Санкт-Петербургский государственный университет

Орачев Егор Станиславович

Выпускная квалификационная работа

Реализация алгоритма поиска путей в графовых базах данных через тензорное произведение на GPGPU

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2017 «Программная инженерия»*

Научный руководитель:
Доцент кафедры информатики, к. ф.-м. н. С. В. Григорьев

Рецензент:
Разработчик биоинформатического ПО, ЗАО “БИОКАД” А.С. Хорошев

Санкт-Петербург
2021

Saint Petersburg State University

Egor Orachev

Bachelor's Thesis

Context-Free path querying by tensor product for graph databases on GPGPU

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2017 «Software Engineering»*

Scientific supervisor:
C.Sc., docent Semyon Grigorev

Reviewer:
Bioinformatics Software Engineer, BIOCAD A.S. Khoroshev

Saint Petersburg
2021

Оглавление

Введение

Все чаще современные системы аналитики и рекомендаций строятся на основе анализа данных, структурированных с использованием *графовой модели*. В данной модели основные сущности представляются вершинами графа, а отношения между сущностями — ориентированными ребрами с различными метками. Подобная модель позволяет относительно легко и практически в явном виде моделировать сложные иерархические структуры, которые не так просто представить, например, в классической *реляционной модели*. В качестве основных областей применения графовой модели можно выделить следующие: графовые базы данных [4], анализ RDF данных [6], биоинформатика [23] и статический анализ кода [13].

Поскольку графовая модель используется для моделирования отношений между объектами, при решении прикладных задач возникает необходимость в выявлении неявных взаимоотношений между объектами. Для этого формируются запросы в специализированных программных средствах для управления графовыми базами данных. В качестве запроса можно использовать некоторый *шаблон* на путь в графе, который будет связывать объекты, т.е. выражать взаимосвязь между ними. В качестве такого шаблона можно использовать формальные грамматики, например, регулярные или контекстно-свободные (КС). Используя вычислительно более выразительные грамматики, можно формировать более сложные запросы и выявлять нестандартные и скрытые ранее взаимоотношения между объектами. Например, *same-generation queries* [1] могут быть выражены КС грамматиками, в отличие от регулярных.

Результатом запроса может быть множество пар объектов, между которыми существует путь в графе, удовлетворяющий заданным ограничениям. Также может возвращаться один экземпляр такого пути для каждой пары объектов или итератор всех путей, что зависит от семантики запроса. Поскольку один и тот же запрос может иметь разную семантику, требуются различные программные и алгоритмические средства для его выполнения.

Запросы с регулярными ограничениями изучены достаточно хорошо, языковая и программная поддержка выполнения подобных запросов присутствует в некоторых в современных графовых базах данных. Однако, полноценная поддержка запросов с КС ограничениями до сих пор не представлена. Существуют алгоритмы [3,6,7,15,17] для вычисления запросов с КС ограничениями, но потребуется еще время, прежде чем появиться полноценная высокпроизводительная реализация одного из алгоритмов, способная обрабатывать реальные графовые данные.

Работы Никиты Мишина и др. [11] и Арсения Терехова и др. [8] показывают, что реализация алгоритма Рустама Азимова [3], основанного на операциях линейной алгебры, с использованием GPGPU для выполнения наиболее вычислительно сложных частей алгоритма, дает *существенный* прирост в производительности.

Недавно представленный алгоритм [7] для вычисления запросов с КС ограничениями также полагается на операции линейной алгебры: тензорное произведение, матричное умножение и сложение в булевом полукольце. Данный алгоритм в сравнении с [8] позволяет выполнять запросы для всех ранее упомянутых семантик и потенциально поддерживает бóльшие по размеру КС запросы.

Для его реализации на GPGPU требуются высокопроизводительные библиотеки операций линейной алгебры. Подобные инструменты для работы со стандартными типами данных, такими как *float*, *double*, *int* и *long*, уже представлены. Однако библиотека, которая бы работала с разреженными данными и имела специализацию указанных ранее операций для булевых значений, еще не разработана.

Поэтому важной задачей является не только реализация перспективного алгоритма [7] на GPGPU, но и разработка библиотеки примитивов булевой алгебры, которая позволит реализовать этот и подобные алгоритмы на данной вычислительной платформе.

1. Цель и задачи

Целью данной работы является реализация алгоритма поиска путей в графовых базах данных через тензорное произведение на GPGPU. Для ее выполнения были поставлены следующие задачи.

- Разработка архитектуры библиотеки примитивов разреженной линейной булевой алгебры для вычислений на GPGPU.
- Реализация библиотеки в соответсвии с разработанной архитектурой и алгоритма поиска путей с КС ограничениями через тензорное произведение с использованием разработанной библиотеки.
- Экспериментальное исследование полученных результатов.

2. Обзор предметной области

2.1. Терминология

Ориентированный граф с метками $\mathcal{G} = \langle V, E, L \rangle$ это тройка объектов, где V конечное непустое множество вершин графа, $E \subseteq V \times L \times V$ конечное множество ребер графа, L конечное множество меток графа. Здесь и далее будем считать, что вершины графа индексируются целыми числами, т.е. $V = \{0 \ ... \ |V| - 1\}$.

Граф $\mathcal{G} = \langle V, E, L \rangle$ можно представить в виде матрицы смежности M размером $|V| \times |V|$, где $M[i, j] = \{l \mid (i, l, j) \in E\}$. Используя булеву матричную декомпозицию, можно представить матрицу смежности в виде набора матриц $\mathcal{M} = \{M^l \mid l \in L, M^l[i, j] = 1 \iff l \in M[i, j]\}$.

Путь π в графе $\mathcal{G} = \langle V, E, L \rangle$ это последовательность ребер e_0, e_1, e_{n-1}, где $e_i = (v_i, l_i, u_i) \in E$ и для любых $e_i, e_{i+1} : u_i = v_{i+1}$. Путь между вершинами v и u будем обозначать как $v\pi u$. Слово, которое формирует путь $\pi = (v_0, l_0, v_1), ..., (v_{n-1}, l_{n-1}, v_n)$ будем обозначать как $\omega(\pi) = l_0 ... l_{n-1}$, что является конкатенацией меток вдоль этого пути π.

Контекстно-свободная (КС) грамматика $G = \langle \Sigma, N, P, S \rangle$ это четверка объектов, где Σ конечное множестве терминалов или алфавит, N конечное множество нетерминалов, P конечное множество правил вывода вида $A \to \gamma, \gamma \in (N \cup \Sigma)^*$, $S \in N$ стартовый нетерминал. Вывод слова w в грамматике из нетерминала S применением одного или нескольких правил вывода обозначается как $S \to_G^* w$.

