РЕФЕРАТ

Магистерская диссертация содержит: 59 страниц, 19 рисунков, 12 использованных источников.

ТЕНЗОРНЫЙ ПОЕЗД, ТЕНЗОРНЫЕ РАЗЛОЖЕНИЯ, КЕСТОВАЯ АППРОКСИМАЦИЯ, TT-CROSS, MAXVOL, МАЛОРАНГОВАЯ АППРОКСИМАЦИЯ.

В данной работе была выполнена параллельная реализация алгоритма TT-Cross с использованием графического процессора архитектуры CUDA. Параллельная реализация показала многократный прирост производительности по сравнению с аналогичной последовательной реализацией. Алгоритм тензорного разложения был применен к задаче вычисления сложной многопараметрической функции

СОДЕРЖАНИЕ

ОСНОВНАЯ ЧАСТЬ 5

ВВЕДЕНИЕ 6

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 9

1.1 Работа с данными большой размерности 9

1.1.1 Тензоры 9

1.1.2 Тензорные разложения 10

1.2 Тензорный поезд 11

1.2.1 Формат «Тензорный поезд» 11

1.2.2 Тензорные операции в TT формате 12

1.2.3 Приведение тензора в формат тензорного поезда 15

1.2.4 Алгоритм TT-SVD 15

1.2.5 Скелетное разложение 17

1.2.6 Алгоритм maxvol 19

1.2.7 Алгоритм крестовой аппроксимации 23

1.2.8 Алгоритм TT-Cross 25

1.2.9 Численные примеры 26

1.2.9.1 Развертка тензора 26

1.2.9.2 Работа алгоритма maxvol 27

1.2.9.3 Крестовая аппроксимация матрицы 28

1.2.9.4 Тензорный поезд и восстановление элемента 30

1.3 Графический процессор архитектуры CUDA 31

1.3.1 О графических процессорах 31

1.3.2 Архитектура CUDA 32

1.3.3 Разработка программ на языке CUDA C 35

2. ПРАКТИЧЕСКАЯ ЧАСТЬ 39

2.1 Используемые инструменты 39

2.2 Детали реализации 40

2.3 Анализ результатов работы 41

2.3.1 Алгоритм MaxVol 42

2.3.2 Крестовая аппроксимация 43
2.3.3 Разложение тензора Гильберта при помощи TT-Cross 47
2.3.4 Функция Розенброка 52
2.3.5 Функция Гривонка 54
ЗАКЛЮЧЕНИЕ 57
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 58

ОСНОВНАЯ ЧАСТЬ

ВВЕДЕНИЕ

В настоящее время имеется множество задач, требующих огромных вычислительных мощностей и объемов памяти. Возьмём в качестве примера модель самолёта. В неё входят сотни параметров, значения которых нельзя точно указать, но зато можно привести интервалы, в которых они находятся. Получается для моделирования, кроме пространственной сетки, в которой может быть огромное количество точек, требуется дополнительно сетка над множеством, образованным интервальными параметрами задачи, в которой каждому узлу соответствует решение при определенных значениях интервальных неопределенностей. В результате, получается экспоненциальная сложность относительно числа интервальных параметров, как по вычислительным затратам, так и по необходимому количеству памяти. Кроме того, в некоторых случаях требуется выполнять моделирование в режиме реального времени. Легко видеть, что решение «в лоб» подобных задач требует колоссальных вычислительных ресурсов. Одна из ключевых подзадач, которую можно выделить, является задача эффективного хранения и взаимодействия с многомерными массивами (далее тензорами).

Хотелось бы, имея некоторое подмножество элементов, имеющее меньшую размерность, уметь вычислять остальные значения многомерного массива. Если рассматривать тензор как функцию многих переменных, то необходимо произвести разделение переменных. С помощью произведения функций одной переменной это получится крайне неточно, но если представить её с помощью суммы таких произведений, то это уже будет давать, в некоторой мере, желаемый результат и представление будет иметь O(dnr) элементов, где d – размерность, r – ранг тензора (в данном случае многомерного массива), а n- количество значений параметров. Но проблема состоит в том, что нет методов, гарантирующих нахождение такого т.н. канонического разложения.

Рассмотрим формат, называемый тензорным поездом. При его построении тензор представляется в виде произведения объектов меньшей размерности, таким образом разделяя переменные. Привести тензор к формату тензорного поезда можно последовательно применяя сингулярное разложение. Такой алгоритм получил название TT-SVD. Если сингулярное разложение заменить на малоранговое разложение, например, крестовую аппроксимацию, то можно уменьшить временную сложность алгоритма. Алгоритм, использующий крестовую аппроксимацию, TT-Cross.

Было доказано, что сложность создания такого представления в общем случае составляет $O(m^{d+1})$ операций, где $m^d = N$ – количество элементов в тензоре, но используя некоторые приёмы можно получить временную сложность $O(Nr^2)$, что при достаточно малых рангах тензора представляет, по сути, линейную сложность относительно количества элементов тензора. При малых размерах тензора построение разложения не вызывает никаких затруднений, но при решении прикладных задач размер тензора будет огромным и потребуются большие вычислительные мощности. Есть несколько вариантов решения данной проблемы на текущем этапе. Первый – находить более оптимальные алгоритмы, что представляет большую сложность, а второй – распараллелить имеющийся алгоритм. Самая примитивная идея, которая может возникнуть – использовать потоки имеющегося процессора, но процессор нельзя нагружать большим количеством потоков, иначе получится обратный эффект - замедление. Другое решение – кластер процессоров, но такое решение достаточно дорогое, т.к. кластер требует денежных затрат, поддержание инфраструктуры и так далее, но существует средство, которое при грамотном использовании позволяет заменить большое количество классических процессоров – графический процессор.

Графический процессор, условно, представляет из себя большое количество процессоров с менее мощными ядрами по сравнению с ядрами

классического процессора, но при работе в параллели они, в общей сложности, позволяют проводить большее количество операций в единицу времени. Использование этого преимущества накладывает некоторые ограничения – алгоритм должен быть подвержен распараллеливанию, то есть необходима возможность некоторые операции проводить независимо друг от друга, иначе ядра будут ждать получения промежуточных результатов и это может не только не дать желаемого ускорения, но и значительно замедлить выполнение программы. Поэтому необходимо выполнить детальный анализ алгоритма с целью поиска мест, которые могут распараллелены.

В данной работе выполняется реализация с использованием технологий CUDA компании Nvidia. Язык CUDA идейно и синтаксически очень схож с языком C, но для него необходим специальный компилятор, поставляемый производителем. Удобство состоит в том, что при знании языка C становится гораздо проще использовать CUDA, кроме того, компилятор поддерживает C++ на центральном процессоре и позволяет использовать его стандартную библиотеку. В идеале, при работе графического процессора, центральный процессор не простаивает и готовит данные для последующей их обработки. За счёт этого, при грамотной реализации, для которой необходимо знать детали работы графического процессора, можно сократить временные затраты на несколько порядков. В настоящее время графический процессор имеется в каждом среднем компьютере, соответственно реализация алгоритма построения TT-разложения с использованием технологии CUDA может быть широко использована общественностью.

В работе исследуются различные аспекты распараллеливания алгоритма TT-Cross с использованием технологии CUDA. При проведении расчётов даже не самой современной видеокарты можно получить ускорение на несколько порядков по сравнению с центральным процессором, даже если процессор использует векторизацию, которая позволяет за один раз производить несколько итераций цикла при наличии однотипных операций.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Работа с данными большой размерности

В данном разделе рассматривается теоретический аспект работы с тензорами. Как было сказано, данные большой размерности часто возникают при моделировании, многопараметрических задачах, а также, к примеру, при решении задач компьютерного зрения или в нейронных сетях. Данный класс задач является очень востребованным в настоящее время и размерности данных постоянно растут.

1.1.1 Тензоры

Под тензорами в данной работе будем иметь в виду многомерные массивы, хранящие в себе цифровые значения.

$$A(i_1, i_2, \dots, i_d), 1 \leq i_k \leq n_k$$

Где d — размерность тензора. Без ограничений общности, в большинстве ситуаций будем считать, что $n_1 = n_2 = \cdots = n_d$. Таким образом, тензор A содержит в себе n^d значений, что означает экспоненциальную сложность по затратам памяти при росте размерности d. Из-за этого возникает сложность при хранении и обработке огромного количества данных при больших размерностях. При работе с большими матрицами за продолжительное время было выработано достаточное количество весьма эффективных методов, но данные методы не всегда возможно применить к тензорам.

Для краткости, при работе с тензорами или матрицами, будет использована нотация MatLab. Например, $A(i, j, k)$ обозначает взятие элемента тензора с индексами i, j и k соответственно. Запись $A(i, :)$, в свою очередь, означает взятие строк i, т.е. по первой размерности берётся только индекс i, а по второй все индексы. При использовании данной нотации, заглавные буквы будут означать некоторое множество индексов. Ещё одно необходимое обозначение – диапазон значений. Диапазон значений от 1 до n будет обозначаться как $1{:}\,n$.

1.1.2 Тензорные разложения

Тем не менее, существует целый класс алгоритмов, позволяющий, хоть и с некоторым набором ограничений, работать на обычном компьютере с тензорами, количество элементов которых даже превосходит объемы данных, способных поместиться в память. Такие алгоритмы называются тензорными разложениями.

Основной идей, которая используется в данном подходе является отказ от хранения избыточных данных, то есть таких элементов тензора, которые можно выразить при помощи некоторого набора других элементов. На практике весьма часто оказывается, что избыточность данных весьма высокая, поэтому такое малопараметрическое представление будет иметь довольно компактный вид, но далеко не каждый алгоритм позволит получить оптимальное разложение.

Например, одним из наиболее известных разложений является каноническое, но на практике малоприменимым из-за отсутствия устойчивых методов по его нахождению. В данной работе основной упор был сделан на практическую реализацию алгоритма TT-Cross [10] в последовательном и параллельном варианте. Данный алгоритм был выбран именно потому что показывает себя наилучшим образом при практическом применении, кроме того, как будет показано далее, данный алгоритм лишен основного недостатка, которыми страдают многие алгоритмы — экспоненциальной сложности по размерности d, как в отношении затрат по памяти. Данное качество является очень важным, так как позволяет работать с тензорами, размерность которых измеряется сотнями, в то время как алгоритмы с экспоненциальной сложностью испытывают затруднения при работе с размерностями, измеряющимися десятками.

1.2 Тензорный поезд

В данном разделе будет рассмотрен формат тензорного поезда [1], а также приведены алгоритмы приведения тензора к такому формату и возможные алгебраические операции.

1.2.1 Формат «Тензорный поезд»

Как было сказано, существует множество вариантов разложения, но одним из наиболее практичных и удобных форматов является формат Тензорного поезда (Tensor train):

$$A(i_1, i_2, \dots, i_d) = G_1^{i_1} G_2^{i2} \dots G_d^{i_d} =$$

$$= \sum_{\alpha_1, \alpha_2, \dots, \alpha_k} G_1(i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \dots G_{d-1}(\alpha_{d-2}, i_{d-1}) G_d\,(\alpha_{d-1}, i_d)\,,$$

Где $G_1^{i_1}$ – матрица-строка, $G_d^{i_d}$ – матрица-столбец, а $G_k^{i_k}, 1 < k < d$ – матрица размером $r_{k-1} \times r_k$. В данном контексте r_k называют рангами аппроксимации, а A_k ядрами разложения или вагонами. При $1 < k < d$ ядра аппроксимации являются тензорами размером $n_k \times r_{k-1} \times r_k$, то есть состоят из n_k матриц $r_{k-1} \times r_k$. Первое и последнее ядро состоят из n_1 строк и n_d столбцов размерами r_1 и r_d соответственно. Именно схожесть формата с поездом из-за «сцепки» ядер друг с другом коэффициентами α_i повлекла за собой появление названия тензорного поезда.

Говоря о таком разложении необходимо знать о том, что такое развертка тензора. Разверткой тензора A_k будем называть матрицу, полученную в результате группировки первых k индексов как строчных, а остальных $d - k$ индексов как столбцовые.

$$A_k = A([i_1, i_2, \dots, i_k], [i_k, i_{k+1}, \dots, i_d]),$$

Таким образом из тензора размером $n_1 \times n_2 \times \dots \times n_d$ получаем матрицу размером $n_1 n_2 \dots n_k \times n_{k+1} n_{k+2} \dots n_d$. Приведя тензор к такому виду, можно

удобно хранить значения в памяти компьютера и использовать все традиционные матричные методы.

Возвращаясь к формату тензорного поезда, стоит отметить, что если хранить все данные в виде линеаризованных разверток, то, по сути, весь тензор представляется в виде произведения матриц и при необходимости может быть восстановлен целиком при помощи операции перемножения. Кроме того, если известно, что ранги тензора ограничены сверху некоторым числом r, то из определения тензорного поезда легко видеть, что в данном формате необходимо хранить только $2nr + (d-2)nr^2$. Для простоты будем считать размерности будут раны $n_k = n, 1 \leq k \leq d$ и аналогично ранги будут так же равны $r_k = r, 1 \leq k \leq d$.

Данный формат является весьма удобным ещё и в виду того, что он позволяет восстановить любой элемент тензора и производить различные тензорные операции оставаясь в этом же формате.

1.2.2 Тензорные операции в TT формате

Рассмотрим операции, которые можно использовать в применении к тензорам в TT формате.

Самая простая и очевидная из операций – умножение тензора на число. Для получения результата умножения тензора A и числа α необходимо умножить любое ядро, например первое, на это число.

Если необходимо восстановить один конкретный элемент тензора $A(i_1, i_2, \dots, i_d)$, то как легко видеть из определения формата необходимо выполнить перемножение соответствующих матриц.

$$A(i_1, i_2, \dots, i_d) = G_1^{i_1} G_2^{i_2} \dots G_{d-1}^{i_{d-1}} G_d^d$$

Матрица $G_1^{i_1}$ имеет размер $1 \times r_1$, $G_k^{i_k}, 1 < k < d$ имеют размер $r_{k-1} \times r_k$, а матрица G_d^d соответственно $r_d \times 1$. Отсюда легко видеть, что результат перемножения имеет размер 1×1, что и является искомым элементом. То

есть мы имеем перемножение строки на матрицу, $d-3$ перемножений матриц и одно умножение матрицы на строку, таким образом вычислительная сложность данной процедуры составляет $O(2r^2 + (d-3)r^3)$, что является довольно дешевой процедурой при условии, что ранг r мал относительно размеров тензора.

Далее рассмотрим поэлементное сложение тензоров. Положим, что имеется два тензора в формате тензорного поезда $A = A(i_1, i_2, \dots, i_d) = A_1^{i_1} A_2^{i_2} \dots A_d^{i_d}$ и $B = B(i_1, i_2, \dots, i_d) = B_1^{i_1} B_2^{i_2} \dots B_d^{i_d}$, тогда результат сложения C можно получить при использовании ядер разложения.

$$C_1^i = [A_1^i \quad B_1^i],$$

$$C_k^i = \begin{bmatrix} A_k^i & O \\ O & B_k^i \end{bmatrix}, 1 < k < d,$$

$$C_d^i = \begin{bmatrix} A_d^i \\ B_d^i \end{bmatrix}$$

Таким образом матрицы ядер C_1 и C_d можно представить в виде блочных матриц, у которой эти самые матрицы будут записаны друг за другом и друг под другом соответственно, а матрицы ядер $C_k, 1 < k < d$ будут записаны как блочные матрицы, у которых на главной диагонали A_k и B_k, а на побочной нулевые блоки O. Операция выполняется весьма просто, но стоит принимать во внимание то, что при каждом сложении ранги ядер будут расти и после нескольких сложений тензор может значительно вырасти, что плохо скажется на производительности в последующем. Таким образом если ядра A_k и B_k имеют размеры $n_k \times r_{A_{k-1}} \times r_{A_k}$ и $n_k \times r_{B_{k-1}} \times r_{B_k}$ соответственно, то ядро C_k будет иметь размеры $n_k \times \left(r_{A_{k-1}} + r_{B_{k-1}}\right) \times \left(r_{A_k} + r_{B_k}\right)$. Из всего выше сказанного можно сделать выводы о алгоритмической сложности операции поэлементного сложения тензоров, если считать все операции копирования элементов, то она составляет

$$O\Big(2n(r_A + r_B) + n(d-2)(r_A^2 + r_B^2)\Big) \approx\ O(4nr\ +\ 2n(d-2)r^2) \approx O(ndr^2),$$

где $r = \max(r_A,\ r_B)$.

