIMD и SIMD.

Задача разбивается на множество блоков, каждый из которых состоит из потоков. Планировщик распределяет блоки по SM (вычислительным блокам), при этом каждый блок выполняется только на одном SM, но на одном SM может выполняться несколько блоков.

Потоки внутри блока объединяются 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 базируется на концепции мультипроцессора и подходе SIMT (Single Instruction Multiple Threads).

В процессе выполнения многопоточной программы на видеокарте CUDA все потоки разделяются на группы, называемые варпами. Внутри каждого варпа все потоки выполняют одну и ту же инструкцию. Группы варпов выполняются на потоковых процессорах, а распределением задач занимается планировщик.

Программа, которая выполняется одновременно на нескольких варпах, называется ядром (kernel).

Особенностью архитектуры CUDA является блочно-сеточная организация, которая отличается от организации многопоточных приложений. Драйвер CUDA самостоятельно распределяет ресурсы устройства между потоками.

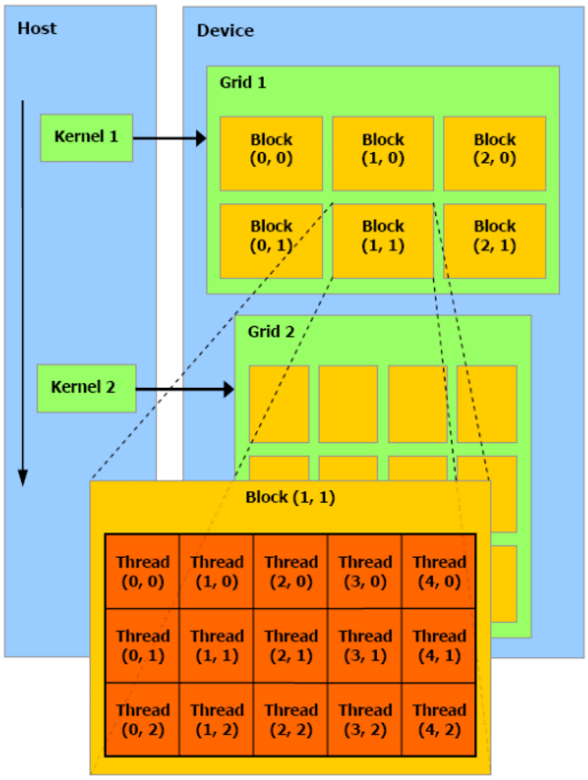


Рис. 1. Организация потоков.

2.4 Устройство памяти

В устройстве, на котором выполняется код, видеокарта имеет свою собственную память, которая не связана с оперативной памятью центрального процессора (ЦП).

Когда на видеокарте выполняется код, доступ возможен только к её собственной памяти. Поэтому для выполнения вычислений и получения результата необходимо перенести данные из оперативной памяти ЦП в память видеокарты и обратно.

Кроме глобальной памяти, на каждом мультипроцессоре есть и другие виды памяти: текстурная, константная, локальная, регистровая, разделяемая и кэш.

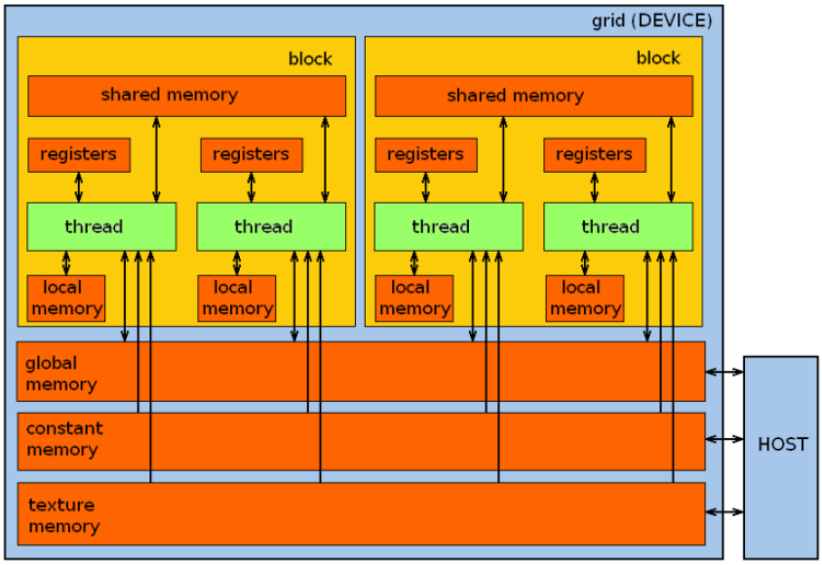


Рис. 2. Схема организации памяти CUDA-программы.