Язык L над конечным алфавитом символов Σ — множество слов, составленных из символов этого алфавита, т.е. $L \subseteq \{w \mid w \in \Sigma^*\}$. Язык, задаваемый грамматикой G, обозначим как $L(G) = \{w \mid S \to_G^* w\}$.

2.2. Поиск путей с ограничениями

При вычислении запроса на поиск путей в графе $\mathcal{G} = \langle V, E, L \rangle$ в качестве ограничения выступает некоторый язык L, которому должны удовлетворять результирующие пути.

Поиск путей в графе с семантикой **достижимости** — это поиск всех таких пар вершин (v, u), что между ними существует путь $v\pi u$ такой, что $\omega(\pi) \in L$. Результат запроса обозначается как $R = \{(v, u) \mid \exists v\pi u : \omega(\pi) \in L\}$.

Поиск путей в графе с семантикой **всех путей** — это поиск всех таких путей $v\pi u$, что $\omega(\pi) \in L$. Результат запроса обозначается как $\Pi = \{v\pi u \mid v\pi u : \omega(\pi) \in L\}$. Необходимо отметить, что множество Π может быть бесконечным, поэтому в качестве результата запроса предполагается не всё множество в явном виде, а некоторый *итератор*, который позволит последовательно извлекать все пути.

Семантика **одного пути** является ослабленной формулировкой семантики всех путей, так как для получения результата достаточно найти всего один путь вида $v\pi u : \omega(\pi) \in L$ для каждой пары $(v, u) \in R$.

Поскольку язык L может быть бесконечным, при составлении запросов используют не множество L в явном виде, а некоторое правило формирования слов этого языка. В качестве таких правил и выступают регулярные выражения или КС грамматики. При именовании запросов отталкиваются от типа правил, поэтому запросы именуются как регулярные или КС соответственно.

2.3. Существующие решения

Впервые проблема выполнения запросов с контекстно-свободными ограничениями была сформулирована в 1990 году в работе Михалиса Яннакакиса [29]. С того времени были представлены многие работы, в которых так или иначе предлагалось решение данной проблемы. Однако в недавнем исследовании Йохем Куиджперс и др. [12] на основе сравнения нескольких алгоритмов [3, 15, 25] для выполнения запросов с контекстно-свободными ограничениями заключили, что существующие алгоритмы неприменимы для анализа реальных данных в силу того, что обработка таких данных занимает значительное время. Стоит отметить, что алгоритмы, используемые в статье, были реализованы на языке программирования *Java* и исполнялись в среде *JVM* в однопоточном режиме, что не является сколь-угодно производительным решением.

Это подтверждают результаты работы Арсения Терехова и др. [8], в которой с использование программных и аппаратных средств Nvidia Cuda был реализован алгоритм Рустама Азимова [3]. В данном алгоритме задача поиска путей с КС ограничениями была сведена к операциям линейной алгебры, что позволило использовать высокопроизводительные библиотеки для выполнения данных операций на GPGPU.

Алгоритм Рустама Азимова [8] способен выполнять запросы только в семантике одного пути. Поскольку в качестве формализма для представления грамматики КС запроса используется *ослабленная нормальная форма Хомского (ОНФХ)* [16], увеличение числа правил в исходной грамматике запроса может приводить к существенному разрастанию ОНФХ, что негативно влияет на время работы алгоритма.

2.4. Поиск путей с КС ограничениями через тензорное произведение

Представленный в работе Егора Орачева и др. [7] алгоритм для выполнения КС запросов использует операции линейной булевой алгебры: произведение Кронекера (частный случай тензорного произведения), матричное умножение и сложение. Данный алгоритм позволяет выполнять запросы в семантике достижимости и всех путей, а также он подходит для реализации на многоядерных системах, что делает его потенциально применимым для анализа реальных данных. Кроме этого, данный алгоритм использует в качестве формализма для представления запроса *рекурсивный автомат* (РА) [2], что потенциально может решить проблему разрастания исходной грамматики запроса.

Идея алгоритма состоит в *пересечении* РА и графа с использованием некоторой модификации классического алгоритма пересечения рекурсивных автоматов [16]. Пересечение выполняется с использованием произведения Кронекера, а множество рекурсивных вызовов учитывается с помощью транзитивного замыкания, что также выражается с использованием матричных операций умножения и поэлементного сложения. Данный процесс итеративный, и он выполняется до тех пор, пока результат не достигнет фиксированной точки.

В листинге 1 представлен псевдокод алгоритма. Необходимо отметить, что алгоритм использует булеву матричную декомпозицию в строках **3 – 4** для представления матрицы переходов РА и матрицы смежности графа, а также использует матричное умножение, сложение и произведение Кронекера в строках **14 – 16**.

Данный алгоритм является относительно простым в реализации, так как всю сложность выполнения он перекладывает на операции линейной алгебры, которые должны быть реализованы в сторонних высокопроизводительных библиотеках.

Listing 1 Поиск путей через произведение Кронекера

1: **function** KroneckerProductBasedCFPQ(G, $\mathcal{G}$)
2: $R \leftarrow$ Рекурсивный автомат для грамматики G
3: $\mathcal{M}_1 \leftarrow$ Матрица переходов R в булевой форме
4: $\mathcal{M}_2 \leftarrow$ Матрица смежности $\mathcal{G}$ в булевой форме
5: $C_3 \leftarrow$ Пустая матрица
6: **for** $s \in \{0, ..., dim(\mathcal{M}_1) - 1\}$ **do**
7: **for** $S \in getNonterminals(R, s, s)$ **do**
8: **for** $i \in \{0, ..., dim(\mathcal{M}_2) - 1\}$ **do**
9: $M_2^S[i, i] \leftarrow \{1\}$
10: **end for**
11: **end for**
12: **end for**
13: **while** Матрица смежности $\mathcal{M}_2$ изменяется **do**
14: $\mathcal{M}_3 \leftarrow \mathcal{M}_1 \otimes \mathcal{M}_2$ ▷ Вычисление произведения Кронекера
15: $M_3' \leftarrow \bigvee_{M_3^a \in \mathcal{M}_3} M_3^a$ ▷ Слияние матриц в одну булеву матрицу достижимости
16: $C_3 \leftarrow transitiveClosure(M_3')$ ▷ Транзитивное замыкание для учета рекурсивных вызовов
17: $n \times n \leftarrow \dim(M_3)$
18: **for** $(i, j) \mid C[i, j] \neq 0$ **do**
19: $s, f \leftarrow getStates(C_3, i, j)$
20: $x, y \leftarrow getCoordinates(C_3, i, j)$
21: **for** $S \in getNonterminals(R, s, f)$ **do**
22: $M_2^S[x, y] \leftarrow \{1\}$
23: **end for**
24: **end for**
25: **end while**
26: **return** $\mathcal{M}_2, C_3$
27: **end function**

2.5. Вычисления на GPGPU

GPGPU (от англ. General-purpose computing on graphics processing units) — техника использования графического процессора видеокарты компьютера для осуществления неспециализированных вычислений, которые обычно проводит центральный процессор. Данная техника позволяет получить значительной прирост производительности, когда необходимо обрабатывать большие массивы данных с фиксированным набором команд по принципу *SIMD*.