Скалярное произведение является ещё одной важной операцией и эту операцию так же можно выполнить над тензорами в TT формате. Скалярное произведение тензоров A и B определяется следующим образом

$$\langle A, B\rangle = \sum_{i_1, i_2, \dots, i_d} A(i_1, i_2, \dots, i_d) B(i_1, i_2, \dots, i_d)$$

Если рассмотреть данную операцию более подробно, то для нахождения результата необходимо сделать промежуточные вычисления.

$$\gamma_1 = \gamma_1(\alpha_1, \beta_1) = \sum_{i_1=1}^{n_1} A_1(i_1, \alpha_1) B_1(i_1, \beta_1),$$

$$\Gamma_k = \Gamma_k(\alpha_{k-1}, \beta_{k-1}, \alpha_k, \beta_k) = \sum_{i_k=1}^{n_k} A_1(\alpha_{k-1}, i_k, \alpha_k) B_1(\beta_{k-1}, i_k, \beta_k),$$

$$\gamma_d = \gamma_d(\alpha_d, \beta_d) = \sum_{i_k=1}^{n_d} A_d(\alpha_d, i_d) B_d(\beta_d, i_d)$$

Вычислив все вышеперечисленные множители, можно вычислить скалярное произведение.

$$\langle A, B\rangle = \gamma_1^T \Gamma_2 \dots \Gamma_{d-1} \gamma_d$$

Сложность предварительных вычислений вытекает из сложности перемножения матриц и составляет $O(dnr^4)$, а вычислительная сложность подсчёта итогового результата составляет $O(dr^4)$.

Из скалярного произведения вытекает ещё несколько важных операций, например Фробениусова норма, которая является обобщением Евклидовой нормы, которая определяется как

$$\|A\|_F = \sqrt{\sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \dots \sum_{i_d=1}^{n_d} A(i_1, i_2, \dots, i_d)^2},$$

Может быть записана как $\|A\|_F = \langle A, A \rangle$, аналогичным образом можно определить норму разности, т.е. расстояние между тензорами $\|A - B\|_F = \|A\|_F - 2\langle A, A \rangle + \|B\|_F$.

1.2.3 Приведение тензора в формат тензорного поезда

Увидев удобство формата тензорного поезда в плане компактности и возможности выполнять операции надо огромными тензорами за малое количество операций, возникает закономерный вопрос, существует ли способ устойчиво и эффективно привести тензор в TT формат?

Такие алгоритмы существуют и, более того, количество доказано [1], что если ранги матриц разверток $r_k = rank\, A_k$, то существует разложение тензора с рангами $r_1, r_2, \dots, r_k$, и такое разложение будет содержать не более чем $2nr + n(d-2)r^2$ элементов, причем погрешность не превышает некоторое заданное наперёд ε. Два широко распространенных алгоритма для разложения тензора в тензорный поезд называются TT-SVD [9] и TT-Cross [10]. Первый, как следует из названия, использует сингулярное разложение, а второй крестовую аппроксимацию. Рассмотрим сначала версию алгоритма, работающую на SVD разложении, а затем перейдём к TT-Cross, так как эти алгоритмы объединены одной идеей, но базируются на разных матричных разложениях.

1.2.4 Алгоритм TT-SVD

Основная идея заключается в последовательном применении матричного разложения к разверткам тензора и выборке наиболее важных элементов, по которым в дальнейшем можно будет восстановить все остальные. В данном алгоритме используется функция, изменяющая размерности матрицы – reshape. К примеру, из матрицы 6×8 можно получить матрицу 12×4, то есть суммарно количество элементов остается таким же, изменяется только форма матрицы.

Входные данные: тензор A, размером $n_1 \times n_2 \times ... \times n_d$. Результат: ядра разложения $G_1, G_2, ..., G_d$.

1. $N = n_2 n_3 ... n_d$.
2. $A_1 = reshape(A, [n_1,\ N])$ // первая развертка тензора A.
3. $U\Lambda V = SVD(A_1)$ // Выполнить сингулярное разложение матрицы-развертки A_1.
4. Отбросить младшие сингулярные числа σ_k таким образом, что ранг r должен удовлетворять условию.

$$\sqrt{\sum_{k=r+1}^{\min(n_1,N)} \sigma_k^2} \leq \frac{\varepsilon}{\sqrt{d-1}} \|A\|_F.$$

5. $G_1 = U$ // Первое ядро разложения.
6. $r_1 = r$ // Положить первый ранг равному r, который был получен в ходе отбрасывания сингулярных чисел.
7. $M = \Lambda V^T$.
8. $for\ i = 2\ to\ d-1\ do$ // цикл по всем размерностям.
9. $N = \frac{N}{n_k}$.
10. $A_i = reshape(M, [r_{i-1} n_k, N])$ // i-ая развертка оставшейся части тензора.
11. $U\Lambda V = SVD(A_i)$ // Выполнить сингулярное разложение матрицы-развертки A_i.
12. Отбросить младшие сингулярные числа σ_k таким образом, что ранг r должен удовлетворять условию.

$$\sqrt{\sum_{k=r+1}^{\min(n_1,N)} \sigma_k^2} \leq \frac{\varepsilon}{\sqrt{d-1}} \|A\|_F.$$

13. $G_i = reshape(U, [n_k, r_{k-1}, r_k])$ // преобразовать матрицу U в тензор и приравнять i-ое ядро разложения ему.
14. $M = \Lambda V$

15. $end\ for$

16. $G_d = M$

Самая большая часть вычислительных затрат заключается в вычислении сингулярного разложения на каждой итерации. На первой итерации это производится для развертки A_1 размером $n \times n^{d-1}$ и требует $O(n^{d+1})$ операций. На каждой последующей итерации, размер развертки будет составлять $nr \times n^{d-k}$, где k номер итерации. Если учесть, что сложность вычисления сингулярного разложения зависит от того первая размерность матрицы больше или вторая, и применить формулы для суммы членов геометрической прогрессии, то итоговая сложность составит $O(n^d r^2)$.

Приведенный выше алгоритм имеет ряд существенных недостатков, самым главным из них является экспоненциальная сложность по d по затратам памяти. То есть т.н. «проклятие размерности» никуда не исчезло, а это означает, что для больших значений d и n на практике данный алгоритм применять не представляется возможным. Кроме того, положение усугубляется тем, что для работы алгоритма необходимо, чтобы были известны значения всех элементов, а на практике могут возникнуть ситуации, когда значения тензора заданы в виде сложной функции, на вычисление которой нужно затратить некоторое время. Для того, чтобы сгладить данные недостатки, можно заменить сингулярное разложение на скелетное.

1.2.5 Скелетное разложение

Скелетное разложение [2][3] является основным алгоритмическим элементом алгоритма TT-Cross, который был реализован в ходе этой работы, поэтому данной аппроксимации будет уделено особое внимание.

Положим, что имеется матрица A, которая имеет размеры $m \times n$ и ранг r. В таком случае, по некоторому набору её строк и столбцов можно восстановить исходную матрицу целиком.

$$A = C\hat{A}^{-1}R = A(:, J)\ A(I, J)^{-1}\ A(J, :), \text{ где}$$

$C = A(:, J)$ и $R = A(I, :)$ являются некоторыми линейно независимыми наборами строк и столбцов в пересечении, матрица $\hat{A} = A(I, J)$ подматрица, находящаяся в их пересечении, а I и J подмножества индексов строк и столбцов размером r соответственно.

Как можно видеть, матрица $\hat{A} = A(I, J)$ должна быть невырожденной, для возможности нахождения обратной матрицы и, в случае скелетного разложения, она должна иметь максимально возможный, либо почти максимальный, определитель. В данном контексте иногда принято называть такую матрицу матрицей максимального объема. Если данное условие выполняется, то проведение матриц $C\hat{A}^{-1}$ будет содержать только не превосходящие 1 по модулю элементы. В таком случае, при построении тензорного поезда, данную матрицу можно рассматривать как альтернативу унитарной матрицу из сингулярного разложения:

$$G = C\hat{A}^{-1}$$

Вычисление такой матрицы G требует точного знания ранга исходной матрицы A. Иначе при попытке найти разложения большего, чем ранг самой матрицы, ранга матрица $\hat{A}$ всегда будет вырожденной или плохо обусловленной, а нахождение матрицы $G = C\hat{A}^{-1}$ не будет являться устойчивой операцией. На практике редко встречаются случаи, когда можно знать заранее наверняка ранг матрицы, а тем более ранги тензорного разложения. Обычно ранги или известны приблизительно, или ограничены сверху, поэтому в таком виде алгоритм является неприменимым.

Одним их возможных вариантов решения данной проблемы является применение QR разложения: $C = QT,$ где $Q -$ унитарная, а $T -$ верхнетреугольная. По определению унитарной матрицы строки и столбцы образуют базис в унитарном пространстве, а значит, если C имеет размеры $m \times r,$ и $m > r,$ то в ней можно найти r линейно независимых строк большего объема. Следовательно, существует матрица $\hat{Q}^{-1}$ и вместо

вычисления обратной матрицы $\hat{A}^{-1}$ напрямую, можно воспользоваться следующим способом:

$$G = C\hat{A}^{-1} = QTT^{-1}\hat{Q}^{-1} = Q\hat{Q}^{-1}$$

Использование данного метода значительно расширяет возможности применения алгоритма крестовой аппроксимации, т.к. если ранг точно не известен, то можно ограничить его сверху и вычислить разложение, минуя обращение вырожденной матрицы.

Как можно было заметить, алгоритм, так или иначе, требует знания подматрицы максимального объема. И так как она заранее неизвестна, то для ее нахождения было найдено несколько алгоритмов. Одним из них является алгоритм maxvol.

1.2.6 Алгоритм maxvol

Как уже было сказано, алгоритм maxvol [5] является алгоритмом для нахождения подматрицы максимального объема. Рассмотрим случай «высокой» матрицы C, размером $m \times r, m > r$, для обратного случая можно транспонировать матрицу, и задача будет сведена к данной.

Подматрицей $\hat{C}$ максимального будем называть такую матрицу, что элементы матрицы $C\hat{C}^{-1}$ по модулю не будут превосходить 1. Рассмотрим структуру результата перемножения матрицы C и $\hat{C}^{-1}$.

$$C\hat{C}^{-1} = \begin{bmatrix} I_{r\times r} \\ Z \end{bmatrix} = B, \text{где}$$

$I_{r\times r}$ является единичным блоком размером $r \times r$, а Z имеет размер $(m-r) \times r$ и его элементы, как уже было сказано, не превосходят единицы по модулю. Верхний блок будет единичным, если $\hat{C}$ составляет первые r строк матрицы C.

Соответственно, задача нахождения подматрицы максимального объема сводится к поиску строк в исходной матрице, которые в результате дадут

матрицу вида B. Так как матрица $\hat{C}$ может быть вырожденной, то обычно перед началом алгоритма случайным образом выбираются строки, которые переставляются в верхний блок. На практике, чаще всего, это позволяет достичь желаемого результата, но иногда приходится повторить эту процедуру несколько раз. Выполняя данные операции, стоит запоминать все перестановки строк, так как результатом являются номера строк, составляющие искомую матрицу. Отсюда вытекает наивный алгоритм maxvol.

Входные данные: матрица A, размером $m \times r, m > r$. Результат: массив размером r, содержащий номера строк матрицы A, составляющих подматрицу максимального объема.

1. $I = 1{:}n$ // массив, который отображает индекс строки в исходной матрице
2. $for\ i = 1\ to\ r\ do$ // выбрать случайные строки в исходной матрице и передвинуть их в верхний блок
3. $randomIndex = random(1, n)$ // случайный индекс
4. $swapRows(A, i, randomIndex)$ // поменять местами строки матрицы
5. $swap(I,\ i,\ ind)$ // поменять местами значения в массиве индексов
6. $end\ for$
7. do
8. $\hat{A} = getRows(A,\ [1{:}r])$ // взять первые r строк
9. $B = A\hat{A}^{-1}$
10. $i, j = argmax(B)$ // индексы максимального элемента B
11. $if\ B(i, j) > 1\ then$
12. $swapRows(A, i, j)$
13. $swap(I, i, j)$
14. $end\ if$

15. $while(B(i,j) > 1)$
16. $resultIndices = I(1:r)$ // строки изначальной матрицы A, которые составляют подматрицу максимального объема

Как можно было заметить, в наивной версии алгоритма присутствует многократное обращение матрицы, но так как меняются местами только две строки, то можно оптимизировать это, воспользовавшись формулой Шермана-Вудбери-Моррисона для перестановки строк матрицы местами.

$$swapRows(A, i\ j) = A + e_j(A(i,:) - A(j,:)) + e_i(A(j,:) - A(i,:)), \text{где}$$

e_i – вектор-строка, у которого на позиции i стоит единица, а на остальных нули. Воспользовавшись данной формулой, можно вывести соотношение для матрицы B, которая будет соответствовать матрице A с переставленными строками.

$$swapRows(B, i, j) = B - \frac{1}{B(i,j)}\big(B(:,j) - e_j + e_i\big)\big(B(i,:) - e_j^T\big).$$

Кроме того, стоит обратить внимание, верхний блок матрицы B является единичным $I_{r\times r}$. Это означает, что все операции над этим блоком являются излишними и их можно не производить, а работать только лишь с блоком Z.

$$swapRows(Z, i, j) = Z - \frac{1}{Z(i-r,j)}(Z(:,j) + e_i)\big(Z(i-r,:) - e_j^T\big).$$

Стоит отметить ещё то, что на практике не обязательно иметь наилучший вариант, достаточно приближенного, поэтому для сокращений количества итераций, можно заменить условие $while(B(i,j) > 1)$ на менее строгое $while(B(i,j) > 1 + \varepsilon)$, где ε является эмпирическим параметром и выбирается равным примерно 0.01. Сложив всё это, получим более оптимальный варианта алгоритма.

Входные данные: матрица A, размером $m \times r, m > r$. Результат: массив размером r, содержащий номера строк матрицы A, составляющих подматрицу максимального объема.

1. $I = 1{:}n$ // массив, который отображает индекс строки в исходной матрице
2. $for\ i = 1\ to\ r\ do$ // выбрать случайные строки в исходной матрице и передвинуть их в верхний блок
3. $ind = random(2, n)$ // случайный индекс
4. $swapRows(A, i, ind)$ // поменять местами строки матрицы
5. $swap(I,\ i,\ ind)$ // поменять местами значения в массиве индексов
6. $end\ for$
7. $\hat{A} = getRows(A,\ [1{:}r])$ // взять первые r строк
8. $B = A\hat{A}^{-1}$
9. $Z = B(r + 1{:}n, :)$
10. do
11. $i, j = argmax(Z)$ // индексы максимального элемента B
12. $if\ Z(i, j) > 1\ then$
13. $Z = Z - \frac{1}{Z(i-r,:)}(Z(:, j)\ +\ e_i)\left(Z(i - r, :)\ -\ e_j^T\right)$
14. $swap(I, i + r, j)$
15. $end\ if$
16. $while(B(i, j) > 1\ +\ \varepsilon)$
17. $resultIndices = I(1{:}r)$ // индексы строк изначальной матрицы A, которые составляют подматрицу максимального объема

Таким образом можно вычислить, что сложность алгоритма maxvol составляет $O\left(k(r^3 + nr^2\ +\ nr\ +\ 2r)\right) = O(k(r^3 + nr^2 + r(2 + n)))$, где k – количество итераций. В то время как в оптимизированном варианте обращение матрицы $\hat{A}$ происходит всего лишь один раз вне цикла и за счет применения формулы Шермана-Вудбери-Моррисона происходит не

перемножение матриц, а только строки на столбец. Поэтому алгоритм будет работать за $O\left(r^3 + k\big((n-r)r\ + 2r(n-r)\big)\right) = O(r^3 + 3rk(n-r))$.

