Оглавление

Введение

Графовая модель данных широко используется в разных сферах, так как способна устранить ограничения реляционной модели данных. В то время, как реляционная модель данных связывает данные неявными связями, графовая модель данных явно излагает зависимости между узлами. Графовая модель данных позволяет легко и быстро извлекать сложные иерархические структуры, которые трудно моделировать в реляционной модели данных. Особенно хорошо графовая модель данных зарекомендовала в таких сферах как: графовые базы данных [4], статический анализ [12], биоинформатика [21], RDF-графы [7].

В данных и других областях графовая модель используется с целью представления отношений между объектами. Имея такие отношения, часто возникает задача вычисления запросов над данными для выявления более сложных зависимостей между ними. Одним из способов описания отношений между объектами графа является накладывание ограничений на пути между вершинами. Для этого можно использовать формальные грамматики (регулярные, контекстно-свободные) над алфавитом меток ребер графа. Контекстно-свободные грамматики выразительнее чем регулярные, например, так называемые *same-generation queries* [1] не могут быть выражены через регулярные грамматики но могут быть выражены через контекстно-свободные грамматики.

Таким образом, выполнение контекстно-свободных запросов заключается в следующем: для заданной пользователем контекстно-свободной грамматики необходимо вернуть множество всех троек вида (A, m, n), для которых верно, что существует путь из вершины m в вершину n конкатенация меток на котором образует слово, выводимое из нетерминала A грамматики.

В то время, как регулярные запросы поддерживаются во многих графовых базах данных, возможность поддержки контекстно-свободных запросов изучена плохо. Существуют алгоритмы [3, 7, 16, 18] для выполнения контекстно-свободных запросов, но они не интегрированы ни с какими графовыми базами данных.

В работе [11] была проведена интеграция некоторых актуальных алгоритмов выполнения контекстно-свободных запросов в Neo4j [1], после чего исследователи пришли к выводу, что текущие алгоритмы не способны обрабатывать графы больших размеров, в силу длительного выполнения запросов.

Тем не менее, Никита Мишин и др. показали [10], что использование *GPGPU* для матричного алгоритма [3] может уменьшить время выполнения контекстно-свободных запросов. Однако, предложенные авторами реализации были не способны работать на реальных графах, в силу ограниченного объема видеопамяти.

Таким образом, матричный алгоритм выглядит перспективным алгоритмом выполнения контекстно-свободных запросов, который позволяет эффективно использовать многоядерные системы и *GPGPU*, что приводит нас к следующим целям и задачам.

Целью данной работы является реализация матричного алгоритма выполнения контекстно-свободных запросов для графовой базы данных с использованием *GPGPU*

Для достижения этой цели решаются следующие задачи:

- реализовать алгоритм с реляционной семантикой
- реализовать алгоритм с возможностью восстановления пути
- реализовать возможность работы алгоритма в условиях недостаточного объема видеопамяти
- провести экспериментальное исследование предложенных реализаций

[1] Neo4j – графовая база данных, страница проекта `https://neo4j.com`. Дата посещения: 20.05.2020

1 Обзор предметной области

1.1 Формальные языки

Введем базовые определения теории формальных языков, которые будут использоваться в дальнейшем.

Определение 1.1 *Алфавит (Σ) — конечное множество. Элементы данного множества - символы.*

Определение 1.2 *Слово (w) над алфавитом — конкатенация конечного числа символов.*

Определение 1.3 *Язык над алфавитом — множество всех слов на алфавитом.*

Определение 1.4 *Контекстно-свободная грамматика — четверка* (N, S, Σ, P)

- N *— конечное множество нетерминалов*
- S *— стартовый нетерминал из множества* N
- Σ *— конечное множество терминалов (алфавит)*
- P *— множество правил грамматики вида:*

$$A \to \gamma, \text{ где } A \in N, a\,\gamma \in (\Sigma \cup N)^*$$

Определение 1.5 *Контекстно-свободная грамматика находится в нормальной форме Хомского если каждое правило грамматики имеет одну из форм:*

- $S \to \varepsilon$
- $A \to BC$, *где* A, B, $C \in N$
- $A \to a$, *где* $A \in N$ *и* $a \in \Sigma$

Обозначение $A \xrightarrow{*} w$ будет говорить о том, что слово w может быть выведено из нетерминала A последовательным применением правил P.