Исторически видеокарты в первую очередь использовались как графические ускорители для создания высококачественной трехмерной графики в режиме реального времени. Позже стало ясно, что мощность графического процессора можно использовать не только для графических вычислений. Так появились программируемые вычислительные блоки (англ. compute shaders), которые позволяют выполнять на видеокарте неграфические вычисления.

На данный момент существует несколько промышленных стандартов для создания программ, использующих графический процессор, одними из которых являются Vulkan [28], OpenGL [27], DirectX [10] как API для графических и неспециализированных вычислительных задач, а также OpenCL [22], Nvidia Cuda [19] как API для неспециализированных вычислений.

В качестве GPGPU в этой работе используется Nvidia Cuda. В то время как OpenCL создавался как кросс-платформенный стандарт для программирования вычислений, Cuda API специфично только для видеокарт производства компании Nvidia, однако оно имеет более широкий набор инструментов как для написания, так и для отладки программ, а также собственный компилятор NVCC, который позволяет осуществлять кросс-компиляцию кода на языке Cuda, и прозрачно использовать его вместе с кодом на языке C/C++. Кроме этого, в данной работе используется результаты исследования Арсения Терехова и др. [8], в котором также использовалось Cuda API.

2.6. Библиотеки линейной алгебры для GPGPU

Для эффективной реализации алгоритмов [3,7] требуются высокопроизводительные библиотеки операций линейной алгебры. Реальные графовые данные насчитывают порядка $10^5 - 10^9$ вершин и являются сильно разреженными, т.е. количество ребер в графе сравнимо с количеством вершин, поэтому плотные матрицы не подходят для представления такого типа данных.

Библиотеки *cuSPARSE* [20] и *CUSP* [9] для платформы Nvidia и *clSPRARSE* [5] для платформы OpenCL предоставляют функциональность для работы с разреженными данными, однако они фокусируются на обработке численных данных и специализируются только на стандартных типах, таких как *float*, *double*, *int* и *long*. Для реализации алгоритмов [3, 7] требуются операции над разреженными булевыми матрицами, поэтому требуется специализация вышеуказанных библиотек для работы с булевыми примитивами. С одной стороны, библиотека cuSPARSE имеет закрытый исходный код, что делает невозможным ее модификацию, с другой стороны, библиотеки CUSP и clSPARSE имеет открытый исходный код и свободную лицензию, однако используемые ими алгоритмы умножения разреженных матриц *достаточно* требовательны к ресурсам памяти [8], что делает их неприменимым для обработки данных большого размера.

В работе Арсения Терехова и др. [8] была предпринята попытка самостоятельно реализовать алгоритм умножения разреженных матриц *Nsparse*, предложенный в работе Юсуке Нагасака и др. [21], и специализировать его для булевых значений. Данный алгоритм эксплуатирует возможности видеокарт Nvidia и засчет бóльшего количества шагов обработки позволяет снизить количество расходуемой видеопамяти. Эксперименты показали, что подобный подход позволяет не только снизить в разы количество расходуемой видеопамяти, но и снизить общее время работы алгоритма Рустама Азимова [3] по сравнению с его реализацией на *CUSP*.

3. Архитектура библиотеки

Архитектура библиотеки представлена на рис. 1. Структура библиотеки и ее конечначная функциональность в основном определяется следующими высокоуровневыми требованиями, которые продиктованы как конечными вычислительными задачами на GPGPU, так и наличием существующей инфраструктуры для осуществления экспериментов [30].

- Поддержка вычислений на Cuda-девайсе.
- Поддержка вычислений на CPU.
- C-совместимое API для работы с библиотекой.
- Python-пакет для работы с примитивами и операциями библиотеки в управляемой высокоуровневой среде языка Python.
- Поддержка логирования, функций для отладки и протопирования конечных пользовательских алгоритмов.

3.1. Компоненты

Core

Класс **Library** поддерживает глобальное состояние библиотеки, осуществляет конфигурацию и инициализацию, выбор конкретного вычислительного бэкенда, первичную валидацию вызовов функций и входных данных пользователя, а также хранит все созданные пользователем объекты.

Класс **Matrix** является proxy-классом, который осуществяет доступ к операциям конкретного вычислительного модуля, выбранного пользователем на этапе инициализации всей библиотеки. Данный подход позволяет не только динамически выбирать платформу вычислений, но и позволяет осуществлять дополнительную обработку ошибок, а также поддерживать дополнительные операции над матрицами.

Класс **Logger** осуществляет логгирование в выбранный пользователем текстовый файл в процессе использования функций библиотеки, а также позволяет профилировать операций и также сохранять время их выполенения в текстовом виде.