Данный алгоритм позволяет весьма эффективно находить подматрицу максимального объема для высокой матрицы, что позволяет на основании maxvol построить алгоритм нахождения крестовой аппроксимации.

1.2.7 Алгоритм крестовой аппроксимации

Как уже было отмечено, для вычисления скелетного разложения матрицы, необходимо иметь устойчивый алгоритм вычисления подматрицы максимального объема и если сложить то, что было сказано о скелетном разложении и maxvol, то можно получить алгоритм крестовой аппроксимации.

Суть данного алгоритма заключается в последовательном применении алгоритма maxvol для строк и столбцов матрицы. Строки, очевидно, перед поиском подматрицы максимального объема необходимо транспонировать. Естественно, нужно не забывать о том, что следует использовать QR разложение, перед применением maxvol для того, чтобы обеспечить наличие матрицы с ненулевым определителем.

Входные данные: матрица A, размером $m \times n$, ранг аппроксимации r, параметр останова δ. Результат: факторы скелетного разложения $C\hat{A}^{-1}R = A$.

1. $I = random(1, m, r)$ // r случайных строчных индексов
2. $J = random(1, n, r)$ // r случайных столбцовых индексов
3. $A_0 = O_{m \times n}$ // $O_{m \times n}$ – матрица с нулевыми элементами размером $m \times n$
4. $k = 1$
5. do
6. $R = A(I, :)$ // строки с индексами I

7. $R = R^T$ // выполнить транспонирование, чтобы получить «высокую» матрицу
8. $QT = QR(R)$ // QR разложение
9. $J = maxvol(Q)$
10. $C = A(:, J)$ // строки с индексами I
11. $QT = QR(C)$ // QR разложение
12. $I = maxvol(Q)$
13. $\hat{Q} = Q(I, :)$
14. $A_k = Q\ \hat{Q}^{-1} A(I, :)$ // k-ое приближение
15. $k = k + 1$
16. $while(\|A_k - A_{k-1}\|_F > \delta \|A_k\|_F)$

В качестве алгоритма QR разложение рекомендуется использовать алгоритм, основанный на отражениях Хаусхолдера [6]. Его сложность составляет $O(mr^2)$ при размерах матрицы $m \times r$, что хорошо вписывается в допущение, что ранг матрицы мал относительно её размеров. Итоговая сложность данного алгоритма составляет примерно $O(mr^2 + r^3 + rm + nr^2 + r^3 + rn) \approx O(nr^2 + r^3)$, если $m \sim n$.

Явным преимуществом крестовой аппроксимации над SVD разложением является то, что для вычисления разложения не нужно заранее знать все элементы матрицы, а необходимые значения можно вычислять на лету. Таким образом, если значения матрицы являются сложно вычислимой функцией, но известно, что ранг достаточно мал, можно весьма быстро построить разложение и восстановить оставшиеся элементы. Кроме того, данный алгоритм позволяет иметь фиксированные ранги, при построении тензорного поезда.

1.2.8 Алгоритм TT-Cross

Алгоритм TT-Cross имеет в основании ту же самую идею, что и TT-SVD. Необходимо последовательно применять матричное разложение к разверткам тензора.

Входные данные: тензор A, размером $n_1 \times n_2 \times \ldots \times n_d$, ранги аппроксимации $r_1, r_2 \ldots, r_{d-1}$. Результат: ядра разложения $G_1, G_2, \ldots, G_d$.

1. $N = n_2 n_3 \ldots n_d$.
2. $A_1 = reshape(A, [n_1,\ N])$ // первая развертка тензора A.
3. $Q\ \hat{Q}^{-1} R = crossApproximation(A_1, r_1)$ // Выполнить крестовую аппроксимацию матрицы-развертки A_1.
4. $G_1 = Q\ \hat{Q}^{-1}$ // Первое ядро разложения.
5. $for\ i = 2\ to\ d-1\ do$ // цикл по всем размерностям.
6. $N = \frac{N}{n_k}$.
7. $A_i = reshape(R, [r_{i-1} n_k, N])$ // i-ая развертка оставшейся части тензора.
8. $Q\ \hat{Q}^{-1} R = crossApproximation(A_i, r_i)$ // Выполнить крестовую аппроксимацию матрицы-развертки A_1.
9. $G_i = reshape\left(Q\ \hat{Q}^{-1}, [n_k, r_{k-1}, r_k]\right)$ // преобразовать матрицу $Q\ \hat{Q}^{-1}$ в тензор и приравнять i-ое ядро разложения ему.
10. $end\ for$
11. $G_d\ =\ R$

Важным свойством данного алгоритма является то, что он имеет фиксированные ранги аппроксимации, которые, в зависимости от ситуации, можно увеличивать или уменьшать. Таким образом для хранения тензора в таком формате необходимо $O(dn + (d-2)nr^2 + dn) = O(2dn + (d-2)nr^2) \approx O(dnr^2)$ элементов, а чтобы привести тензор в формат тензорного поезда необходимо выполнить примерно $\sum_{i=1}^{d} \left((rn)r^2 + r^3 + \frac{N}{n^i} r^2 + r^3\right) =$

$dnr^3 + 2dr^3 + \sum_{i=1}^{d} \frac{n^d}{n^i} r^2 = dnr^3 + 2dr^3 + \frac{n^d\left(1-\frac{1}{n^d}\right)}{\left(1-\frac{1}{n}\right)} \approx O(d(n+1)r^3 + n^d) \approx O(dnr^3 + n^d)$, что составляет линейную сложность относительно количества элементов тензора. Хоть временная сложность и составляет $O(n^d)$, так как в худшем случае потребуются все элементы тензора, но при этом нет необходимости всегда иметь все элементы в памяти.

1.2.9 Численные примеры

Для большей наглядности будут приведены некоторые численные примеры. Для некоторых примеров будет использован тензор Гильберта. Каждый его элемент $a_{i_0,i_1,\ldots,i_d}$ равен $\max(1, \sum_{k=0}^{d} i_k)$. Данный тензор является очень удобным для отладки программ, так как при больших размерностях он является очень разреженным и при этом имеет весьма простую структуру. Примеры будут больше нацелены на реализацию, поэтому отсчёт, как принято в программировании, будет начинаться с нуля. Вещественные числа будут приведены с точностью до трёх знаков после запятой.

1.2.9.1 Развертка тензора

Рассмотрим тензор Гильберта с размерностями $2 \times 3 \times 2$.

$$A(:,:,0) = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}, A(:,:,1) = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix},$$

Тогда развертки A_1 и A_2 для данного тензора будут выглядеть следующим образом:

$$A_1 = \begin{bmatrix} 1 & 1 & 2 & 1 & 2 & 3 \\ 1 & 2 & 3 & 2 & 3 & 4 \end{bmatrix},$$

$$A_2 = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \end{bmatrix}.$$

1.2.9.2 Работа алгоритма maxvol

Далее рассмотрим пример работы алгоритма maxvol для матрицы C. $n = 4, r = 2$. Массив индексов содержит изначальные номера строк, $I = [0, 1, 2, 3]$. Для простоты ε примем за 0.

$$C = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 4 & 1 \\ 2 & 5 \end{bmatrix}, \hat{C} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}, \hat{C}^{-1} = \begin{bmatrix} 3 & -2 \\ -1 & 1 \end{bmatrix}.$$

Соответственно, матрица $B = C\hat{C}^{-1}$ будет равна:

$$B = \begin{bmatrix} I_{2\times 2} \\ Z \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 8 & -5 \\ 1 & 1 \end{bmatrix}.$$

Как можно помнить, в оптимизированном варианте алгоритма, верхнюю часть матрицы B можно отбросить и работать только с матрицей $Z = \begin{bmatrix} 8 & -5 \\ 1 & 1 \end{bmatrix}$. Максимальный элемент данной матрицы $Z(0,0) = B(0+r, 0) = B(2,0) = 8 > 1$. Соответственно, необходимо поменять местами строки под номерами 0 и 2.

$$swapRows(Z, 2, 0) = Z - \frac{1}{Z(i-r, j)}(Z(:, j) + e_i)\left(Z(i-r, :) - e_j^T\right) =$$

$$= \begin{bmatrix} 8 & -5 \\ 1 & 1 \end{bmatrix} - \frac{1}{8}\left(\begin{bmatrix} 8 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)([8 \quad -5] - [1 \quad 0]) = \begin{bmatrix} 8 & -5 \\ 1 & 1 \end{bmatrix} - \frac{1}{8}\begin{bmatrix} 63 & -45 \\ 7 & -5 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & 5 \\ \frac{8}{1} & \frac{8}{13} \\ \frac{8}{8} & \frac{8}{8} \end{bmatrix}.$$

После этого необходимо произвести перестановку в массиве индексов. $I = swap(I, 2, 0) = [2, 1, 0, 3]$. Очевидно, что значение максимального по модулю элемента в обновленной матрице Z превышает 1. $Z(1,1) = B(3,1) = \frac{13}{8}$, значит необходимо поменять местами строки 1 и 3.

$$swapRows(Z,3,1) = \begin{bmatrix} \frac{1}{8} & \frac{5}{8} \\ \frac{1}{8} & \frac{13}{8} \end{bmatrix} - \frac{8}{13}\begin{bmatrix} \frac{5}{8} \\ \frac{13}{8}+1 \end{bmatrix}\begin{bmatrix} \frac{1}{8} & \frac{13}{8}-1 \end{bmatrix} =$$

$$= \begin{bmatrix} \frac{1}{8} & \frac{5}{8} \\ \frac{1}{8} & \frac{13}{8} \end{bmatrix} - \frac{8}{13}\begin{bmatrix} \frac{5}{8} \\ \frac{21}{8} \end{bmatrix}\begin{bmatrix} \frac{1}{8} & \frac{5}{8} \end{bmatrix} = \begin{bmatrix} \frac{1}{8} & \frac{5}{8} \\ \frac{1}{8} & \frac{13}{8} \end{bmatrix} - \frac{8}{13}\begin{bmatrix} \frac{5}{64} & \frac{25}{64} \\ \frac{21}{64} & \frac{105}{64} \end{bmatrix} \approx$$

$$\approx \begin{bmatrix} 0.07 & 0.38 \\ -0.07 & 0.61 \end{bmatrix}.$$

И аналогично следует поменять элементы в массиве с индексами. $I = swap(I,3,1) = [2,3,0,1]$.

Как можно видеть, матрица Z больше не содержит элементов, по модулю превосходящих 1, а значит алгоритм завершает свою работу и первые $r = 2$ строки содержат подматрицу максимального объема $I(0{:}\,2) = [2,3]$. А сама подматрица максимального объема равна

$$\hat{C} = \begin{bmatrix} 4 & 1 \\ 2 & 5 \end{bmatrix}.$$

Если перебрать все возможные варианты, коих не очень много, то можно будет увидеть, что данная подматрица имеет максимальный определитель.

1.2.9.3 Крестовая аппроксимация матрицы

Рассмотрим пример работы алгоритма крестовой аппроксимации для следующей матрицы A.

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Легко видеть, что ранг этой матрицы составляет $r = 1$. Рассмотрим работу алгоритма крестовой аппроксимации для данной матрицы. Положим, что $I = [0]$ и $J = [0]$.

$$R = A(I,:)^T = \begin{bmatrix}1\\0\\2\\0\end{bmatrix}, QT = QR(R), Q = \begin{bmatrix}-0.45\\0\\-0.89\\0\end{bmatrix}, T = [-2.24].$$

Получив матрицу Q, нужно найти в ней подматрицу максимального объема. Очевидно, что $J = \mathrm{m}axvol(Q,r) = [2]$. Далее необходимо выполнить аналогичную процедуру для столбцов.

$$C = A(:,J) = \begin{bmatrix}2\\0\\0\\4\\0\end{bmatrix}, QT = QR(C), Q = \begin{bmatrix}-0.45\\0\\0\\-0.89\\0\end{bmatrix}, T = [-4.47]$$

Аналогично предыдущему шагу достаточно просто заметить, что $I = maxvol(Q,r) = [3]$.

$$\hat{Q} = Q(I,:) = [-0.89], \hat{Q}^{-1} = [-1.12]$$

Выполнив эти операции, можно построить приближение A_1.

$$A_1 = Q\,\hat{Q}^{-1}A(I,:) = \begin{bmatrix}-0.45\\0\\0\\-0.89\\0\end{bmatrix}[-1.12][2 \quad 0 \quad 4 \quad 0] = \begin{bmatrix}1&0&2&0\\0&0&0&0\\0&0&0&0\\2&0&4&0\\0&0&0&0\end{bmatrix},$$

Что в точности является нашей исходной матрицей. Так как в данном примере значения берутся с округлением, то перемножив данные матрицы точного результата не получится, но, если повторить те же самые операции при помощи компьютера, можно получить желаемы результат.

1.2.9.4 Тензорный поезд и восстановление элемента.

Рассмотрим тензорный поезд для тензора Гильберта A, размером $4 \times 4 \times 4$, который выглядит следующим образом:

$$A(:,:,0) = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix}, A(:,:,1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix},$$

$$A(:,:,2) = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \end{bmatrix}, A(:,:,3) = \begin{bmatrix} 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \\ 6 & 7 & 8 & 9 \end{bmatrix}.$$

Данный тензор разложить в тензорный поезд при помощи Алгоритма TT-Cross с рангами $[3,\ 3]$, при этом имея возможность точно восстановить любой элемент. В таком случае можно получить следующие ядра:

$$G_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix},$$

$$G_1^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.6 & 0.4 \end{bmatrix}, G_1^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.8 & 0.2 \\ 0 & 0.4 & 0.6 \end{bmatrix},$$

$$G_1^2 = \begin{bmatrix} 0 & 0.8 & 0.2 \\ 0 & 0.6 & 0.4 \\ 0 & 0.2 & 0.8 \end{bmatrix}, G_1^3 = \begin{bmatrix} 0 & 0.6 & 0.4 \\ 0 & 0.4 & 0.6 \\ 0 & 0 & 1 \end{bmatrix},$$

$$G_2 = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 6 & 7 & 8 & 9 \end{bmatrix}.$$

По определению тензорного поезда, G_0 состоит из 4 строк 1×3, G_1 из 4 матриц размером 3×3, а G_2 из 4 столбцов 3×1. Чтобы восстановить элемент $A(3,1,2)$ необходимо выполнить следующие действия:

$$A(3,1,2) = G_0(3,:)G_1^1 G(:,2) =$$

$$= [0 \quad 0 \quad 1] \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.8 & 0.2 \\ 0 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 8 \end{bmatrix} = 6.$$

Как можно видеть, элемент было точно восстановлен. Стоит помнить, что если тензор был разложен с погрешностью, то и восстановление элементов будет происходить с погрешностью.

1.3 Графический процессор архитектуры CUDA

1.3.1 О графических процессорах

Ещё в относительно недавнее время в 1980-ые года вычислительная мощность отдельно взятого процессора была весьма малой. Время шло и закон Мура, говорящий ой том, что количество транзисторов на интегральной схеме удваивается каждые два года, а повышение тактовой частоты процессора пока что позволяло добавиться большей производительности.

В то же время существовало отдельное направление по увеличению мощности компьютера в целом. Сотни и тысячи процессоров объединялись в суперкомпьютеры, но это было не доступно обычным пользователям по причине высокой цены. Поэтому эту идею попытались перенести на процессоры и так в 2000-ных годах появились многоядерные процессора, начиналось всё с двухъядерных и трехъядерных процессоров. В настоящее время пользовательские процессоры могут иметь 16 ядер и даже больше.