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.

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 создаёт копию такой переменной в каждом блоке, который запускается на GPU. Все нити, работающие в одном блоке, разделяют эту переменную, но не могут ни увидеть, ни изменить её копии, видимые в других блоках. Это создаёт отличный механизм взаимодействия и совместной работы нитей в одном блоке.

Кроме того, буферы разделяемой памяти физически находятся на самом GPU, а не в DRAM (динамическом запоминающем устройстве с произвольной выборкой) вне кристалла. Это означает, что время задержки при доступе к разделяемой памяти

значительно меньше, чем при доступе к обычным буферам. Разделяемая память играет роль внутриблочного программно управляемого кэша.

2.5.2 Глобальная память

Глобальная память имеет значительный объём — до 4 ГБ. Она обеспечивает произвольный доступ для всех мультипроцессоров, а также позволяет осуществлять запись и чтение с хоста.

Однако эта память работает медленно и не имеет кэша, поэтому рекомендуется минимизировать обращения к ней.

Важно отметить, что глобальная память находится на хосте, в то время как разделяемая память расположена на плате. Это приводит к тому, что накладные расходы на передачу данных по каналам связи для глобальной памяти выше, чем для разделяемой памяти, хотя сами по себе они идентичны.

2.5.3 Регистровая память

В распоряжении каждого многопроцессорного устройства имеется 8192 регистра, каждый из которых имеет разрядность 32 бита. Эти регистры распределяются между потоками. Регистровая память отличается наибольшей скоростью среди всех типов памяти.

2.5.4 Локальная память

Это ограниченная область памяти, доступная только для одного процессора. Её скорость примерно такая же, как у глобальной памяти.

2.5.5 Константная память

Из самого названия следует, что константная память предназначена для хранения данных, которые не подвергаются изменениям в процессе работы ядра.

Оборудование NVIDIA предоставляет 64 килобайта константной памяти, которая работает по-другому, чем стандартная глобальная память. В некоторых ситуациях использование константной памяти вместо глобальной позволяет сократить объём данных, передаваемых 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) — это функция, которая будет выполняться на графическом процессоре (GPU) N раз с использованием N потоков. Ядро вызывается только с хоста.

В CUDA C есть понятие потоков (stream). Они позволяют выполнять команды CUDA на GPU в порядке, определённом для каждого потока.

С точки зрения GPU потоков не существует, и все команды, поступающие на GPU, выполняются в порядке общей очереди для всех потоков. Это знание может быть полезно при оптимизации.

Планировщик может сгладить последовательность выполнения команд, запустив одновременно копирование с хоста, выполнение ядра и копирование на хост. Если ядро использует менее 50% мощности GPU, планировщик может запустить следующее ядро из другого потока, если оно готово к выполнению.

Для оптимизации времени выполнения задач в разных потоках необходимо учитывать, что на GPU команды выполняются в том порядке, в котором они были вызваны в коде хоста. Это означает, что не всегда лучше заполнять задачами один поток, затем второй и так далее. Более оптимальным подходом будет равномерный запуск задач по всем потокам.

Например, нужно выполнить в двух разных потоках аналогичные операции: копирование на устройство, запуск ядра, копирование на хост. Для этого можно сначала заполнить один поток командами, а затем другой. Тогда команды второго потока будут ожидать завершения выполнения команд первого потока или, что более вероятно, начала копирования в память

2.7 Компиляция программы

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

Для компиляции программы используется компилятор nvcc, который входит в пакет инструментов разработчика CUDA Toolkit. Этот пакет можно загрузить с сайта Nvidia.

2.8 Планировщик задач

В языке CUDA C есть такая концепция, как потоки (stream). Они позволяют запускать команды CUDA на графическом процессоре (GPU) в определённом порядке, который задаётся контекстом одного потока.

С точки зрения GPU потоки не существуют как отдельные сущности, и все команды, поступающие на GPU, обрабатываются в общей очереди. Это знание может быть полезно для оптимизации.