Определение 1.6 *Язык контекстно-свободной грамматики* $G = (N, S, \Sigma, P)$ *будем обозначать как* $L(G)$ *и определим следующим образом:*

$L(G) = \{w \in \Sigma^* | S \xrightarrow{*} w\}$

1.2 Поиск путей с контекстно-свободными ограничениями

Рассмотрим ориентированный граф $D = (V, E)$, где V — множество вершин, E — множество ребер с метками. Множество всех меток образует алфавит Σ. Любой путь в данном графе будет образовывать слово над алфавитом Σ путем конкатенации меток на ребрах данного пути. Путь обозначим как π, слово как $l(\pi)$. Введем сокращение $n\pi m$ чтобы обозначить путь, который начался в вершине $n \in V$ и закончился в вершине $m \in V$. Для графа D и контекстно-свободной грамматики $G = (N, S, \Sigma, P)$ введем отношения $R_A \in V \times V$ для каждого нетерминала грамматики $A \in N$:

$$R_A = \{(n, m) \mid \exists\, n\pi m :\ l(\pi) \in L(G_A)\} \tag{1}$$

Тогда задача поиска путей с контекстно-свободными ограничениями заданными грамматикой G с реляционной семантикой определяется как задача вычисления всех отношений R_A (1) для каждого нетерминала грамматики G.

Кроме того, существует задача поиска путей с контекстно-свободными ограничениями с семантикой восстановления всех путей. В данной задаче, помимо вычисления отношения R_A (1) для всех нетерминалов грамматики G, требуется предоставить все пути $n\pi m$, такие, что $l(\pi) \in L(G_A)$.

Но задача с семантикой восстановления всех путей может может не решаться за конечное время, так как путей может существовать беско-

нечно много. Поэтому, дополнительно существуют две семантики:

- кратчайшего пути — в этом случае возвращается кратчайший путь
- одного пути — в этом случае возвращается один путь

1.3 Существующие решения

Задача выполнения контекстно-свободных запросов с реляционной семантикой была впервые сформулирована Яннакакисом [24]. После постановки задачи были попытки предоставить алгоритм для решения данной задачи [7, 16], но это были прежде всего теоретические работы, в которых предлагались алгоритмы, изучались их свойства.

Первая исследовательская работа по интеграции алгоритмов исполнения контекстно-свободных запросов с графовой базой данных для изучения применимости существующих алгоритмов была проведена Йохемом Куйперсом и др. [11]. В своей работе они исследовали три самых актуальных алгоритма выполнения контекстно-свободных запросов. Целью их работы было исследовать поведение данных алгоритмов на различных данных, в том числе на больших графах из реального мира. Так как подавляющее большинство предшествующих работ представляли собой теоретические работы с описанием алгоритмов и тестированием лишь на малых графах. В качестве хранилища графов авторы использовали графовую базу данных Neo4j.

Авторы выбрали три алгоритма:

- алгоритм основанный на LR разборе [22]
- алгоритм Хеллингса [16]
- матричный алгоритм [3]

Эксперименты проводились как на синтетически сгенерированных графах так и на реальных данных. Для синтетических графов использовалась модель генерирующая масштабно-инвариантные сети, предложенная Альберт-Ласло Барабаши [2]. Данная модель была выбрана,

так как хорошо описывает естественно возникающие сети из реального мира – социальные, биологические и др. В качестве реальных данных исследователи использовали онтологию Geospecies [20].

Для моделирования запросов к реальным данным использовались same-generation queries [1]. Количество вершин в сгенерированных графах достигало 10000, а количество ребер 100000. Количество вершин в онтологии Geospecies достигает 450000, полное количество ребер 2300000, но только 20000 из них имеют метки, обозначенные в запросе.

Результаты эксперимента показали, что время выполнения запросов на синтетических графах достигало приемлемых десятков секунд. В то время как на онтологии Geospecies время выполнения заняло 7000 секунд. Из чего исследователями сделан вывод о неприменимости существующих алгоритмов к обработке графов больших размеров существующими алгоритмами.

Однако, стоит обратить внимание, что выбранные исследователями алгоритмы были реализованы не достаточно эффективно, как минимум потому, что все алгоритмы были реализованы используя лишь одно ядро центрального процессора.