С 1990-ых годов можно было наблюдать взрывной рост графических пользовательских интерфейсов для операционных систем, который начался с появлением Windows. Вместе с этим появилась потребность в графических ускорителях, которые умели работать с 2D и 3D графикой. Особенность графических ускорителей заключалась в том, что они были спроектированы таким образом, чтобы выполнять несколько арифметических операций за раз, а вычисление значения нескольких пикселей за цикл. Первые графические процессоры имели весьма жесткий API, который был завязан на библиотеки

компьютерной графики OpenGL или DirectX, соответственно можно было знать механизмы работы операций, связанных с графикой, например высчитывание цвета пикселя.

Не смотря на такие жесткие ограничения, нашлись исследователи, которые попытались переложить алгоритмы линейной алгебры на операции ускорителей, так как скорость выполнения арифметических операций была очень высока. Такое движение заставило производителей обратить внимание и в 2006 году компания NVIDIA представила графический процессора на собственной архитектуре CUDA [11], которая позволяла программе использовать любое для выполнения вычислений общего назначения арифметически-логическое устройство микросхемы. Для этой же цели операции для вещественных чисел были спроектированы таким образом, чтобы реализовывать стандарт IEEE для чисел с плавающей точкой, а так же предоставлена возможность произвольного доступа к памяти, как для чтения, так и для записи. Но и этого было недостаточно, все вычисления все ещё были привязаны в OpenGL или DirectX, поэтому чтобы избавить разработчиков от изучения данных библиотек, был разработан специальный язык программирования, который получил название CUDA C, и предоставлен компилятор. В таком виде для графического процессора было найдено множество применений, например для обработки изображений медицинских анализов, обсчета численных моделей в области гидродинамики или моделирования взаимодействия молекул.

1.3.2 Архитектура CUDA

Архитектура графического процессора существенно отличается от архитектуры центрального процессора. На рис. 1.1 схематически представлена архитектура центрального процессора. Каждое ядро центрального процессора является полностью самостоятельным процессором и имеет собственный кэш, помимо общего, но при этом сам вычислительный модуль (АЛУ) занимает небольшую часть ядра. При такой архитектуре

каждое ядро имеет архитектуру SISD, то есть одна инструкция и одни входные данные. Но стоит помнить, что современные процессоры имеют механизм векторизации – если выполняется несколько одинаковых операций подряд, то они могут быть выполнены на регистре большего размера за одну инструкцию, в таком случае, даже ядро центрального процессора можно считать SIMD устройством (одна инструкция, множественные входные данные), а процессор в целом в таком случае является MIMD устройством. Но даже такая оптимизация не позволяет центральному процессору иметь такой успех в параллельных вычислениях, так как за раз производится до 16 операций с плавающей точкой для 64-битных чисел [7].

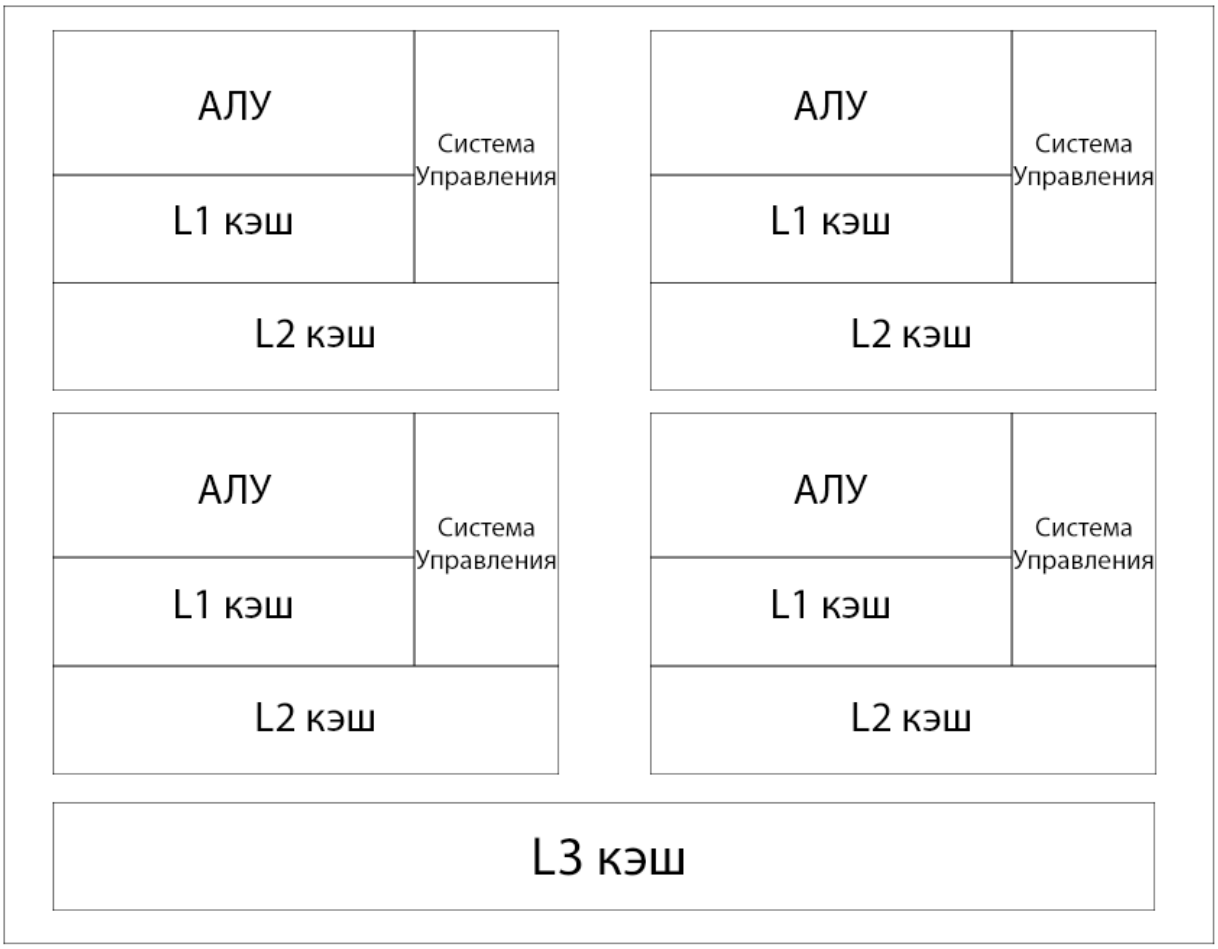

Рис 1.1 Архитектура центрального процессора

В тоже время видеокарта имеет несколько сотен, а иногда и более тысячи, вычислительных ядер. В случае, когда части одной работы делятся

между ядрами, для эффективной работы необходимо определить части работы для каждого из ядер. Причем необходимо, чтобы эти части работы были независимы друг от друга, иначе ядрам может потребоваться дождаться окончания работы других ядер и скорость вычисления может быть даже ниже, чем при последовательном выполнении программы на центральном процессора. На рис. 1.2 можно видеть схематично изображенную архитектуру графического процессора, как можно видеть, его ядра являются гораздо более простыми и предназначены, в основном, для выполнения арифметических операций.

Рис 1.2 Архитектура центрального процессора

Не строго говоря, графический процессор состоит из нескольких мультипроцессоров, в видеокартах последнего поколения их количество уже приближается к 50, а для более старых видеокарт около 10. Каждый

мультипроцессор состоит из некоторого количества вычислительных ядер, причем эти ядра объединены в структурные группы, для наиболее удобного использования.

Каждое ядро имеет поток исполнения, такой поток называются тредом (thread), каждые 32 таких потока объединены в варп (warp). Если использовать прямой перевод, то можно было увидеть, что «канат» состоит из 32 «нитей». Варп представляет собой группу потоков, в котором инструкции выполняются физически одновременно. Помимо варпов, потоки так же образуют блоки, а все потоки одного блока принадлежат одному мультипроцессору. Блоки, в свою очередь, образуют сетку (grid). Если проводить нестрогую аналогию, то можно сравнить мультипроцессор с ядром центрального процессора, каждый из них может иметь собственные данные и поток исполняемых процедур.

1.3.3 Разработка программ на языке CUDA C

Для того, чтобы выполнить процедуру на графическом процессоре, центральный процессор должен подготовить данные и разместить их в памяти GPU. После этого центральный процессор должен вызвать ядро – специальную функцию, которая является точкой входа для программы GPU, каждое ядро должно быть отмечено специальным модификатором __global__. Вызов функции ядра является неблокирующим, то есть процессор продолжит выполнение своей программы после его вызова. Работа графического процессора выстроена так, что гарантируется то, что ядра будут выполнены в порядке их вызова процессором. Если есть необходимость дождаться выполнения всех ядер, например для выгрузки данных обратно в оперативную память компьютера, то для этого существует функция, которая блокирует поток выполнения процессора, пока все ядра не будут завершены.

Графический процессор имеет 6 [12] видов памяти, которые предназначены для использования в разных ситуациях. Некоторые типы

аналогичны кэшам центрального процессора и не поддаются управлению, а некоторые аналогичны оперативной памяти:

1. Глобальная память. Основная DRAM память видеокарты, скорость ее низкая, относительно других видов памяти, но если потоки одного варпа обращаются к непрерывному участку памяти, то это позволяет получить данные за одно вычитывание.
2. Регистровая память. Предназначена для хранения локальных переменных мультипроцессора. Размер ее крайне ограничен, в зависимости от моделей составляет около 32 Кб. Скорость доступа к ней наиболее высокая и обратиться к ней может любой тред.
3. Локальная память. Расположена в DRAM, каждый поток имеет собственную локальную память, используется в случае нехватки регистровой.
4. Разделяемая память. Медленнее регистровой, но скорости обращения сопоставимы. Размер составляет примерно 48 Кб, каждый блок имеет собственную разделяемую память. Механизм доступа к памяти утроен таким образом, что весь объем разделен на 32 банка памяти и обращения к одному и тому же банку выполняются последовательно. То есть если каждый поток обратится к разному банку, то они сразу же получат необходимые данные, а если обратятся к одному и тому же, то каждый прочет данные последовательно, один за другим.
5. Константная память. Аналогично глобальной располагается в DRAM и имеет низкую скорость доступа. Память общая на уровне всего ядра и используется для размещения в ней аргументов запуска. Возможно кеширование данных, так как они не изменяются.

6. Текстурная память. Общая на уровне всего ядра, используется в качестве специального кэша для графических текстур, располагается в DRAM.

Из всех вышеперечисленных видов памяти регистровая и локальная память являются неуправляемыми, то есть нельзя с помощью программ располагать в них конкретные данные. Эти виды памяти управляются на уровне архитектуры самого графического процессора, то есть GPU самостоятельно принимает решение, какие данные сейчас хранить в регистровой памяти, а какие в локальной. В отличие от этих двух видов памяти, остальные являются программируемыми.

Чтобы была возможность выстроить программу так, что все треды могли определить их часть работы, каждое из ядер имеет порядковый номер в блоке, а каждый блок имеет порядковый номер в сетке. Эти данные помещаются в специальные структуры dim3 и uint3, которые имеют поля x, y и z, то есть представляют собой векторы, состоящие из трёх элементов. Это необходимо для возможности иметь двухмерные или трёхмерные сетки, которые бывают удобны при решении определенных задач. Всего каждый поток будет обладать четырьмя такими переменными. Переменяя threadIdx предназначена для хранения индекса треда в рамках одного блока, аналогично blockIdx хранит информацию о координатах блока в сетке. В свою очередь blockDim содержит информацию о размерах блока, а gridDim о размерности сетки.

Имея данные о положении треда в блоке и блока в сетке, можно определить индекс треда в сетке, а зависимости от него будет выполнена соответствующая часть работы. Но если данных больше, чем всего тредов, то часть данных может быть не обработана, поэтому необходимо далее циклично выполнять сдвиг на количество тредов в сетке, пока вся работа не будет выполнена. Для большей наглядности рассмотрим пример

простейшего ядра, который каждому элемента массива присваивает значение, равное его индексу.

```
1. __global__ void device_kernel(int* d_array, int size) {
2.     int idx = blockIdx.x * blockDim.x + threadIdx.x; // позиция треда в
   сетке
3.     int offset = blockDim.x * gridDim.x; // общее количество тредов
4.     for (int i = idx; i < size; i += offset) {
5.         d_array[i] = i;
6.     }
7. }
```

Таким образом каждый элемент массива будет обработан одним из потоков и ни один не будет пропущен.

2. ПРАКТИЧЕСКАЯ ЧАСТЬ

В данной работе была выполнена реализация алгоритма TT-Cross в двух вариантах, последовательном и параллельном. Параллельная версия алгоритма была выполнена при помощи средств графического процессора. Параллельная реализация показала существенный прирост производительности на тестовых примерах.

2.1 Используемые инструменты

Программный комплекс для построения тензорного поезда содержит последовательную реализацию для центрального процессора и параллельную для графического, для них был выбран язык C++ и CUDA C соответственно.

В последовательной реализации была использована библиотека LAPACKE, которая представляет собой интерфейс к открытой библиотеке LAPACK на языке Fortran. Данная библиотека является широко распространенной библиотекой алгоритмов линейной алгебры. За долгое время ее существования она была качественно отлажена и оптимизирована.

Для реализации, выполненной на графическом процессоре, была использована библиотека MAGMA. Данная библиотека является аналогом LAPACK для графических процессоров и предоставляет реализацию алгоритмов линейной алгебры в уже в параллельном варианте. Стоит принимать во внимание то, что обычные графические процессоры предназначены для работы с числами одинарной точности, так как одинарной точности хватает для решения задач компьютерной графики. Поэтому их архитектура устроена так, чтобы наиболее быстро производились операции одинарной точности, но численные методы чаще всего требуют использование чисел двойной точности. Для работы с числами двойной точности компании выпускают специальные научные графические процессоры, которые оптимизированы специально для этого. Были приведены сравнительные тесты производительности [8], которые показали, что MAGMA, при работе с 64-битными вещественными числами не уступает

по производительности библиотеке cuBLAS от разработчиков NVIDIA, при этом имеет более широкий функционал, поэтому к использованию была выбрана именно она.

Программный комплекс был разработан таким образом, чтобы можно было независимо друг от друга использовать модули для CPU и GPU. Модули компилируются отдельно, модуль на C++ компилируется при помощи компилятора GCC, а программа на CUDA C компилируется при помощи nvcc. Для компиляции проекта используется система для автоматизации сборки проекта CMake, которая позволяет использовать разные компиляторы для отдельных модулей, что значительно упрощает разработку.

2.2 Детали реализации

Как уже было сказано, реализации для центрального процессора разнесены по разным модулям и их можно использовать независимо или вместе, но стоит помнить, что загрузка данных в память видеокарты и их выгрузка занимают некоторое время и при возможности лучше всего хранить все данные в памяти видеокарты при ее использовании.

CUDA C позволяет создавать классы, аналогичные классам в C++, при помощи которых можно скрыть от пользователя детали работы с GPU таким образом, что даже не зная язык CUDA C, но зная C++ пользователь сможет воспользоваться разработанным программным комплексом, так как вызовы операций, связанных с графическим процессором, скрыты в методах класса. Кроме этого, алгоритмы были выделены в отдельный подмодуль, причем каждый из них, например MaxVol и крестовая аппроксимация, находятся в отдельной директории.

Говоря о хранении данных, нужно упомянуть как в матрицы и тензоры хранятся в памяти. Так как хочется иметь возможность легко менять размерность тензора, то создание n-мерного массива заранее является

нерациональным решением, так как будет требоваться изменение программы каждый раз. Поэтому применяется линеаризация, то есть отображение многомерных индексов матрицы или тензора в одномерный индекс массива. Например, матрицы хранятся в массиве по столбцам, то есть элемент $A(i,j)$ в одномерном массиве будет иметь индекс $z = i + jm$, где A – матрица, размером $l \times m$. Аналогично для 3-мерного тензора $l \times m \times n$ индекс для элемента $A(i,j,k)$ будет преобразован в линейный индекс $z = i + jm + kmn$. Нетрудно заметить, что организация многопоточной программы для графического процессора с выбором номера потока имеет схожие принципы, поэтому данный метод линеаризации хорошо ложится на GPU. Стоит отметить, что при данном способе хранения операция reshape, которая используется в алгоритме TT-Cross для получения следующей развертки тензора, выполняется за константное время и не требует перестановки элементов тензора.