Порядок выполнения команд может быть оптимизирован планировщиком, который может запускать одновременно копирование данных с хоста на хост и выполне-

ние ядра. Если ядро использует менее 50% мощности GPU, то планировщик может запустить следующее ядро из другого потока, если оно готово к выполнению.

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

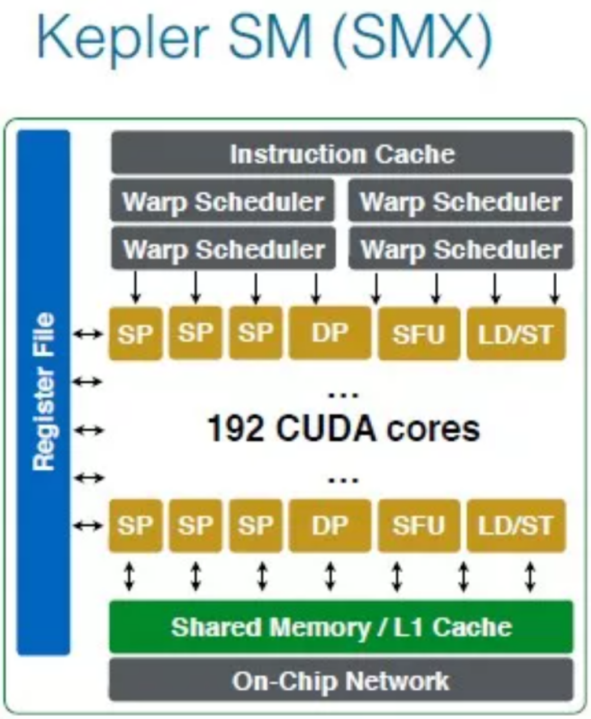


Рис. 3. Планировщик задач в архитектуре Kepler.