Другая исследовательская работа по экспериментальному изучению выполнения контекстно-свободных запросов была выполнена Никитой Мишиным и др. [10]. В своей работе авторы исследовали применимость высоко-производительных библиотек для увеличения производительности матричного алгоритма [3] выполнения запросов. Помимо высокопроизводительных библиотек, использующих центральный процессор, авторы воспользовались *GPGPU*.

На всех экспериментальных данных, реализации использующие *GPGPU* оказались быстрее, чем реализации использующие один лишь центральный процессор. Однако, исследуемые реализации все еще не были приспособлены к работе с большими графами, так как исследуемые реализации работали с матрицами смежности графов в плотном представлении и максимальный размер графа в их работе составлял 80000 вершин.

1.4 Матричный алгоритм выполнения запросов

Из существующих на сегодняшний день алгоритмов выполнения контекстно-свободных запросов необходимо выделить матричный алгоритм, как алгоритм, который хорошо подлежит распараллеливанию на многоядерных системах, так как использует матричные операции. Рассмотрим матричный алгоритм выполнения запросов с реляционной семантикой из работы Рустама Азимова [3].

Алгоритм с реляционной семантикой

Обозначим за $G = (N, \Sigma, P)$ контекстно-свободную грамматику, за $D = (V, E)$ граф. Определим бинарную операцию $(\cdot)$ для двух произвольных подмножеств нетерминалов грамматики G:

$$N_1 \cdot N_2 = \{A \mid \exists B \in N_1, \exists C \in N_2 \text{ такие, что } (A \to BC) \in P\} \qquad (2)$$

Используя операцию (2) в качестве операции умножения, а операцию объединения множеств в качестве сложения, мы можем определить матричное произведение $c = a \times b$ матриц, элементами которой являются подмножества нетерминалов:

$$c_{i,j} = \bigcup_{k=1}^{n} a_{i,k} \cdot b_{k,j} \qquad (3)$$

Теперь можно перейти непосредственно к алгоритму. Пронумеруем все вершины в графе D от 0 до |V| - 1. На первом шаге алгоритма инициализируется матрица T размера |V|×|V| с элементами, представляющими собой подмножества нетерминалов грамматики G:

$$T_{i,j} = \{A_k \mid ((i, x, j) \in E) \wedge ((A_k \to x) \in P)\} \qquad (4)$$

Следующий шаг — поиск транзитивного замыкания матрицы T, используя следующую операцию:

$$T_{new} = T_{old} \cup (T_{old} \times T_{old}) \qquad (5)$$

Операция (5) выполняется до тех пор, пока в матрице T происходят изменения. В результате получается матрица T, в каждой ячейке (i, j) которой находится подмножество нетерминалов таких, что для каждого из них существует путь из вершины i в вершину j такой, что конкатенация меток на этом пути будет выводима из данного нетерминала.

Однако, матрица T имеет достаточно сложную структуру, так как в ее элементах находятся подмножества. Сделаем переход от матрицы T в элементах которой находятся подмножества множества N из не более чем $|N|$ элементов к $|N|$ булевым матрицам $T^{A_m}, A_m \in N$, таким что

$$T_{i,j}^{A_m} = \begin{cases} true, \text{ если } \exists i \pi j : l(\pi) \in L(A_m) \\ false, \text{ в противном случае} \end{cases} \tag{6}$$

Заметим, что теперь каждому нетерминалу A_m грамматики ставится в соответствие булева матрица T^{A_m}. Теперь задача состоит не в отыскании одной матрицы T, а в отыскании $|N|$ матриц $T^{A_m}, A_m \in N$. Общая схема алгоритма приведена в листинге 1.

Критически важной частью в алгоритме 1 являются операции над булевыми матрицами — сложение и умножение в строке 14.

Асимптотическая сложность алгоритма оказывается следующей:

$$O(|V|^2|N|^3(BMM(|V|) + BMU(|V|))) \tag{7}$$

где BMM — алгоритм умножения булевых матриц, BMU — алгоритм объединения булевых матриц. На реальных данных алгоритму достаточно лишь небольшого количества матричных операций для сходимости, что делает данный алгоритм применимым.