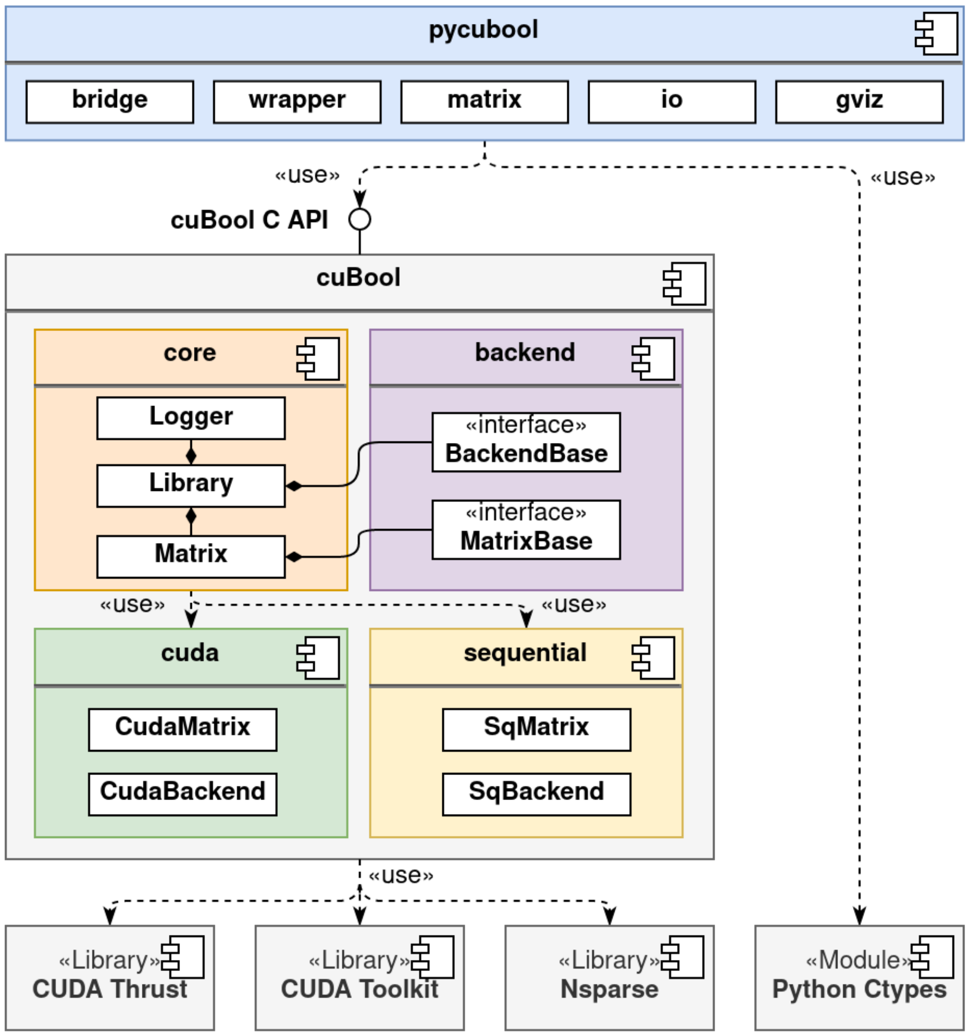

Рис. 1: Архитектура разработанной библиотеки

Backend

Интерфейс **MatrixBase** предоставляет набор основных функций и операций, которые каждый вычислительный модуль должен реализовывать, чтобы предоставляемые им матрицы можно было использовать в **Core** непосредственно для вычислений.

Интерфейс **BackendBase** описывает базовый контракт, которой должен предоставлять вычислительный модуль. Данный интерфейс включает в себя функции для создания и удаления матриц, спефичных для этого модуля, а также функции для корректной инициализации, поддержания глобального состояния и завершения работы.

Cuda

Класс **CudaMatrix** предоставляет реализацию матрицы и операций для осуществления вычислений на Cuda-девайсе. **CudaMatrix** хранит структуру и данные матрицы (ненулевые элементы) в видео-памяти и использует Nvidia GPU для вычислений.

Данный вычислительный модуль выбирается по умолчанию, если в компьтере пользователся имеется Cuda-девайс. Однако пользователь всегда может выбрать **Sequential** вычисления, если это требуется.

Sequential

Предоставляет реализацию класса матрицы и операций над ней для вычислений на CPU. Все вычислений осуществляются последовательно, в однопоточном режиме, что не требует дополнительных библиотек или компонентов.

Данных вычислительный модуль используется по умолчанию на устройствах без Cuda-девайса. Данный подход позволяет использовать библиотеку всем пользователям без исключения. Также данный подход может быть удобен для прототипирования алгоритмов на локальном компьютере, чтобы позже запустить вычисления на высокопроизводительном сервере с поддержкой Cuda.

Pycubool

Python-пакет предоставляет доступ к примитивам и операциям библиотеки в языковой среде Python. Модуль **matrix** предоставляет доступ к классу матрицы и основным операциям, дотсупным в C API. Модуль **bridge** осуществляет коммуникацию с библиотекой через механизмы вызова нативных методов. Модуль **wrapper** поддерживает глобальное состояние библиотеки во время работы Python-интерпретатора. Модули **io** и **gviz** предоставляют доступ к операциям ввода/вывода данных, позволяют загружать или сохранять матрцы в текстовом формате, а также экспортировать набор матриц в виде графа в формате GraphViz, что может быть полезно для отладки пользовательских алгоритмов.

3.2. Последовательнось обработки операций

На рис. 2 представлена последовательность обработки вычислительной операции над матрицей на Cuda-девайсе.

Пользовательский Python-код инициирует выполнение операции над мамтрицей или несколькими матрицами. Этот вызов обрабатывает пакет **pycubool**, который осуществляет первичную базовую валидацию аргументов, запаковывает их и передает в нативную функцию **cuBool C API**. На стороне реализации данного интерфейса полученные аргументы приводятся к требуемому типу и передаются далее в модуль **Core**, который поддерживает состояние библиотеки, осуществяет валидацию аргументов, а также определяет допустимость выполнения операции. Далее вызов передается непосредственно вычислительному модулю **Cuda**, который осуществляет подготовку и непосредственный запуск вычислений на сороне **Nvidia GPU**.

Когда вычисления завершаются, **Cuda**-модуль обновляет состояние матриц в соответсвии с полученными результатами. Модуль **Core** осуществяет финальное логирование операции, а также сохраняет временные показатели выполнения вычислений в файл (опционально), и возвращает в качестве результата выполнения статус операции или возможное исключение, которое могло возникнуть на этапе выполнения операции. **cuBool C API** осуществяет финальную обработку исключения (если таковое возникло), и возвращет вызываещему числовой идентефикатор статуса операции.

В результате выполнения операции **pucubool** уведомляет пользователя о потенциально возникших ошибках и возвращает управление из вызываемой функции. Обновленное состояние библиотеки находится в **Core**, а состояние матриц после выполнения операций хранится на стороне **Cuda**-модуля.