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 tm3u28@login1.hpc.spbstu.ru -i .\11
```

    - В данной команде:
        - tm3u28 - логин пользователя
        - login1.hpc.spbstu.ru - адрес узла СКЦ.
        - -i .\11 - 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:** tm3u28
    - Для аутентификации:
        - В разделе «Ещё... -> SSH -> Аутентификация» был выбран файл закрытого ключа, полученный от администрации СКЦ.

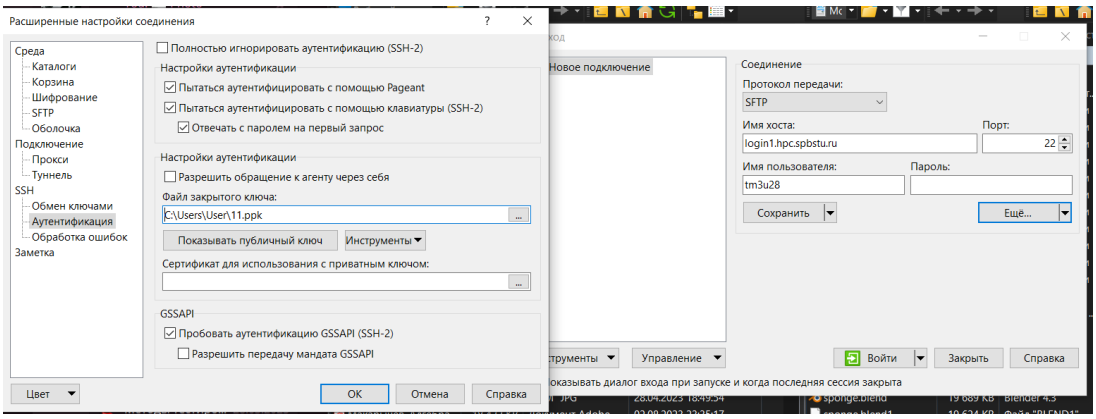


Рис. 4. Параметры WinSCP

4 Постановка решаемой практической задачи

Дано: массив P- полигон (nxn) - целых чисел, h – целое число

Надо: вычислить значение P(I,j), суммируя его в соответствии с 4-мя условиями:

$$P(i,j) = \begin{cases} -3, & \text{если } P(i,j) - P(i-1,j) < h \\ -6, & \text{если } P(i,j) - P(i,j+1) < h \\ -5, & \text{если } P(i,j) - P(i+1,j) < h \\ -7, & \text{если } P(i,j) - P(i,j-1) < h \\ P(i,j), & \text{иначе} \end{cases}$$

Ограничения: значения 1<P(I,j)<10

5 Алгоритм решения задачи

Правила преобразования:

$$P(i,j) = \begin{cases} -3, & \text{если } P(i,j) - P(i-1,j) < h \\ -6, & \text{если } P(i,j) - P(i,j+1) < h \\ -5, & \text{если } P(i,j) - P(i+1,j) < h \\ -7, & \text{если } P(i,j) - P(i,j-1) < h \\ P(i,j), & \text{иначе} \end{cases}$$

Пример расчета Исходная матрица при $h = 2$:

$$P = \begin{pmatrix} 5 & 8 & 4 \\ 3 & 6 & 2 \\ 7 & 9 & 5 \end{pmatrix}$$

Пошаговый расчет:

1. $P(1,1) = 5$:
   - $5 - 3 = 22$ (нижний сосед)
   - $5 - 8 = -3 < 2$ (правый сосед) $\Rightarrow -6$
2. $P(1,2) = 8$:
   - $8 - 5 = 32$
   - $8 - 6 = 22$
   - $8 - 4 = 42$
   - $\Rightarrow 8$ (не изменяется)
3. $P(1,3) = 4$:
   - $4 - 8 = -4 < 2$ (левый сосед) $\Rightarrow -7$
4. $P(2,1) = 3$:
   - $3 - 5 = -2 < 2$ (верхний сосед) $\Rightarrow -3$
5. $P(2,2) = 6$:
   - $6 - 8 = -2 < 2$ (верхний сосед) $\Rightarrow -3$
6. $P(2,3) = 2$:
   - $2 - 4 = -2 < 2$ (верхний сосед) $\Rightarrow -3$
7. $P(3,1) = 7$:
   - $7 - 9 = -2 < 2$ (правый сосед) $\Rightarrow -6$

8. $P(3,2) = 9$:
   - Ни одно условие не выполняется $\Rightarrow 9$
9. $P(3,3) = 5$:
   - $5 - 9 = -4 < 2$ (левый сосед) $\Rightarrow -7$

5.1 Результат

Преобразованная матрица:

$$P_{\text{new}} = \begin{pmatrix} -6 & 8 & -7 \\ -3 & -3 & -3 \\ -6 & 9 & -7 \end{pmatrix}$$

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:**
   - N - количество элементов в матрице. Запускается 1D-сетка потоков, где каждому потоку назначается s элементов результирующей матрицы:
   $$s = \lceil \texttt{N}/(\texttt{gridDim.x} \times \texttt{blockDim.x}) \rceil$$
   - Индекс потока (`threadId`) и индексы элементов (`idx`):
   $$\texttt{threadId} = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$$
   $$\texttt{idx} = \texttt{threadId} + i \times (\texttt{gridDim.x} \times \texttt{blockDim.x})$$
   где $i = 1, 2, \ldots, s$,
   $$\texttt{threadId} + i \times (\texttt{gridDim.x} \times \texttt{blockDim.x}) < \texttt{N}$$
   - Каждый поток копирует элементы между матрицами, используя только глобальную память.
2. **Копирование данных:**
   - Координаты элементов вычисляются как:
   $$y = \texttt{idx} \div \texttt{cols}$$
   $$x = \texttt{idx} \bmod \texttt{cols}$$
   - Логика обработки:
   $$\begin{aligned} P(i,j) &= -3 \quad \text{если} \quad P(i,j) - P(i-1,j) < h \\ P(i,j) &= -6 \quad \text{если} \quad P(i,j) - P(i,j+1) < h \\ P(i,j) &= -5 \quad \text{если} \quad P(i,j) - P(i+1,j) < h \\ P(i,j) &= -7 \quad \text{если} \quad P(i,j) - P(i,j-1) < h \end{aligned}$$

5.1.2 Разделяемая память

1. Разбиение матрицы на блоки данных:

    - Запускается одномерная сетка потоков, которая делится на одномерные блоки.
    - Каждый блок отвечает за вычисление своей подматрицы размером blockDim.x $\times$ s.
    - Каждый поток отвечает за вычисление s элементов матрицы.

2. Копирование данных в разделяемую память:

    - Каждый поток параллельно загружает свою порцию из s элементов.
    - После загрузки данных в разделяемую память происходит их синхронизация. Это гарантирует, что все элементы подматрицы будут размещены в разделяемой памяти до начала дальнейших вычислений.

3. Копирование данных в глобальную память:

    - Процесс копирования данных из разделяемой памяти в глобальную аналогичен предыдущему этапу. Потоки параллельно копируют данные.

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: 1024, 5041, 10000.
- Количество блоков: 1, 10, 100, 1000, 10000.
- Количество потоков в блоке: 1, 10, 100, 1000.

Так как матрицы в постановке задачи квадратные - были взяты матрицы 32*32, 71*71, 100*100.

Для каждой комбинации параметров было проведено 100 измерений и взято среднее значение.

7 Анализ результатов

В таблицах 1-3 приведены усреднённые результаты измерения времени исполнения алгоритма с использованием глобальной памяти.