Алгоритм с возможностью восстановления пути

Алгоритм 1 позволяет отвечать на вопрос о существовании пути, но он не дает возможности получить или восстановить сам путь. Рассмотрим модификацию предыдущего алгоритма (1) из другой работы [8], которая позволит восстановить путь, обладающий свойством, что де-

Algorithm 1 Алгоритм выполнения контекстно-свободных запросов с реляционной семантикой

1: **function** CFPQRelational($D = (V, E), G = (N, \Sigma, P)$)
2: $n \leftarrow |\mathrm{V}|$
3: $T \leftarrow \{T^{A_i} \mid A_i \in N, T^{A_i}$ is a matrix $n \times n$, $T^{A_i}_{k,l} \leftarrow$ `false`$\}$
4: **for all** $(i, x, j) \in E$, $A_k \mid A_k \to x \in P$ **do**
5: $T^{A_k}_{i,j} \leftarrow$ `true`
6: **end for**
7: **for all** $A_k \mid A_k \to \varepsilon \in P$ **do**
8: **for all** $i \in \{0, \ldots, n-1\}$ **do**
9: $T^{A_k}_{i,i} \leftarrow$ `true`
10: **end for**
11: **end for**
12: **while** any matrix in T is changing **do**
13: **for** $A_i \to A_j A_k \in P$ **do**
14: $T^{A_i} \leftarrow T^{A_i} + (T^{A_j} \times T^{A_k})$
15: **end for**
16: **end while**
17: **return** T
18: **end function**

рево разбора слова, образованного конкатенацией меток на этом пути, будет иметь наименьшую высоту.

Идея алгоритма останется прежней, но теперь будем рассматривать не булевы матрицы, а матрицы, элементами которой являются следующей структуры:

$$PathIndex = (left, right, middle, height) \tag{8}$$

Данная структура позволяет восстановить найденный путь:

- $left$ — описывает вершину начала пути
- $right$ — описывает вершину конца пути
- $middle$ — промежуточная вершина, использованная для конкатенации результирующего пути из двух путей, пути из $left$ в $middle$ и из $middle$ в $right$

- $height$ — высота дерева разбора слова, образованного конкатенацией меток на пути из $left$ в $right$

Если не существует пути π между вершинами i и j такого, что слово $l(\pi)$ было бы выводимо из нетерминала A_m, то в данную позицию в матрице поместим специальное значение, которое обозначим за $\perp$: $T_{i,j}^{A_m} = \perp$, иначе $T_{i,j}^{A_m} = (i, j, middle, height)$.

Определим бинарную операцию $\otimes$ для структуры $PathIndex$ при условии, что $PI_1.rigth = PI_2.left$, следующим образом:

$$PI_1 \otimes PI_2 = \begin{cases} \perp, \text{ если } PI_1 = \perp \text{ или } PI_2 = \perp \\ (PI_1.left, PI_2.right, PI_1.rigth, max(PI_1.height, PI_2.height) + 1) \end{cases} \tag{9}$$

Определим бинарную операцию $\oplus$ для структуры $PathIndex$ при условии, что $PI_1.rigth = PI_2.rigth$ и $PI_1.left = PI_2.left$, следующим образом:

$$PI_1 \oplus PI_2 = \begin{cases} PI_1, \text{ если } PI_2 = \perp \\ PI_2, \text{ если } PI_1 = \perp \\ PI_1, \text{ если } PI_1.height < PI_2.height \text{ иначе } PI_2 \end{cases} \tag{10}$$

Используя операцию $\otimes$ в качестве операции умножения, а операцию $\oplus$ в качестве операции сложения, определим операцию умножения матриц $c = a \odot b$, состоящих из элементов $PathIndex$:

$$c_{i,j} = \bigoplus_{k=1}^{n} a_{i,k} \otimes b_{k,j} \tag{11}$$

Алгоритм для построения индекса приведен в листинге 2

Далее, необходимо уметь восстанавливать путь из построенного индекса. Алгоритм приведенный в листинге 3 выполняет восстановление рекурсивным образом для заданной на вход пары вершин.

Таким образом, можно заметить что структурно алгоритмы 1 и 2 похожи, за исключением того, что они выполняют матричные операции над разными полукольцами.