Алгоритм TT-Cross реализован таким образом, что на вход подаются ранги аппроксимации, так как обычно очень важно иметь их фиксированными. Соответственно, на вход алгоритм принимает сам тензор и массив с рангами. При необходимости можно модифицировать алгоритм так, чтобы ранги подбирались до оптимального значения, например бинарным поиском, но это повлечет дополнительные временные затраты.

2.3 Анализ результатов работы

Тестирование и анализ полученных результатов производились на каждом этапе работы. Реализация производилась от меньшего алгоритма к большему, то есть сначала MaxVol, затем крестовая аппроксимация, затем TT-Cross.

2.3.1 Алгоритм MaxVol

После завершения отладки алгоритма на специально подобранных данных, было произведено нагрузочное тестирование, для которого была использована матрица, значения которой были сгенерированы случайным образом. Для объективности результатов тестирование двух реализаций производилось одних и тех же входных данных.

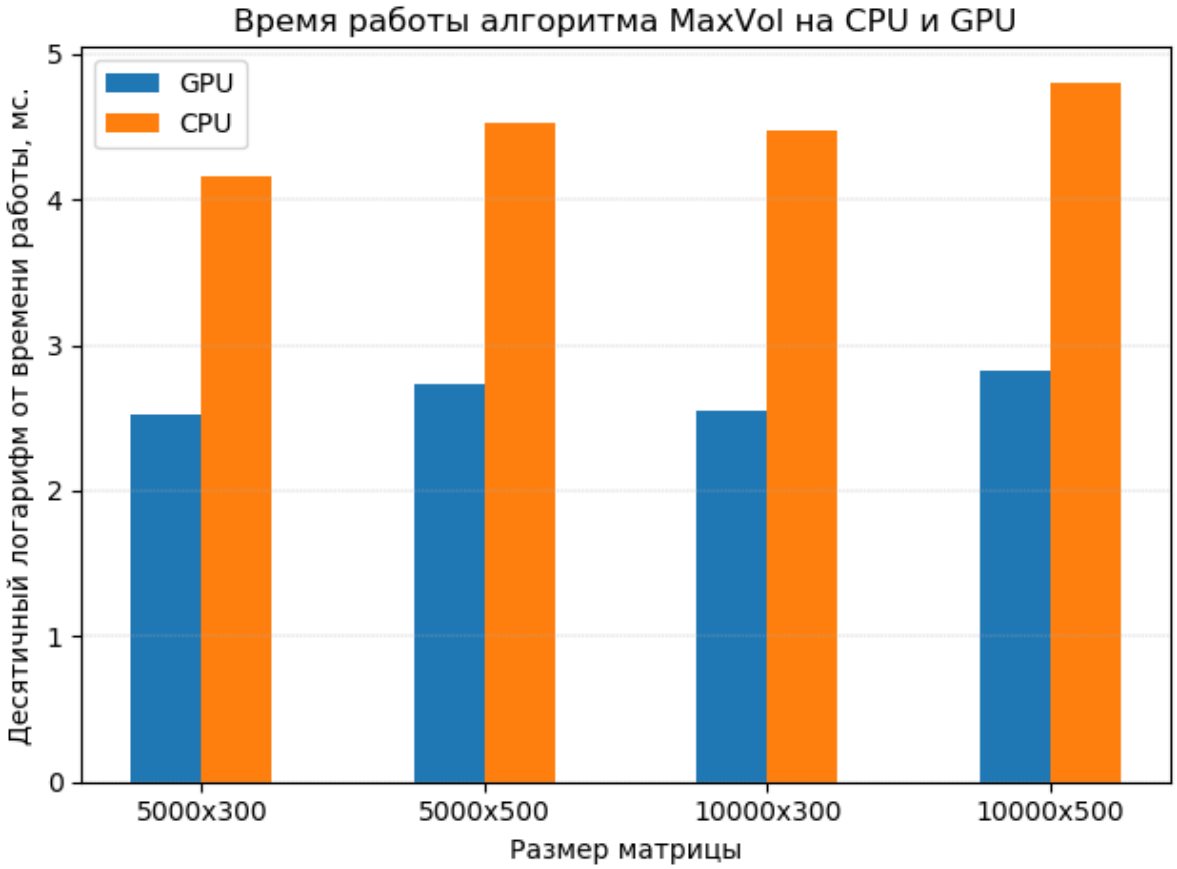

Рис. 2.1 Сравнение времени работы MaxVol для матриц разного размера

Для большей наглядности диаграмм, по оси ординат взят десятичный логарифм. Как можно видеть из рис. 2.1 было получено ускорение более 80 раз. Самой трудозатратой частью алгоритма является обращение верхнего блока матрицы, а данная операция имеет хороший потенциал для распараллеливания, что позволяет показывать отличные результаты.

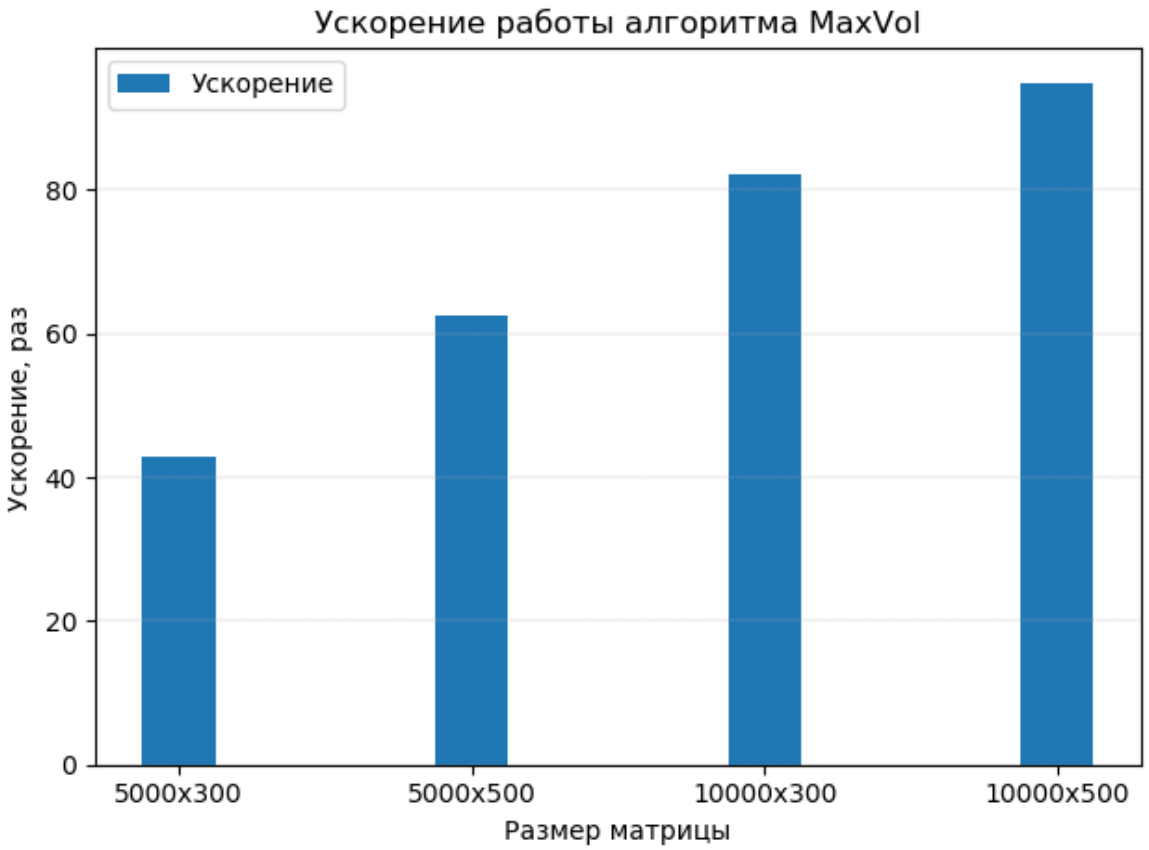

Рис 2.2 Ускорение времени работы MaxVol

2.3.2 Крестовая аппроксимация

Следом за алгоритмом MaxVol была реализована крестовая аппроксимация. Отладку и функциональное тестирование удобнее всего проводить на специально сгенерированных матрицах заданного ранга. Чтобы получить матрицу $m \times n$ ранга r следует перемножить матрицу $m \times r$ на $r \times n$. То есть алгоритм корректно работает, если при ранге аппроксимации большим или равным r матрица будет в точности восстановлена из полученного разложения, причем аппроксимация рангом более чем r должна корректно находиться. А при ранге аппроксимации меньшем r будут возникать погрешности.

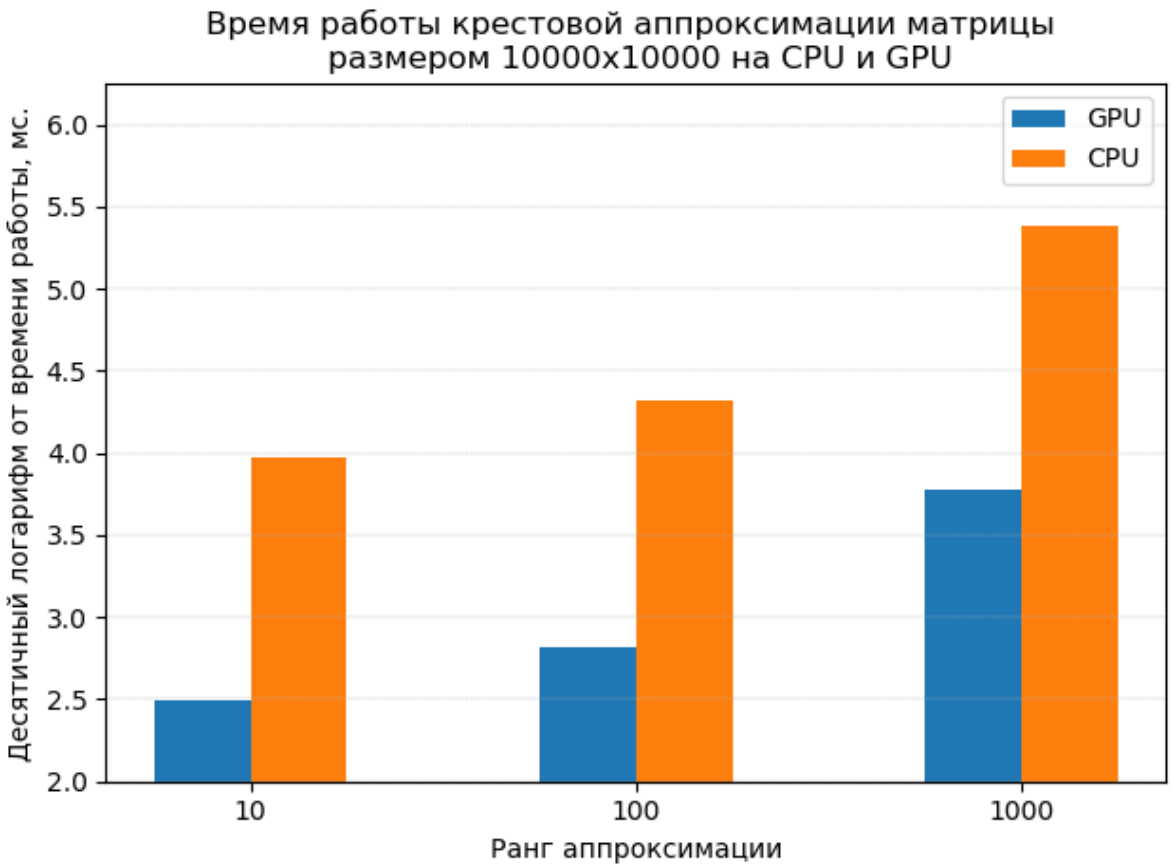

Рис 2.3 Время работы алгоритма крестовой аппроксимации

Проведя исследование при помощи нагрузочного теста на большой матрице, было выявлено, что при ранге 1000, когда размер самой матрицы составляет 10000×10000. Это означает, что можно смягчать ограничение на ранг, используя реализацию для графического процессора. Рассмотрев рис 2.3 и рис. 2.4 можно увидеть, что на ранге 1000 достигается ускорение в 40 раз. То есть время нахождения аппроксимации сокращается примерно с 4 минут до 6 секунд, что является очень значительным приростом. При этом качество восстановления матрицы остается неизменным. Поэлементные разности между исходными и восстановленными матрицами, полученными в результате работы обеих реализаций, составляют около 10^{-16} , что граничит с машинной точностью для 64-битных вещественных чисел. А разница евклидовых норм около 10^{-9}.

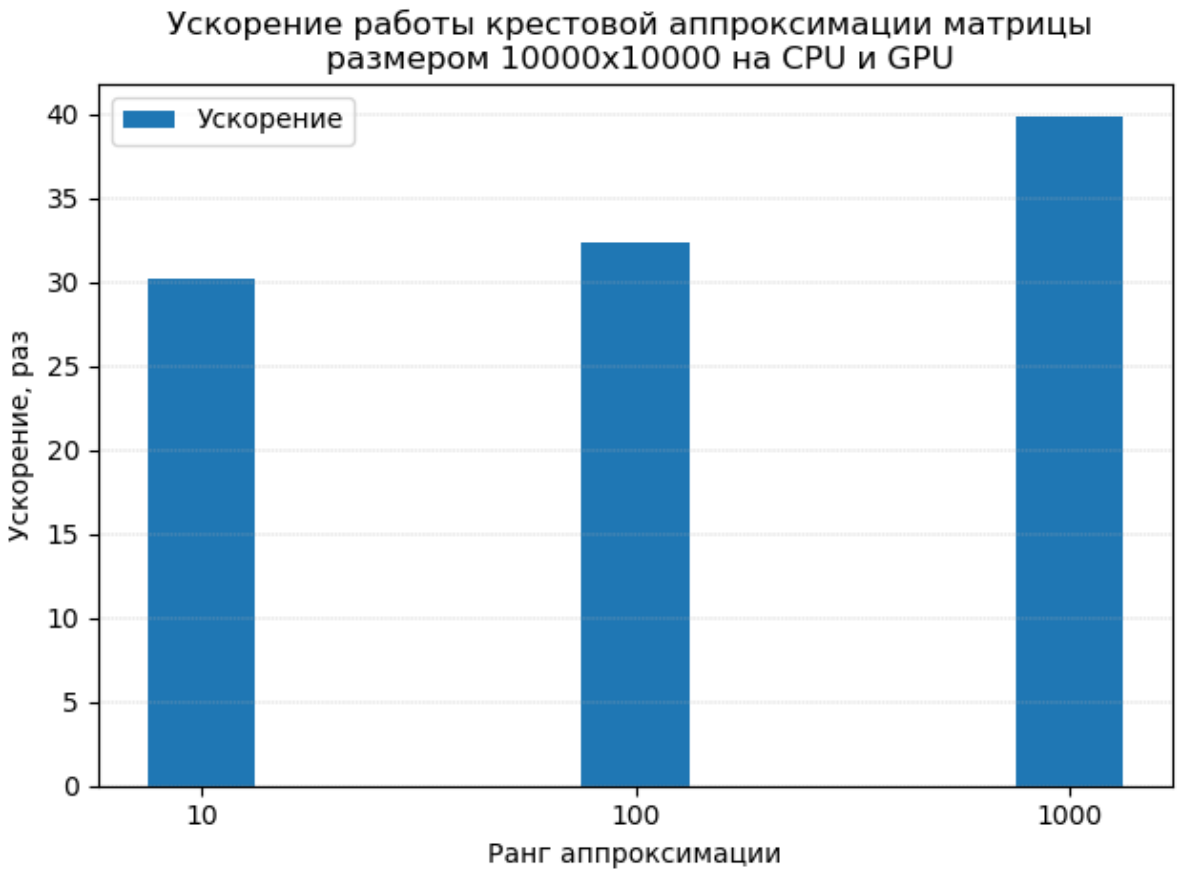

Рис 2.4 Время работы алгоритма крестовой аппроксимации

Аналогичный тест был произведен с фиксированным рангом при увеличении размера матрицы. Для матрицы размером 100×100 реализация на CPU оказалась быстрее, так как подготовка для запуска процедуры на GPU требует большего времени для подготовки и для получения прироста необходимо, чтобы время подготовки процедуры было незначительным по сравнению с объемом выполняемой работы, что, соответственно, можно наблюдать на рис 2.5, аналогично на рис 2.6 для наименьшего размера матрицы ускорение составляет менее единицы, но при увеличении размера матрицы, ускорение возрастает до 25 раз, при том, что ранг является довольно малым относительно размера матрицы.