Таблица 1. Усреднённые результаты измерения времени исполнения алгоритма с использованием глобальной памяти для массива из 1024 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	0.3421	0.0647	0.0238	0.0203	0.0473
10	0.0711	0.0395	**0.0180**	0.0270	0.0357
100	0.0366	0.0187	0.0414	0.0511	0.0675
1000	0.0183	0.0194	0.0198	0.0762	0.5494

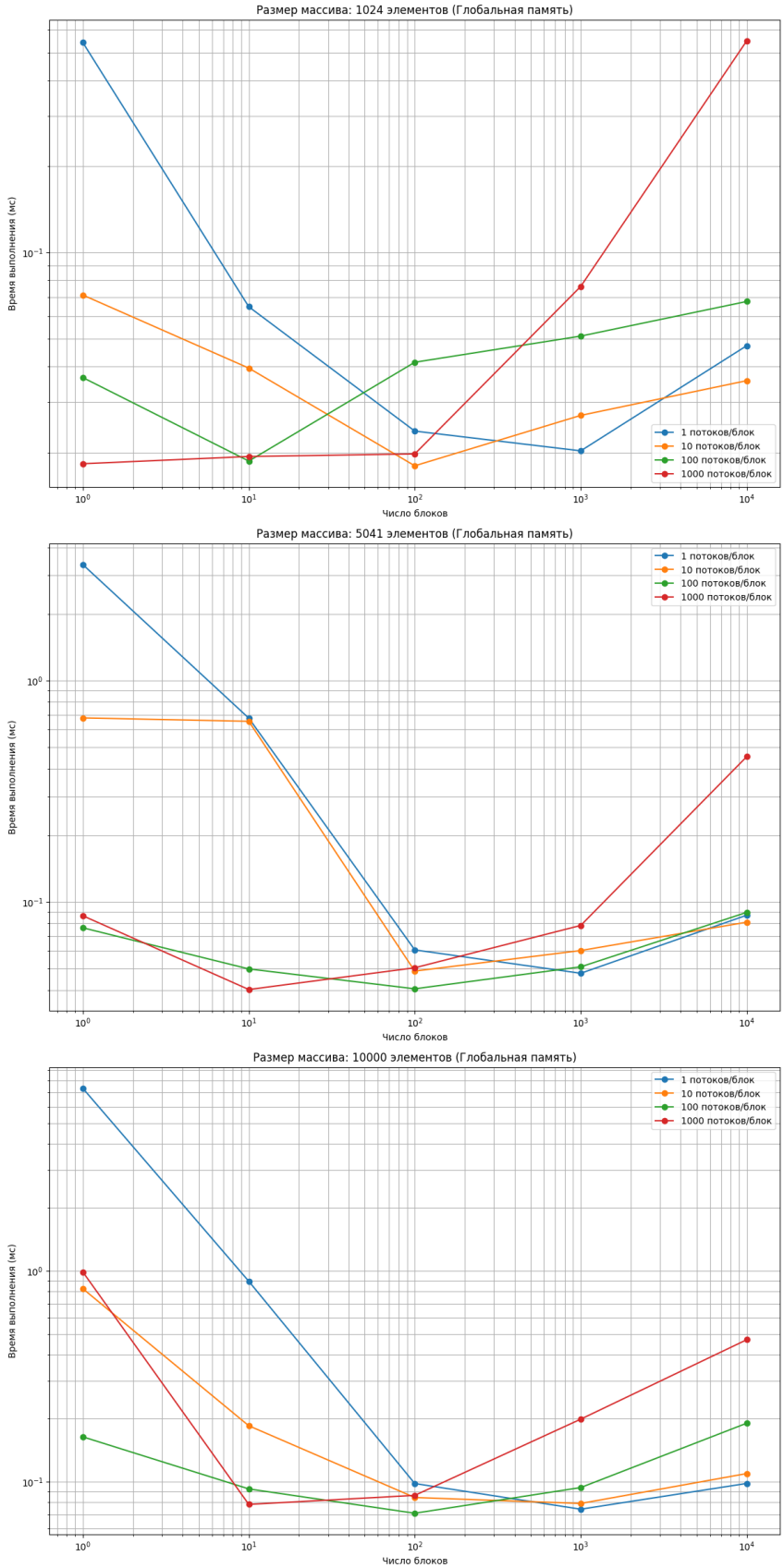
Таблица 2. Усреднённые результаты измерения времени исполнения алгоритма с использованием глобальной памяти для массива из 5041 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	3.3426	0.6765	0.0607	0.0476	0.0871
10	0.6783	0.6546	0.0487	0.0603	0.0810
100	0.0764	0.0498	0.0405	0.0509	0.0897
1000	0.0866	**0.0402**	0.0504	0.0783	0.4532

Таблица 3. Усреднённые результаты измерения времени исполнения алгоритма 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	7.3429	0.8932	0.0982	0.0742	0.0983
10	0.8239	0.1845	0.0843	0.0790	0.1095
100	0.1635	0.0926	**0.0710**	0.0940	0.1897
1000	0.9892	0.0782	0.8631	0.1983	0.4720

На рис.5 представлены зависимости времени от количества блоков для 1, 10, 100,1000 потоков в блоке при использовании глобальной памяти.



В таблицах 4-6 приведены усреднённые результаты измерения времени исполнения алгоритма с использованием разделённой памяти.

Таблица 4. Усреднённые результаты измерения времени исполнения алгоритма с использованием разделённой памяти для массива из 1024 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	0.3590	0.0783	0.0309	0.0199	0.0509
10	0.0984	0.0401	**0.0178**	0.0289	0.0589
100	0.0440	0.0187	0.0301	0.0349	0.0403
1000	0.0199	0.0181	0.0215	0.0763	0.6742

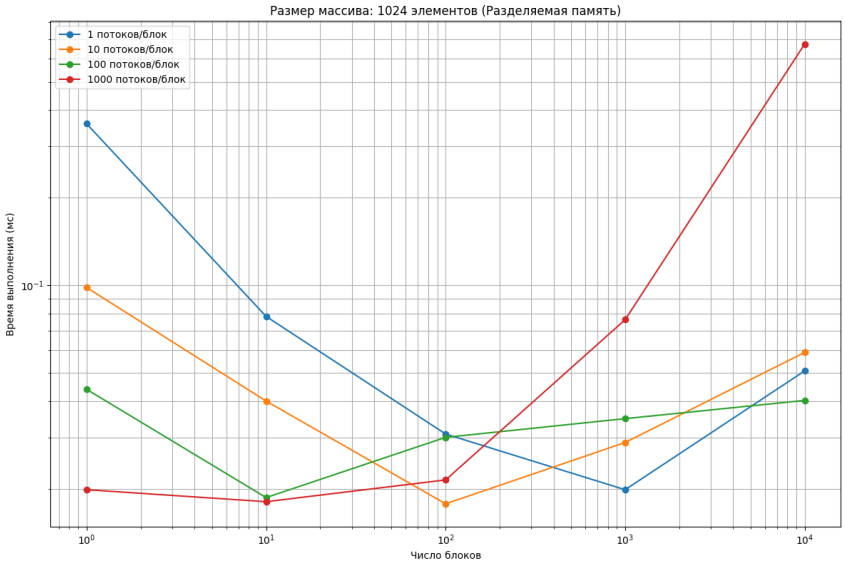
Таблица 5. Усреднённые результаты измерения времени исполнения алгоритма с использованием разделённой памяти для массива из 5041 элементов

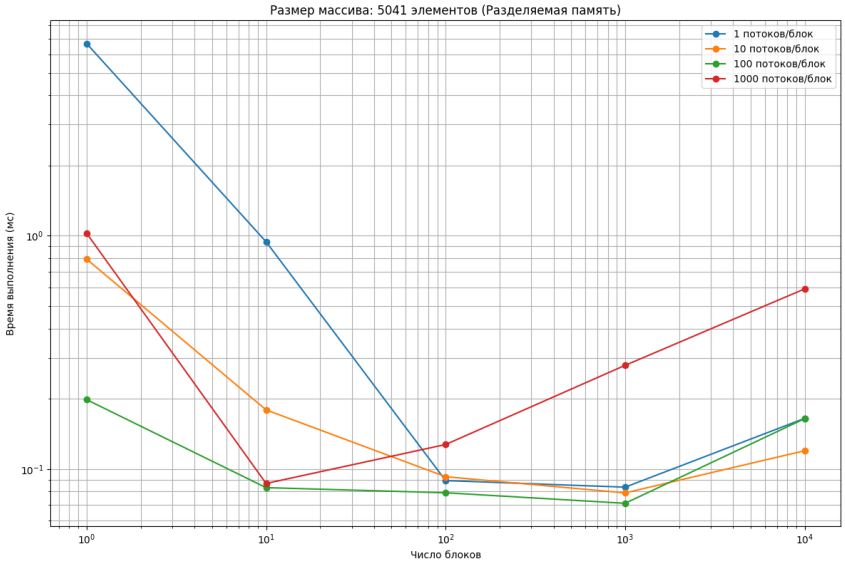
Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	3.0184	0.6027	0.0583	0.0452	0.0862
10	0.6329	0.5736	0.0418	0.0591	0.0839
100	0.0829	0.0509	0.0401	0.0678	0.0976
1000	0.0965	**0.0399**	0.0498	0.0886	0.726

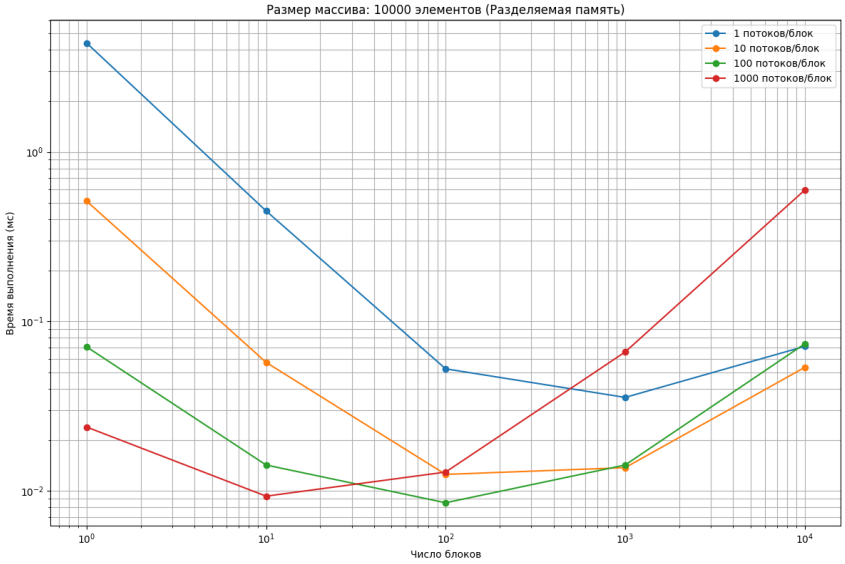
Таблица 6. Усреднённые результаты измерения времени исполнения алгоритма с использованием разделённой памяти для массива из 10000 элементов

Время выполнения, мс	Число блоков, шт				
Число потоков в блоке, шт	1	10	100	1000	10000
1	6.6784	0.9427	0.0892	0.0836	0.1649
10	0.7946	0.1793	0.0927	0.0790	0.1198
100	0.1984	0.0832	0.0791	**0.0713**	0.1645