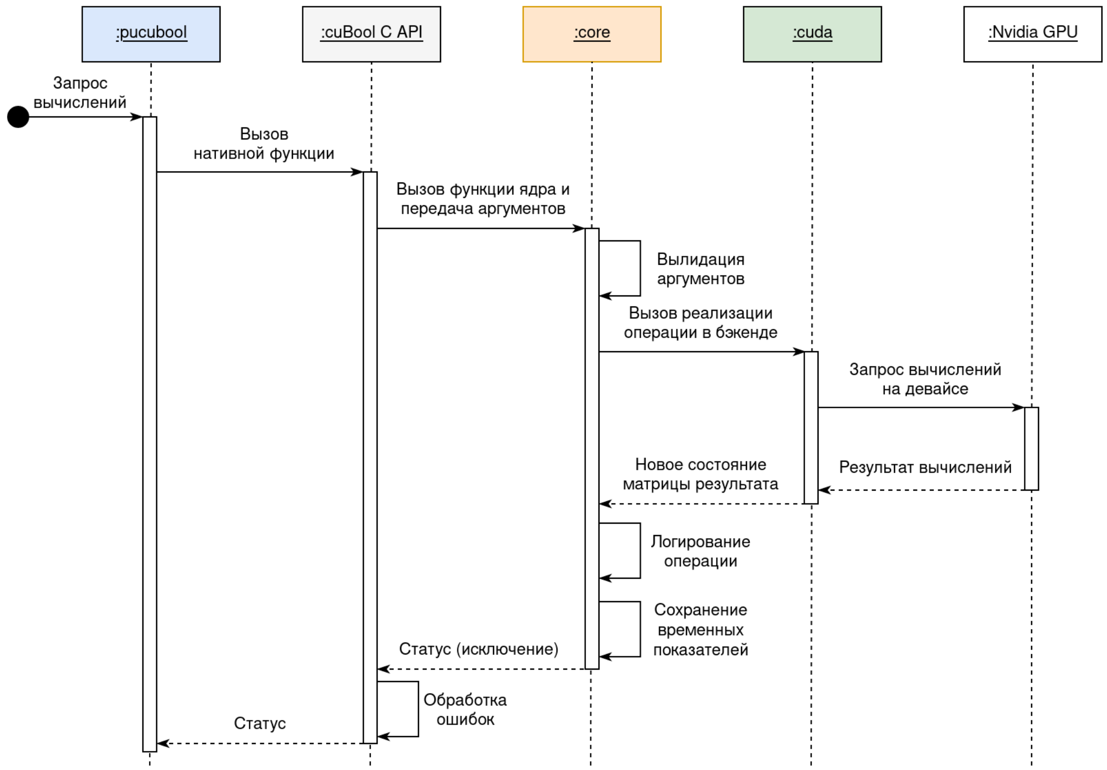

Рис. 2: Последовательность выполнения вычислительной матричной операции на Nvidia GPU с использованием pycubool

4. Детали реализации

Разработка библиотеки разреженной линейной булевой алгебры *cuBool* осуществлялась в рамках исследовательского проекта лаборатории языковых инструментов JetBrains Research.

В качестве языка программирования для реализации библиотеки используется C++, так как он предоставляет механизмы для ручного управления ресурсами, а также позволяет использовать CUDA C/C++ в рамках единого компилируемого приложения. Интерфейс библиотеки реализован в виде C-совместимого API. Исходный код компилируется в разделяемую библиотеку **libcubool.so**, которая может быть динамически загружена в конечное пользовательское приложение. В качестве целевой платформы для исполнения поддерживается семейство операционных систем на базе ядра Linux.

4.1. Примитивы и операции

Основным примитивом библиотеки является разреженная матрица булевых значений, которая хранится в видеопамяти видеокарты в формате *CSR* (compressed sparse row), который позволяет использовать $O(V + E)$ памяти для хранения матрицы смежности графа. Существуют и другие форматы хранения разреженных матриц: *CSC* (compressed sparse column), *COO* (coordinate list) и так далее. Однако CSR формат был выбран на основе результатов исследования Юсуке Нагасака и др. [21], так как он позволяет эффективно реализовать операцию матричного умножения в условиях ограниченного объема доступной видеопамяти.

В качестве поэлементных операций сложения и умножения используются *логическое-или* и *логическое-и*. Основные функции работы с матрицами представлены ниже:

- Создание матрицы M размера $m \times n$
- Удаление матрицы M и освобождение занятых ею ресурсов
- Заполнение матрицы M списком значений $\{(i, j)_k\}_k$
- Чтение из матрицы M списка значений $\{(i, j) \mid M[i, j] = 1\}$
- Транспонирование матрицы $M = N^T$
- Извлечение подматрицы $M = N[i..m, j..n]$
- Редуцирование матрицы к вектору $V = reduceToColumn(M)$
- Сложение матриц $C+ = M$
- Умножение матриц $C+ = M \times N$
- Произведение Кронекера для двух матриц $C = M \otimes N$

4.2. Cuda-модуль

Операции линейной алгебры для работы с матрицами реализованы на CUDA C/C++. В качестве основы для реализации операций умножения и сложения матриц используется библиотека **Nsparse**, представленная в исследовательской работе Арсения Терехова и др. [8]. Данная библиотека была доработана, чтобы добавить возможность динамически конфигурировать механизмы использования видеопамяти.

Для реализации произведения Кронекера, операций транспонирования, редуцирования и извлечения подматрицы использовались примитивы библиотеки **Thrust**. Данная библиотека позволяет оперировать данными в терминах высокоуровневых операций *свертки*, *отображения* и *префиксной суммы* [18], которые выполняются на графическом процессоре. **Thrust** поставляется совместно с интсрументами CUDA-разработки и не требует настройки дополнительных зависимостей.

4.3. Python-пакет

Все примитивы и операции библиотеки cuBool доступны внутри Python-пакета pycubool. Для публикации пакета используется стандартная инфраструктура PyPI. Вызов нативных методов из **cuBool C API**, находящихся в скомпилированной библиотеке **libcubool.so**, осуществляется с помощью модуля **Ctypes**. Данный модуль поставляется вместе с инфраструктурой Python и не требует настройки сторонних зависимостей. Также в pycubool добавлены дополнительные операции, которые облегчают использование данного пакета и предоставляют конечному пользователю дополнительную функциональность.

- Загрузка и сохранение матрицы в *Matrix market* формате.
- Экспортирование набора матриц в *GraphViz* формате.
- *Красивая* печать матриц в текстовом виде
- Текстовые маркеры и имена матриц для отладки

4.4. Пример использования

В качестве примера рассмотрим проблему вычисления *транзитивного замыкания* (англ. transitive closure) для некоторого ориентированного графа без меток $\mathcal{G} = \langle V, E \rangle$. Результатом вычисления транзитивного замыкания является новый граф $\mathcal{G}_{tc} = \langle V, E_{tc} \rangle$, для которого верно следующее: $e = (v, u) \in E_{tc} \iff \exists v \pi u$ в $\mathcal{G}$. Данную проблему можно решить в терминах линейной алгебры, если представить граф в виде матрицы смежности с булевыми значениями.