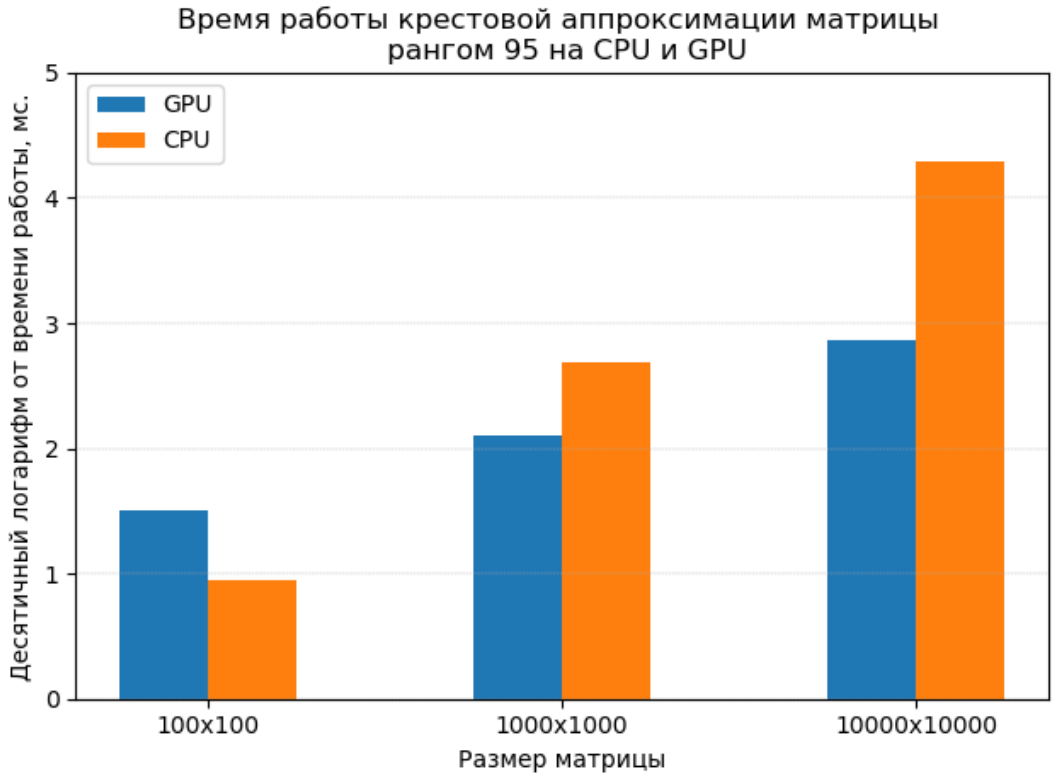

Рис 2.5 Время работы аппроксимации для матриц разного размера

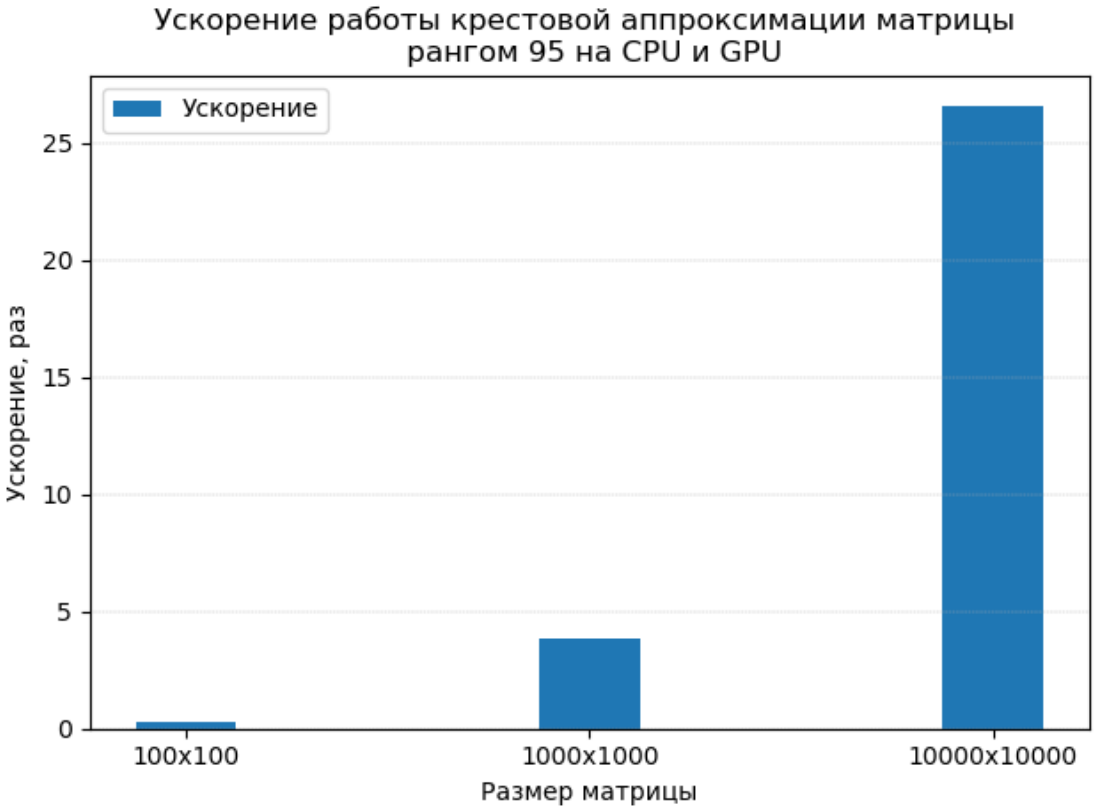

Рис 2.6 Ускорение работы аппроксимации для матриц разного размера

2.3.3 Разложение тензора Гильберта при помощи TT-Cross

Так как ключевым алгоритмом в TT-Cross является крестовая аппроксимация, то успешная ее реализация позволила выполнить реализацию TT-Cross. Первым тензором для тестирования стал тензор Гильберта. Текущее и все последующие тесты выполнялись таким образом, что ранг аппроксимации был подобран так, что по построенному тензорному поезду можно в точности восстановить все элементы исходного тензора. В случае тензора Гильберта ранг будет равен 5. Как можно видеть из рис. 2.7, на таком тензоре достигается десятикратное ускорение.

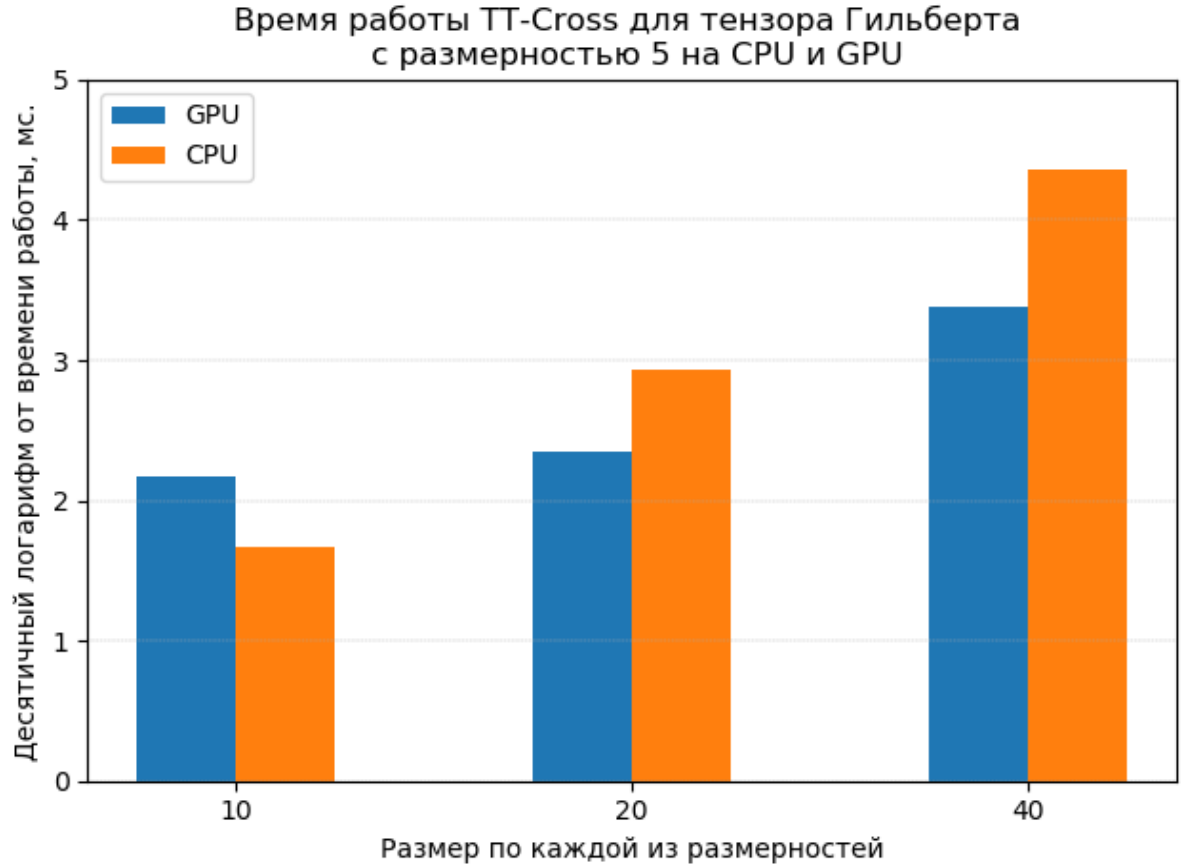

Рис. 2.7 Время разложения тензора Гильберта разного размера

Кроме того, было замерено время восстановления всех элементов тензора. Так как тензор данного размера помещается в память графического процессора и оперативную память компьютера, то было выполнено восстановление всех элементов разом. Как можно видеть, из рис. 2.8, время

работы для наибольшего тензора больше на 2 порядка, то есть, как видно на рис. 2.9 ускорение составляет 100 раз с 3648 миллисекунд до 35.