1000	1.0263	0.0867	0.1274	0.2784	0.5922

На рис.6 представлены зависимости времени от количества блоков для 1, 10, 100,1000 потоков в блоке при использовании разделяемой памяти.


Размер массива: 1024 элементов (Разделяемая память)
1 потоков/блок
10 потоков/блок
100 потоков/блок
1000 потоков/блок
Время выполнения (мс)
Число блоков


Размер массива: 5041 элементов (Разделяемая память)
1 потоков/блок
10 потоков/блок
100 потоков/блок
1000 потоков/блок
Время выполнения (мс)
Число блоков


Размер массива: 10000 элементов (Разделяемая память)
1 потоков/блок
10 потоков/блок
100 потоков/блок
1000 потоков/блок
Время выполнения (мс)
Число блоков

8 Анализ результатов эксперимента на CUDA C

Эксперимент проводился для матриц разного размера (1024, 5041 и 10000 элементов) с использованием двух типов памяти: глобальной (Таблицы 1-3) и разделённой (Таблицы 4-6). Для каждого случая измерялось время выполнения в миллисекундах в зависимости от количества блоков и потоков в блоке.

8.1 Минимальное и максимальное время выполнения

Таблица 7. Минимальное и максимальное время выполнения (мс)

Таблица	Размер матрицы	Тип памяти	Мин.время(мс)	Макс.время(мс)
Таблица 1	1024	Глобальная	0.0180	0.5494
Таблица 2	5041	Глобальная	0.0402	3.3426
Таблица 3	10000	Глобальная	0.0710	7.3429
Таблица 4	1024	Разделённая	0.0181	0.6742
Таблица 5	5041	Разделённая	0.0399	3.0184
Таблица 6	10000	Разделённая	0.0713	6.6784

8.2 Анализ результатов

8.2.1 Зависимость от размера матрицы

Для всех таблиц наблюдается увеличение времени выполнения с ростом размера матрицы. Например:

- Минимальное время для матрицы 1024 элементов составляет 0.0183 мс (Таблица 1)
- Для матрицы 10000 элементов минимальное время составляет 0.0710 мс (Таблица 3)

Это связано с увеличением объёма данных, которые необходимо обработать.

8.2.2 Влияние количества блоков и потоков

- Наименьшее время достигается при оптимальном соотношении количества блоков и потоков. Например, для матрицы 1024 элементов (Таблица 1) минимальное время (0.0180 мс) достигается при 100 блоков и 10 потоков в блоке.
- Увеличение количества блоков или потоков сверх оптимального значения приводит к росту времени выполнения из-за накладных расходов. Например, для матрицы 10000 элементов (Таблица 6) при 10000 блоков и 1000 потоков время возрастает до 0.5922 мс.

8.2.3 Сравнение глобальной и разделённой памяти