Listing 2 Пример вычисления транзитивного замыкания с использованием cuBool C API

```c
#include <cubool/cubool.h>

cuBool_Status TransitiveClosure(cuBool_Matrix A, cuBool_Matrix* T) {
    cuBool_Matrix_Duplicate(A, T);                   /* Копируем матрицу смежности A */

    cuBool_Index total = 0;
    cuBool_Index current;
    cuBool_Matrix_Nvals(*T, &current);               /* Количество ненулевых значений */

    while (current != total) {                       /* Пока результат меняется  */
        total = current;
        cuBool_MxM(*T, *T, *T, CUBOOL_HINT_ACCUMULATE);  /* T += T x T */
        cuBool_Matrix_Nvals(*T, &current);
    }

    return CUBOOL_STATUS_SUCCESS;
}
```

Listing 3 Пример вычисления транзитивного замыкания с использованием пакета pycubool

```python
import pycubool

def transitive_closure(a: pycubool.Matrix):
    t = a.duplicate()                       # Копируем матрицу смежности A
    total = 0                               # Количество ненулевых значений результата

    while total != t.nvals:                 # Пока результат меняется
        total = t.nvals
        t.mxm(t, out=t, accumulate=True)    # t += t x t

    return t
```

В листинге 2 представлен фрагмент кода на языке C, который решает данную задачу. В качестве аргументов функция принимает матрицу смежности исходного графа, а также указатель на идентификатор, который необходимо использовать при сохранении результирующей матрицы смежности графа после транзитивного замыкания.

В листинге 3 представлен похожий фрагмент кода, однако он уже решает поставленную в задачу на языке Python. Здесь в качестве входного аргумента используется матрица смежности графа, в качестве результата возвращается матрица смежности графа после транзитивного замыкания.

4.5. Алгоритм поиска путей с КС ограничениями

Алгоритм [7] поиска путей с КС ограничениями через тензорное произведение реализован с использованием разработнанного пакета pycubool. Его реализация доступна в рамках проекта PyAlgo-CFPQ [30]. Алгоритм встроен в существующую инфраструктуру для осуществления замеров производительности, а также для подключения к графовой базе данных RedisGraph [24] для загрузки данных, требуемых для экспериментов.

На вход алгоритм получает граф и КС грамматику. Граф представлен в виде булевой матричной декомпозиции матрицы смежности графа. КС грамматика закодирована в виде рекурсивного автомата. Его матрица переходов также представлена в булевой матричной декомпозиции На выходе алгоритм возвращает матрицу смежности графа достижимости, а также индекс, который позволяет восстанавливать все пути в графе в соответствии с входной грамматикой.

Также с использованием pycubool реализован классический матричный алгоритм Рустама Азимова [3], треьуемый для корректного сравнения производительности с алгоритмом на основе тензерного произведения.

5. Экспериментальное исследование

5.1. Постановка экспериментов

Для экспериментов использовалась рабочая станция с процессором Intel Core i7-6790, тактовой частотой 3.40GHz, RAM DDR4 с объемом памяти 64Gb, видеокартой GeForce GTX 1070 с 8Gb VRAM, ОС под управлением Ubuntu 20.04.

Данные, необходимые для замеров, предварительно загружаются в RAM или VRAM в формате, требуемом для тестируемого интсрумента. Время, необходимое на чтение данных с диска, их конвертацию, а также подготовку начального состояния входных матриц исключено из замеров.

Исследовательские вопросы

Для того, чтобы стуктурировать исследование, были сформулированы следующие вопросы.

В1: Какова производительность отдельных операций реализвонной библиотеки примитивов разреженной линейной булевой алгебры на GPGPU по сравнению с существующими аналогами?

В2: Какова производиетльность реализованного алгоритма поиска путей через тензорное произведение на GPGPU по сравнению с существующими аналагами, также полагающимися на примтивы линейной алгебры?

В1 направлен на определение эффективности отдельных матричных операций в реализованной библиотеке. В качестве таких операций выступают *матричное умножение* и *матрицное сложение* в булевом полукольце, как наиболее распространненные и критически важные операции в прикладных алгоритмах. Для сравнения производительности в этих операциях предлагается использовать популярные существующие библиотеки разреженной линейной алгебры для платформ Nvidia Cuda, OpenCL и CPU. В качетсве таких библиотек ьыли выбраны были выбраны CUSP и cuSPRASE для Nvidia Cuda, clSPARSE для OpenCL, и SuiteSparse для CPU. CUSP предаставляет реализацию операций, основанную на шаблонах для параметризации используемого типа данных, однако библиотека не делает каких-либо дополнительных оптимизация конкретно для булевых значений. cuSPARSE и clSPRASE предоставляют операции только для основных типов данных с плавающей запятой. Однако данное ограничение можно обойти, если интерпретировать ненулевые значения как *true*. Библиотека SuiteSprase является эталонной реализацией GraphBLAS API и имеет встроенное булево полукольцо для вычислений.

В2 направлен на определение эффективности реализованного алгоритма и его сравнение с алгоритмом Рустама Азимова [8], который также полагается на операции линейной булевой алгебры. Данный алгоритм также реализован с использованием разработанного в данной работе Python-пакета, что делает сравнение корректным.

Набор данных

Для замеров производительности отдельных операций реализованной библиотеки были выбраны 10 различных квадратных матриц из известной коллекции университета Флориды [26] для проверки эффективности алгоритмов, реализующих операции над разряженными матрицами. Информация о матрицах представлена в таблице 1. Для обозначения числа ненулевых элементов используется аббревиатура Nnz (англ. number of non-zero elements). В таблице приведено официальное название матрицы, количество строк (соответсвует числу столбцов), а также количество ненулевых элментов в данной матрице и в производных от нее, полученных умножением матрицы самой на себя, что обозначается как степень M^2, и поэлементным сложением данной матрицы с собой также возведенной в степень, что обозначается как $M + M^2$. Вычисление данных артефатков имитирует шаг транзитивного замыкания. Эффективное вычисление этого шага во многом определяет производительность конечных пользовательских алгоритмов на графах.