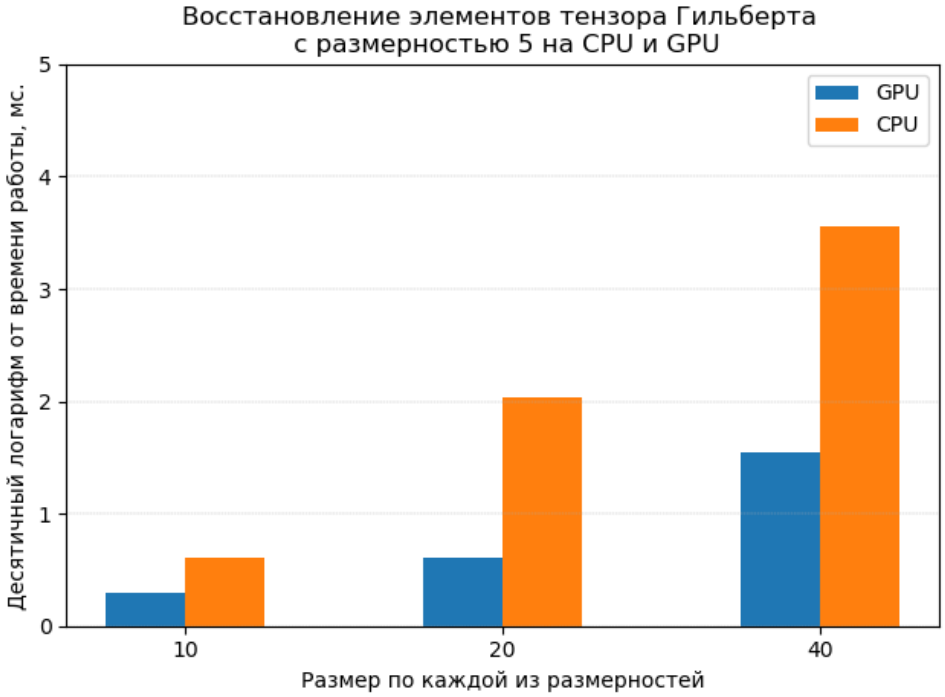

Рис. 2.8 Время восстановления элементов тензора Гильберта

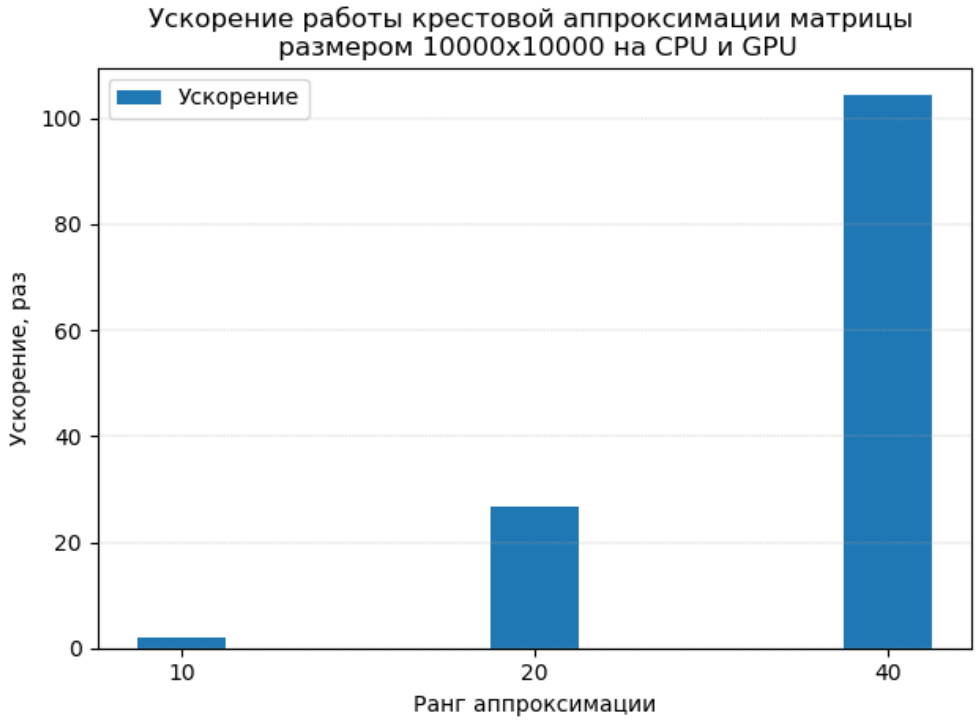

Рис. 2.9 Ускорение восстановления элементов тензора Гильберта

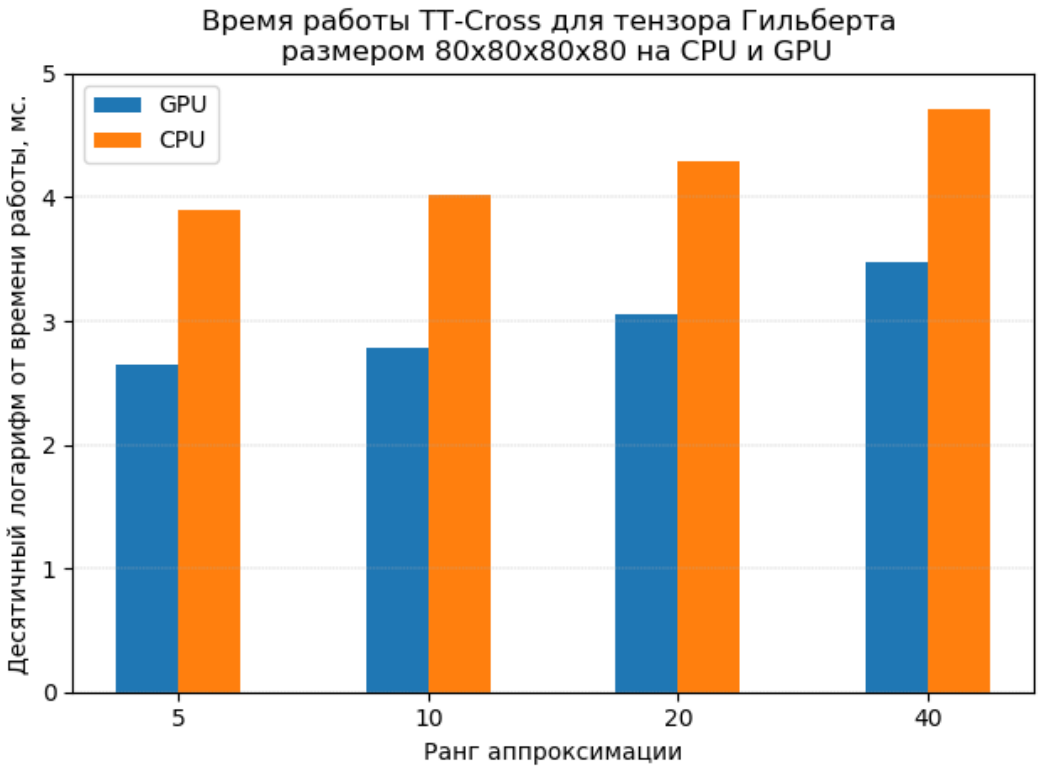

Рис. 2.10 Работа TT-Cross с тензором Гильберта при различных рангах

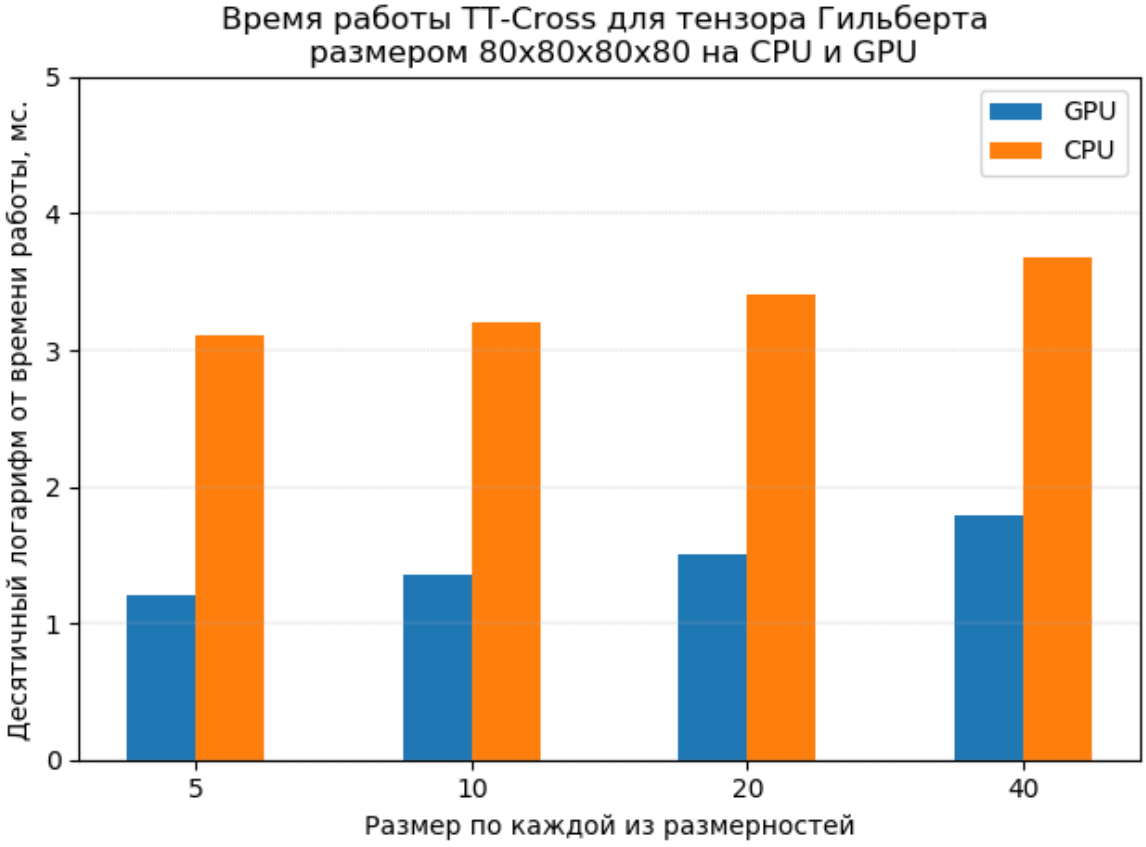

Рис. 2.11 Время восстановления элементов тензора в зависимости от ранга

Аналогичное исследование было проведено для увеличения размерности и ранга аппроксимации. На рис 2.10 и 2.11 можно видеть время, необходимое для приведения тензора Гильберта в формат тензорного поезда. Как можно видеть, при ранге 40 достигается примерно 18 кратное ускорение при построении тензорного поезда. Реализация на центральном процессоре показала результат примерно 51 секунду, а реализация на графическом процессора немного меньше 3 секунд. Восстановление всех элементов было произведено за 54 миллисекунды вместо 4822, что дает ускорение примерно в 90 раз.

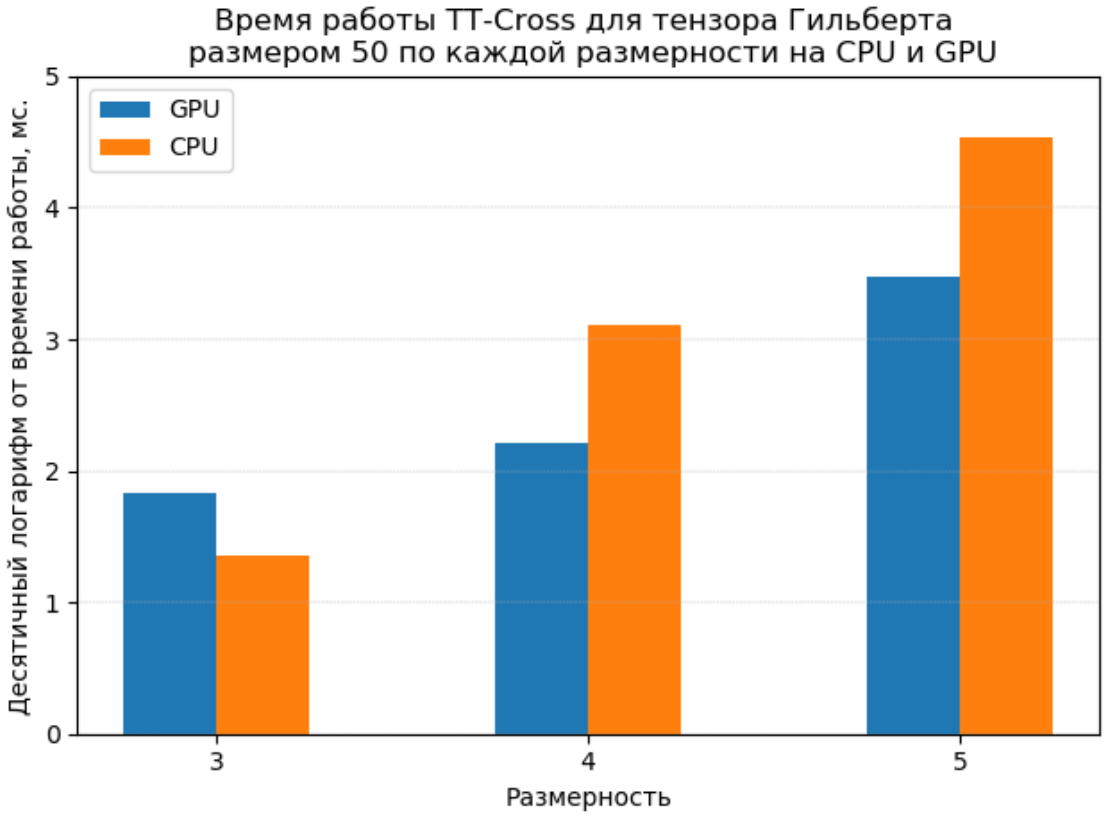

Рис. 2.12 Время разложения тензора в зависимости от размерности

При фиксированных значения размера и ранга при увеличении наблюдаются аналогичные результаты, ускорение построения тензорного поезд составляет около 13 раз, что можно видеть на рис 2.12, а ускорение при восстановлении элементов тензора, что показывает рис 2.13, составляет около 90 раз. Данный эксперимент показал, что ускорение меньше, чем при

увеличении ранга аппроксимации, так как при его увеличении наиболее быстро растёт размер матрицы, что существенно сказывается на производительности последовательной реализации.

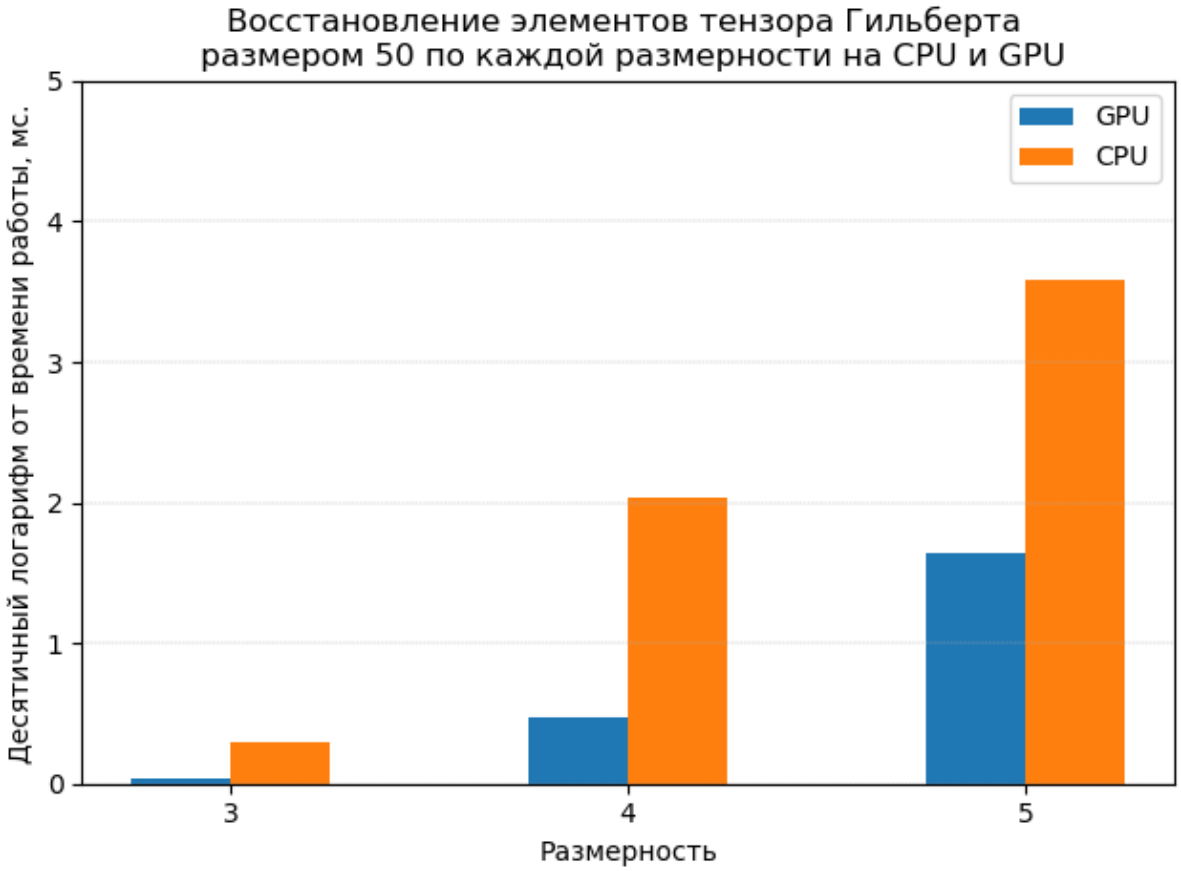

Рис. 2.13 Время восстановления элементов тензора в зависимости от ранга

Проведенное исследование показало, что восстановление элементов тензора происходят значительно быстрее в параллельной реализации алгоритма. Приняв во внимание тот факт, что крестовая аппроксимация не требует вычисления всех элементов тензора, был сделан вывод, что если значения многопараметрической функции образуют тензор неполного ранга, при этом функция является трудно вычислимой в плане необходимого для вычисления значения времени, то построив тензорный поезд, можно будет иметь возможность восстанавливать элементы гораздо быстрее, нежели напрямую вычислять ее элементы. Исходя из данных соображений для дальнейших исследований были выбраны функции Розенброка и Гривонка,

причем функция Гривонка требует больших усилий для вычисления, так как она требует многократных вычислений функций косинуса и экспоненты.

2.3.4 Функция Розенброка

Функция Розенброка является невыпуклой функцией, которую часто используют для проверки качества работы алгоритмом оптимизации, она имеет следующий вид:

$$f(x) = \sum_{i=0}^{d-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2 \right),$$

Где x – d-мерный вектор. Вычислив значения данной функции на сетке, было выявлено, что для восстановления значений функции с точностью 10^{-9} по тензорному поезду достаточно использовать ранги аппроксимации, равные 3. Отсюда были получены следующие результаты.

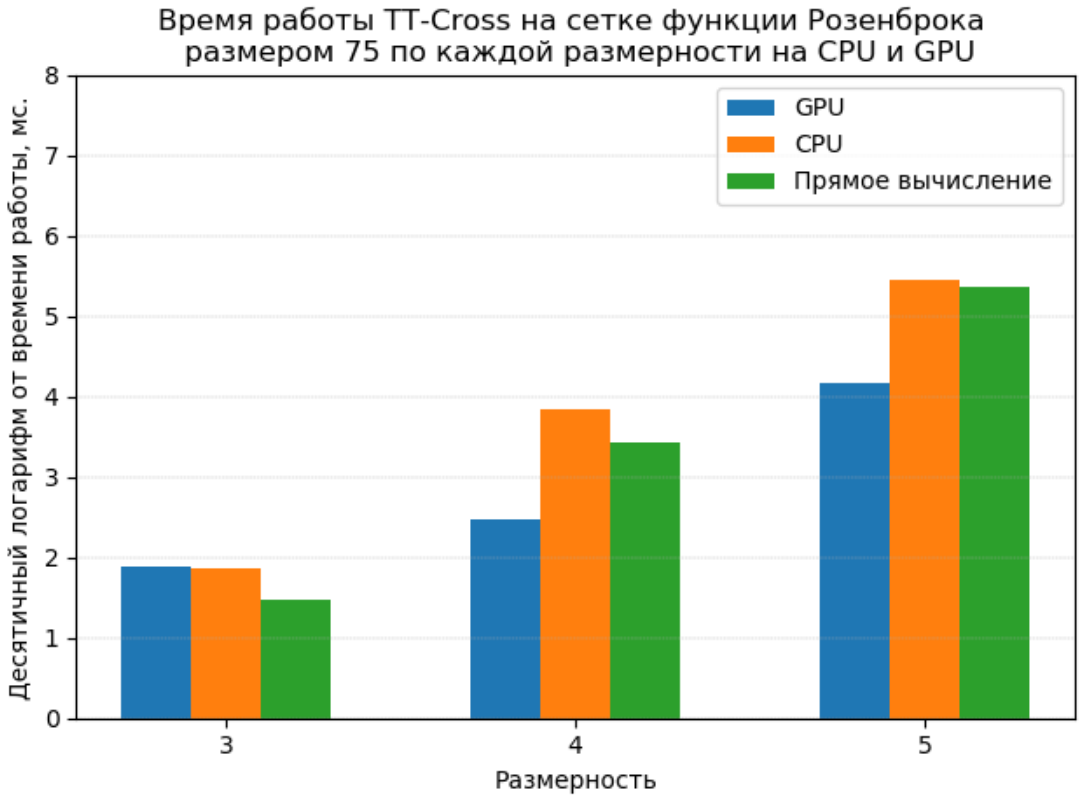

Рис. 2.14 Построение тензорного поезда для сетки на функции Розенброка

Как можно видеть из рис 2.14, для тензора размерностью 5 время построения тензорного поезда на центральном процессоре даже немного превышает время прямого вычисления функции, но параллельный вариант алгоритма строит тензорный поезд гораздо быстрее, в 17 раз быстрее прямого вычисления и в 19 раз быстрее последовательной реализации. Время построение тензорного поезда в параллельном и последовательном варианте составляет примерно 13 и 287 секунд соответственно, а время прямого вычисления элементов 234 секунды. Максимальное ускорение достигается на больших сетках, именно с которыми возникают проблемы при работе. Но стоит отметить, что это только построение функции.