- Разделённая память (Таблицы 4-6) в некоторых случаях показывает лучшее время выполнения. Например, для матрицы 5041 элементов минимальное время при использовании разделённой памяти (0.0399 мс, Таблица 5) меньше, чем при использовании глобальной памяти (0.0402 мс, Таблица 2).
- Однако при больших размерах матриц и неоптимальных настройках разделённая память может уступать глобальной из-за ограниченного размера и необходимости синхронизации.

8.2.4 Аномалии

- В некоторых случаях наблюдается резкое увеличение времени выполнения. Например, для матрицы 10000 элементов (Таблица 6) при 1000 потоков и 1000 блоков время составляет 0.2784 мс, а при 10000 блоков - 0.5922 мс.
- Это может быть вызвано перегрузкой системы или недостаточной оптимизацией алгоритма для больших объёмов данных.

Заключение

В ходе выполнения курсовой работы была изучена технология параллельного программирования с использованием архитектуры NVIDIA CUDA. Основные этапы работы включали:

- Разработана программа на языке CUDA C для построения полигона
- Первичная отладка выполнена на персональном компьютере с видеокартой NVIDIA
- Финальное тестирование проведено на суперкомпьютере «Политехнический»
- Использовался вычислительный узел "Политехник - РСК Торнадо" с графическим ускорителем NVIDIA K40
- Получено 120 усреднённых временных измерений
- Исследованы различные конфигурации параметров распараллеливания
- Проведён сравнительный анализ производительности

Использование разделяемой памяти не дало значительного улучшения производительности при расчете полигона. Это связано с тем, что после чтения элемента матрицы из глобальной памяти он записывается обратно по новым индексам, что не позволяет эффективно использовать преимущества разделяемой памяти. В задачах, где между чтением и записью выполняются дополнительные операции, такие как умножение матриц, разделяемая память может значительно ускорить вычисления.

Тестирование показало, что производительность зависит от количества потоков в блоке, числа блоков и типа используемой памяти. Например, для матрицы из 1024 элементов минимальное время выполнения составило 0.0180 мс при использовании 100 блоков по 10 потоков. Увеличение числа блоков и потоков приводит к росту времени выполнения из-за накладных расходов на управление потоками и конкуренции за ресурсы.

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

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