Таблица 1: Разреженные матричные данные

Матрица M	Кол-во Строк	Nnz M	Nnz M^2	Nnz $M + M^2$
wing	62,032	243,088	714,200	917,178
luxembourg_osm	114,599	239,332	393,261	632,185
amazon0312	400,727	3,200,400	14,390,544	14,968,909
amazon-2008	735,323	5,158,388	25,366,745	26,402,678
web-Google	916,428	5,105,039	29,710,164	30,811,855
roadNet-PA	1,090,920	3,083,796	7,238,920	9,931,528
roadNet-TX	1,393,383	3,843,320	8,903,897	12,264,987
belgium_osm	1,441,295	3,099,940	5,323,073	8,408,599
roadNet-CA	1,971,281	5,533,214	12,908,450	17,743,342
netherlands_osm	2,216,688	4,882,476	8,755,758	13,626,132

Для замеров производительности алгоритмов поиска путей с КС ограничениями используется коллекция графовых данных лаборатории языковых интсрументов JetBrains Research [14], которая использовались в ряде работ [3, 7, 8, 11] для подобных экспериментов. Данная колекция содержит RDF данные, онтологии, графы программ для анализа указателей, а также ряд сгенерированных графов для анализа особых случаем.

Метрики

Для ответа на поставленные исследовательские вопросы в качестве метрик производительности используется время, требуемое для выполнения операции, а также пиковое количество потребяемой видеопамяти на GPU в момент вычисления. Показатели времени усреднены по 10 запускам. Предварительно совершался не учитывающийся в замерах запуск, чтобы проинициализировать начальное состояние тестируемых библиотек. Показатели потребления видеопамяти получены с помощью интсрумента *nvidia-smi*, который с точностью до 1 миллисикунды позволяет отслеживать количество потребляемой памяти процессом ОС на стороне видеокарты.

5.2. Результаты

1) В1: Какова производительность отдельных операций реализвонной библиотеки примитивов разреженной линейной булевой алгебры на GPGPU по сравнению с существующими аналогами?

Результаты эксперимента по сравнению производительности матричного произведения представлены в таблице 2. Реализованная библиотека cuBool показывает лучшие результаты по сравнению с другими библиотеками. Используемы в реализации алгоритм Nsparse позволяет получить прирост в скорости до 5 раз, а также сократить потребление видеопамяти до 8 раз, что особенно заметно в сравнении с такими библиотеками как CUSP или clSPARSE.

Результаты эксперимента по сравнению производительности матричного поэлементного сложения представлены в таблице 3. Библиотека clSPRARSE не реализует данную операцию, поэтому относящаяся к ней колонка с результатами оставлена пустой. cuBool демонстрируют хорошую производительность, его показатели времени сравнимы с такими промышленными библиотеками как CUSP или cuSPRASE и отличаются незначительно как в большую, так и меньшую сторону. Однако используемая cuBool операция сложения потребляет значительно меньше видеопамяти во время обработки, что позволяет местами достигать до 3 раз меньших значений в сравнении с CUSP.

2) В2: Какова производиетльность реализованного алгоритма поиска путей через тензорное произведение на GPGPU по сравнению с существующими аналагами, также полагающимися на примтивы линейной алгебры?

Таблица 2: Матричное умножение (время (t) в миллисекундах, память (m) в мегабайтах)

Матрица	cuBool		CUSP		cuSPRS		clSPRS		SuiteSprs	
	t	m	t	m	t	m	t	m	t	m
wing	1.9	93	5.2	125	20.1	155	4.2	105	7.9	22
luxembourg_osm	2.4	91	3.7	111	1.7	151	6.9	97	3.1	169
amazon0312	23.2	165	108.5	897	412.8	301	52.2	459	257.6	283
amazon-2008	33.3	225	172.0	1409	184.8	407	77.4	701	369.5	319
web-Google	41.8	241	246.2	1717	4761.3	439	207.5	1085	673.3	318
roadNet-PA	18.1	157	42.1	481	37.5	247	56.6	283	66.6	294
roadNet-TX	22.6	167	53.1	581	46.7	271	70.4	329	80.7	328
belgium_osm	23.2	151	32.9	397	26.7	235	68.2	259	56.9	302
roadNet-CA	32.0	199	74.4	771	65.8	325	98.2	433	114.5	344
netherlands_osm	35.3	191	51.0	585	51.4	291	102.8	361	90.9	311

Таблица 3: Поэлементное матричное сложение (время (t) в миллисекундах, память (m) в мегабайтах)

Матрица	cuBool		CUSP		cuSPRS		clSPRS		SuiteSprs	
	t	m	t	m	t	m	t	m	t	m
wing	1.1	95	1.4	105	2.4	163	-	-	2.3	176
luxembourg_osm	1.7	95	1.0	97	0.8	151	-	-	1.6	174
amazon0312	11.4	221	16.2	455	24.3	405	-	-	37.2	297
amazon-2008	17.5	323	29.5	723	27.2	595	-	-	64.8	319
web-Google	24.8	355	31.9	815	89.0	659	-	-	77.2	318
roadNet-PA	16.9	189	11.2	329	11.6	317	-	-	36.6	287
roadNet-TX	19.6	209	14.5	385	16.9	357	-	-	45.3	319
belgium_osm	19.5	179	10.2	303	10.5	297	-	-	28.5	302
roadNet-CA	30.5	259	19.4	513	20.2	447	-	-	65.2	331
netherlands_osm	30.1	233	14.8	423	18.3	385	-	-	50.2	311

6. Заключение

В рамках выполнения данной работы были получены следующие результаты:

- Спроектирована библиотека примитивов линейной булевой алгебры для работы с разреженными данными на GPGPU. Данная библиотека экспортирует C-совместимый интерфейс, имеет поддержку различных вычислительных модулей, а также предоставляет Python-пакет для работы конечного пользователя с примитивами библиотеки в высокоуровневой среде вычислений с управляемыми ресурсами.

- Реализована библиотека cuBool в соответствии с разработанной архитектурой. Ядро библиотеки написано на языке C++, а математические операции, выполняющиеся на GPGPU, реализованы на языке CUDA C/C++. Библиотека предоставляет модуль CPU вычислений для компьютеров без Cuda девайсов. Также создан Python-пакет pycubool, который оступен для скачивания через пакетный менеджер PyPI. С использованием данного пакета реализован алгоритм поиска путей с КС ограничениями через тензорное произведение. Данный алгоритм использует операции матричного умножения, сложения и произведение Кронекера в булевом полукольце, а также различные операции для манипуляций над значениями матриц.

- Выполнено экспериментальное исследование полученных артефактов. Матричное умножение показывает ускорение до 5 раз по сравнению с существующими аналогами, матричное сложение сравнимо по времени с существующими аналогами, однако операции потребляет до 3 раз меньше видеопамяти. Реализованный алгоритм поиска путей с КС ограничениями показывает ускорение до 6 раз по сравнению с CPU версией, что делает его GPGPU-версию более применимой для реального анализа данных.