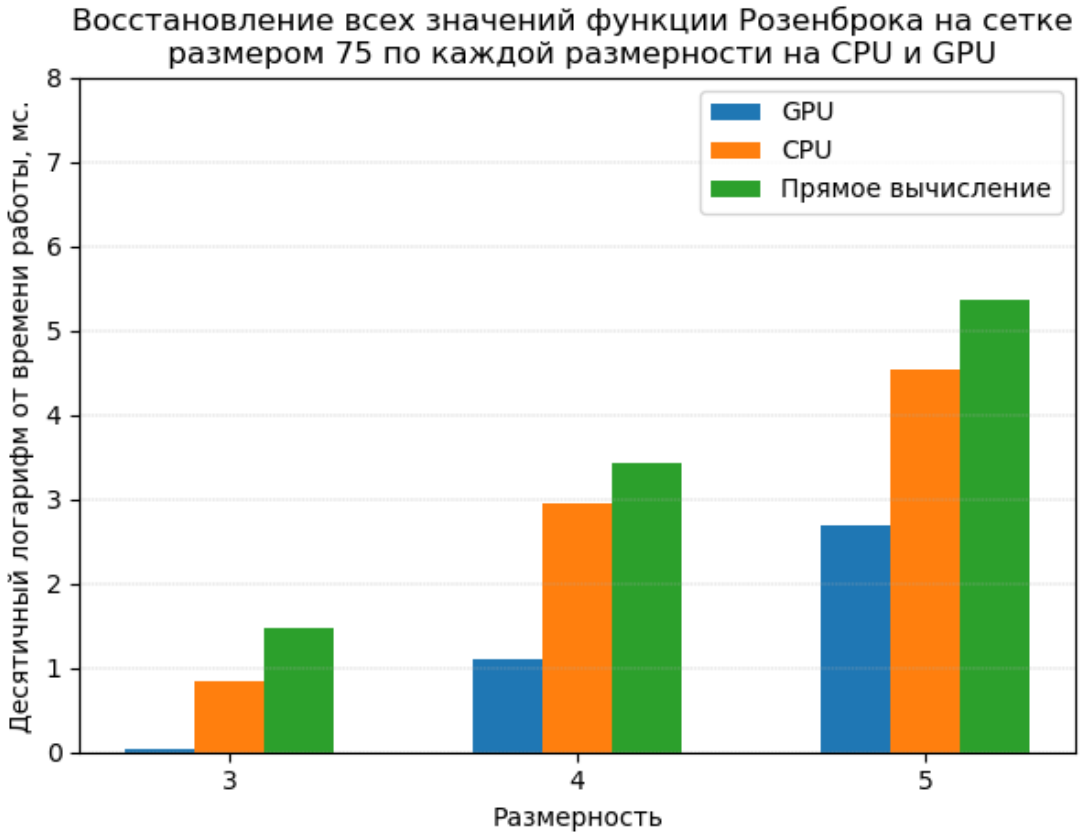

Рис. 2.15 Восстановление всех значений функции Розенброка на сетке

Рассмотрим рис 2.15, как можно видеть, для сетки 75^5 восстановление всех её элементов из тензорного поезда в последовательной и параллельной версии составляет 0.4 и 34 секунды соответственно, то есть ускорение составляет примерно 70 раз. Сложив результаты, полученные в ходе

построения и восстановления тензора получим итоговое время работы для расчета всей сетки. Прямое построение сетки занимает 234 секунды, построение тензорного поезда и восстановление всех элементов сетки в последовательной версии 321 секунду, а в параллельной всего лишь 15 секунд, что даёт ускорение почти в 16 раз. Аналогичные результаты получаются в ходе исследования при увеличении размера сетки стоит учитывать, что функция Розенброка вычисляется довольно быстро, но даже при этих условиях параллельная реализация работает гораздо быстрее.

2.3.5 Функция Гривонка

После проведения исследования с функцией Розенброка, к рассмотрению была выбрана функция Гривонка, как более требовательная к вычислительным мощностям.

$$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right),$$

Где x – d-мерный вектор. Проведя исследования, аналогичные тем, что были проведены для сетки значений функции Розенброка, было выявлено, что для точного восстановления всех значений тензора достаточно аппроксимации рангом 3.

Как можно видеть из рис. 2.16, уже на сетке 75^4 наблюдается выигрыш параллельной реализации по времени уже на 1 порядок, при этом данный выигрыш даже увеличивается при увеличении сетки до 75^5. Построение тензора тензорного поезда для сетки размером 75^5 требует примерно 11 в случае параллельной реализации, а в случае последовательной 220, при том, что прямое вычисление всех значений сетки потребовало 351 секунду. В свою очередь, восстановление, как позано на рис. 2.17 всех элементов заняло около 0.5 и 35 секунд для GPU и CPU соответственно.

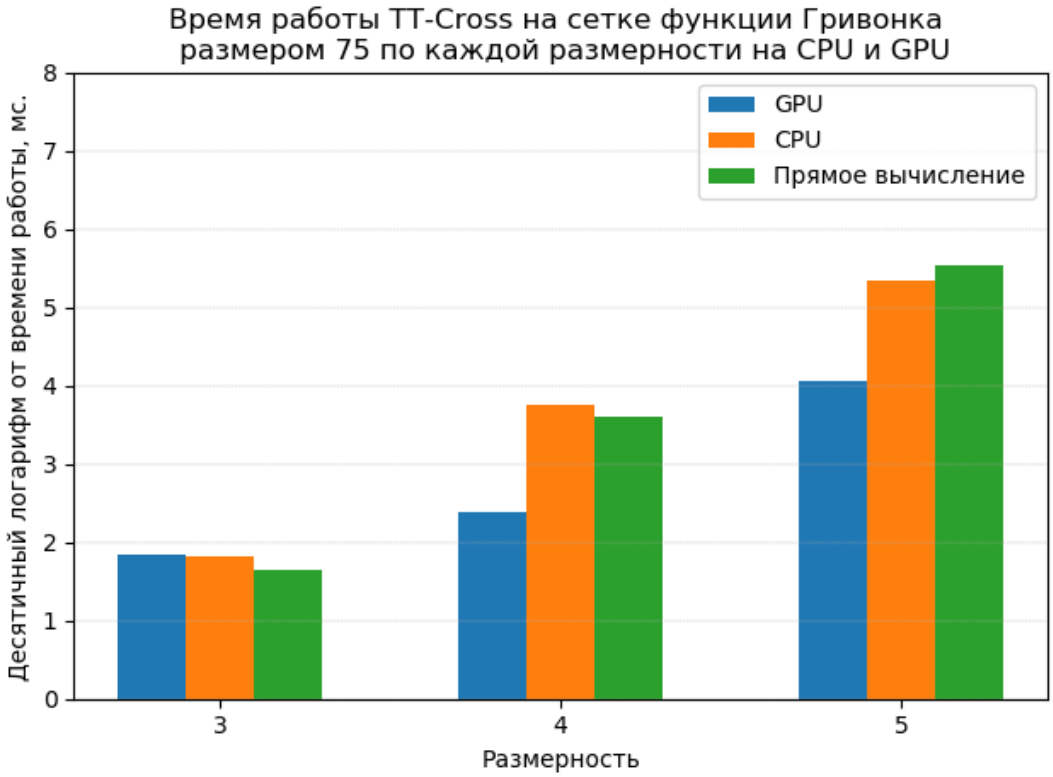

Рис. 2.16 Построение тензорного поезда для сетки на функции Гривонка

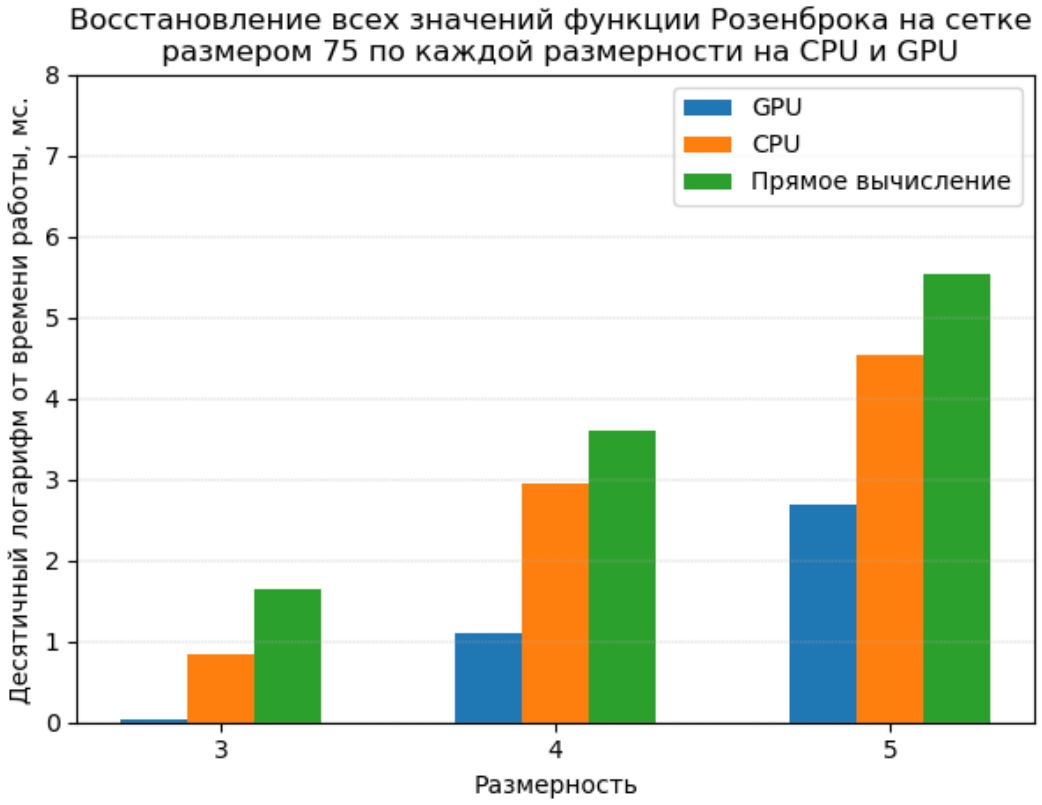

Рис. 2.17 Восстановление всех значений функции Розенброка на сетке

Таким образом итоговое время работы для параллельной реализации составляет 12 секунд, а последовательной 255 секунд, то есть ускорение составляет 29 раз и 1.4 раза соответственно, что говорит о том, что при увеличении времени, требуемого на вычисление функции, ускорение, достигаемое при помощи тензорного поезда, будет возрастать.

Кроме того, следует принять во внимание тот факт, что сетка на 75^5 элементов двойной точности занимает в памяти почти 18 гигабайт, но алгоритм крестовой аппроксимации, позволяет не вычислять все элементы тензора, а только некоторое подмножество, что и было использовано в ходе построения тензорного поезда. Таким образом, используя видеокарту с 6 гигабайтами памяти был обработан тензор, полное представление которого потребовало бы в 3 раза больше памяти, чем максимально допустимый объем. Кроме того, вычисление только части элементов и позволило добиться такого ускорения в параллельной реализации.

ЗАКЛЮЧЕНИЕ

При выполнении данной работы был реализован программный комплекс, позволяющий выполнять построение тензорного поезда при помощи центрального процессора и графического процессора. Таким образом было реализовано две версии алгоритма – последовательная и параллельная.

Разработанное программное обеспечение также было применено к тензорам, образованными значениями сложновычислимых многомерных функций на регулярной сетке. В этом случае был получен прирост скорости в 30 раз. Таким образом данный программный комплекс позволяет выполнять сложные вычисления без использования вычислительных кластеров.

Тестирование проводилось на графическом процессоре прошлого поклонения NVIDIA GTX 1060, содержащим около 1000 вычислительных ядер. В настоящее время компания выпустила гораздо более мощные процессоры RTX, которые имеют более 4000 вычислительных ядер, и больший объем памяти по сравнению с линейкой видеокарт GTX. Использование данного класса видеокарт позволит обрабатывать ещё большие тензоры с помощью разработанного программного комплекса. Увеличение вычислительных мощностей графических процессоров говорит о том, что использование параллельных алгоритмов на их базе будет только расти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Оселедец, И. В. Тензорные методы и их применения : Дис. … док. физ.-мат. наук: 01.01.07 / Иван Валерьевич Оселедец // Учреждение Российской академии наук Институт вычислительной математики РАН, Москва 2009 – 282 с.
2. Tyrtyshnikov E. E., Goreinov S. A., Zamarashki N. L. Pseudo Skeleton Approximations // 6 с.
3. Желтков, Д. А., Тыртышников, Е. Е. Параллельная реализация матричного крестового метода // Вычислительные методы и программирование № 16. С. 369-375
4. Mikhalev A., Oseledets I. V. Rectangular maximum-volume submatrices and their applications // King Abdullah University of Science and Technology, Saudi Arabia. – 2017 – 24 с.
5. Goreinov S. A., Oseledets I. V., Savostyanov D. V., Tyrtyshnikov E. E. How to find a good submatrix // Institute for Computational Mathematics Hong Kong Baptist University, China. – 2008 – 10 с.
6. Stoer J., Bulirsch R. Introduction to Numerical Analysis // Springer-Verlag Berlin Heidelberg. – 2000, С. 380-400
7. Optimizing Performance with Intel® Advanced Vector Extensions // Intel Corporation. – 2014 – 5 с.
8. Chrzeszczyk A., Anders J. Matrix computations on the GPU // Jan Kochanowski University, Poland. – 2017 – 455 с.
9. Oseledets I. V. Tensor-Train Decomposition // Skolkovo Institute of Science and Technology, Moscow 2011 – 24 с.
10. Oseledets I. V., Tyrtyshnikov E. E. TT-cross approximation for multidimensional arrays // Institute of Numerical Mathematics, Russian Academy of Sciences, Moscow. – 2009 – 19 с.
11. Сандерс Д., Кэндрот Э. Технология CUDA в примерах. Введение в программирование графических процессоров. – Изд-во ДМК Пресс, 2012. – 231 с. – 1000 экз. – ISBN 978-5-94074-504-4

12. Морозов А. Ю., Ревизников Д. Л. Методы компьютерного моделирования динамических систем с интервальными параметрами. – М.: Изд-во МАИ, 2019. – 160 с.