На основе результатов, полученных в данном иследовании, была написана статья, принятая на конференцию GrAPL 2021[1].

Библиотека cuBool и Python-пакет для работы с данной библиотекой доступны для скачивания через следующие онлайн ресурсы: https://github.com/JetBrains-Research/cuBool и https://test.pypi.org/project/pycubool/.

[1]GrAPL 2021: Workshop on Graphs, Architectures, Programming, and Learning. Дата обращения: 1.04.2021. Сайт конференции: https://hpc.pnl.gov/grapl/.

Список литературы

[1] Abiteboul Serge, Hull Richard, Vianu Victor. Foundations of Databases. — 1995. — 01. — ISBN: 0-201-53771-0.

[2] Analysis of Recursive State Machines / Rajeev Alur, Michael Benedikt, Kousha Etessami et al. // ACM Trans. Program. Lang. Syst. — 2005. — Jul. — Vol. 27, no. 4. — P. 786–818. — Access mode: https://doi.org/10.1145/1075382.1075387.

[3] Azimov Rustam, Grigorev Semyon. Context-free path querying by matrix multiplication. — 2018. — 06. — P. 1–10.

[4] Barceló Baeza Pablo. Querying Graph Databases // Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. — PODS '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 175–188. — Access mode: https://doi.org/10.1145/2463664.2465216.

[5] ClSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library / Joseph L. Greathouse, Kent Knox, Jakub Poła et al. // Proceedings of the 4th International Workshop on OpenCL. — IWOCL '16. — New York, NY, USA : Association for Computing Machinery, 2016. — Access mode: https://doi.org/10.1145/2909437.2909442.

[6] Context-Free Path Queries on RDF Graphs / Xiaowang Zhang, Zhiyong Feng, Xin Wang et al. // CoRR. — 2015. — Vol. abs/1506.00743. — 1506.00743.

[7] Context-Free Path Querying by Kronecker Product / Egor Orachev, Ilya Epelbaum, Rustam Azimov, Semyon Grigorev. — 2020. — 08. — P. 49–59. — ISBN: 978-3-030-54831-5.

[8] Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication / Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, Semyon Grigorev. — 2020. — 06. — P. 1–12.

[9] Dalton Steven, Bell Nathan, Olson Luke, Garland Michael. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. — 2014. — Version 0.5.0. Access mode: http://cusplibrary.github.io/.

[10] Direct3D 12 Graphics // Microsoft Online Documents. — 2018. — Access mode: https://docs.microsoft.com/ru-ru/windows/win32/direct3d12/direct3d-12-graphics?redirectedfrom=MSDN (online; accessed: 08.12.2020).

[11] Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication / Nikita Mishin, Iaroslav Sokolov, Egor Spirin et al. — 2019. — 06. — P. 1–5.

[12] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker // Proceedings of the 31st International Conference on Scientific and Statistical Database Management. — SSDBM '19. — New York, NY, USA : ACM, 2019. — P. 121–132. — Access mode: http://doi.acm.org/10.1145/3335783.3335791.

[13] Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis / Qirun Zhang, Michael R. Lyu, Hao Yuan, Zhendong Su // SIGPLAN Not. — 2013. — Jun. — Vol. 48, no. 6. — P. 435–446. — Access mode: https://doi.org/10.1145/2499370.2462159.

[14] Graphs and grammars for Context-Free Path Querying algorithms evaluation // Github. — 2021. — Access mode: https://github.com/JetBrains-Research/CFPQ_Data (online; accessed: 11.03.2021).

[15] Hellings Jelle. Path Results for Context-free Grammar Queries on Graphs. — 2015. — 02.

[16] Hopcroft John E., Motwani Rajeev, Ullman Jeffrey D. Introduction to Automata Theory, Languages, and Computation (3rd Edition). — USA : Addison-Wesley Longman Publishing Co., Inc., 2006. — ISBN: 0321455363.

[17] Medeiros Ciro, Musicante Martin, Costa Umberto. An Algorithm for Context-Free Path Queries over Graph Databases. — 2020. — 04.

[18] NVIDIA. CUDA Thrust // NVIDIA Developer Zone. — 2020. — Access mode: https://docs.nvidia.com/cuda/thrust/index.html (online; accessed: 16.12.2020).

[19] NVIDIA. CUDA Toolkit Documentation // NVIDIA Developer Zone. — 2020. — Access mode: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (online; accessed: 01.12.2020).

[20] NVIDIA. cuSPARSE reference guide // NVIDIA Developer Zone. — 2020. — Access mode: https://docs.nvidia.com/cuda/cusparse/index.html (online; accessed: 09.12.2020).

[21] Nagasaka Yusuke, Nukada Akira, Matsuoka Satoshi. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. — 2017. — 08. — P. 101–110.

[22] OpenCL: Open Standard for Parallel Programming of Heterogeneous Systems // Khronos website. — 2020. — Access mode: https://www.khronos.org/opencl/ (online; accessed: 08.12.2020).

[23] Quantifying variances in comparative RNA secondary structure prediction / James Anderson, Adám Novák, Zsuzsanna Sükösd et al. // BMC bioinformatics. — 2013. — 05. — Vol. 14. — P. 149.

[24] RedisGraph GraphBLAS Enabled Graph Database / P. Cailliau, T. Davis, V. Gadepally et al. // 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2019. — P. 285–286.

[25] Santos Fred, Costa Umberto, Musicante Martin. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. — 2018. — 01. — P. 225–233. — ISBN: 978-3-319-91661-3.

[26] T. Davis. The SuiteSparse Matrix Collection (the University of Florida Sparse Matrix Collection). — 2020. — Access mode: https://sparse.tamu.edu (online; accessed: 09.03.2021).

[27] The Khronos Working Group. OpenGL 4.4 Specification // Khronos Registry. — 2014. — Access mode: https://www.khronos.org/registry/OpenGL/specs/gl/glspec44.core.pdf (online; accessed: 08.12.2020).

[28] The Khronos Working Group. Vulkan 1.1 API Specification // Khronos Registry. — 2019. — Access mode: https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html (online; accessed: 08.12.2020).

[29] Yannakakis Mihalis. Graph-Theoretic Methods in Database Theory // Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. — PODS '90. — New York, NY, USA : Association for Computing Machinery, 1990. — P. 230–242. — Access mode: https://doi.org/10.1145/298514.298576.

[30] A collection of CFPQ algorithms implemented in PyGraphBLAS // Github. — 2020. — Access mode: https://github.com/JetBrains-Research/CFPQ_PyAlgo (online; accessed: 16.12.2020).