Algorithm 2 Алгоритм для построения индекса необходимого для восстановления пути

1: **function** CFPQPathIndex($D = (V, E), G = (N, \Sigma, P)$)
2: $n \leftarrow |\mathrm{V}|$
3: $T \leftarrow \{T^{A_i} \mid A_i \in N, T^{A_i}$ is a matrix $n \times n$, $T^{A_i}_{k,l} \leftarrow \perp \}$
4: **for all** $(i, x, j) \in E$, $A_k \mid A_k \to x \in P$ **do**
5: $T^{A_k}_{i,j} \leftarrow (i, j, i, 1)$
6: **end for**
7: **for** $A_k \mid A_k \to \varepsilon \in P$ **do** $T^{A_k}_{i,i} \leftarrow (i, i, i, 1)$
8: **end for**
9: **while** any matrix in T is changing **do**
10: **for** $A_i \to A_j A_k \in P$ **do**
11: $T^{A_i} \leftarrow T^{A_i} + (T^{A_j} \odot T^{A_k})$
12: **end for**
13: **end while**
14: **return** T
15: **end function**

1.5 Архитектура CUDA

На сегодняшний день есть два инструмента для GPGPU[2] разработки. Стандарт OpenCL[3] и CUDA[4]. В потребительском и высокопроизводительном сегменте наиболее популярны видеокарты от Nvidia, которые плохо поддерживают стандарт OpenCL, либо поддерживают лишь его старые версии, что приводит нас к выбору CUDA как инструмента для GPGPU разработки.

Физическая архитектура

CUDA совместимые GPU состоят из одного или нескольких *стриминговых мультипроцессоров (SM)*. Каждый *SM* имеет набор *стриминговых процессоров (SP)*, также навязываемых CUDA ядрами. CUDA ядра группируются вместе для выполнения инструкций, что в терминах NVIDA называется *warp*. *Warp* означает группу *потоков*, которые

[2]Неспециализированные вычисления на графических процессорах

[3]Открытый стандарт для параллельного программирования, страница проекта: `https://www.khronos.org/opencl/`. Дата посещения: 20.05.2020

[4]Платформа для парралельной разработки от Nvidia, страница проекта: `https://www.geforce.com/hardware/technology/cuda`. Дата посещения: 20.05.2020

Algorithm 3 Алгоритм восстановления пути

```
1: function extractPath(i, j, A, T = {T^{A_i}}, G = (N, Σ, P))
2:     index ← T^A_{i,j}
3:     if index = ⊥ then
4:         return π_∅                          ▷ Such a path does not exist
5:     end if
6:     if index.height = 1 then
7:         if index.length = 0 then
8:             return []                       ▷ Return an empty path
9:         end if
10:        for all x | (i, x, j) ∈ E do
11:            if A → x ∈ P then
12:                return [(i, x, j)]          ▷ Return a path of length one
13:            end if
14:        end for
15:    end if
16:    for all A → BC ∈ P do
17:        index_B ← T^B_{i,index.middle}
18:        index_C ← T^C_{index.middle,j}
19:        if (index_B ≠ ⊥) ∧ (index_C ≠ ⊥) then
20:            maxH ← max(index_B.height, index_C.height)
21:            if index.height = maxH + 1 then
22:                π_1 ← extractPath(i, index.middle, B, T, G)
23:                π_2 ← extractPath(index.middle, j, C, T, G)
24:                return π_1 + π_2            ▷ Return the concatenation of paths
25:            end if
26:        end if
27:    end for
28: end function
```

запланированы вместе, чтобы исполнить одинаковую инструкцию в текущий цикл. На сегодняшний день все GPU используют одинаковый размер *warp — 32*. Каждый *SM* имеет как минимум один планировщик *warp*'ов, который отвечает за то, чтобы *32* пользовательским *потокам* предоставить CUDA ядра для исполнения.

Виртуальная архитектура

При разработке на CUDA помимо понятия *потока*, существует понятие *блок потоков* и *сетка блоков*. При запуске функции, которая должна выполниться на GPU, разработчик указывает сколько *потоков* будет в одном *блоке потоков* и сколько будет всего *блоков потоков*, тем самым определяя *сетку блоков*.

При запуске каждый *блок* назначается на конкретный *SM*. Это позволяет *потокам* внутри одного *блока* использовать *shared memory*, которая физически находится на текущем *SM*. Если *блок потоков* не использует все ресурсы текущего *SM*, тогда другие *блоки* тоже могут быть назначены на текущий *SM*. Если ресурсы всех *SM* заняты, то не назначенные *блоки* будут ожидать освобождения ресурсов.

Потоки одного *блока* будучи назначенным на *SM*, разделяются на *warp*'ы, и в дальнейшем выполняются на CUDA ядрах. Так как весь контекст выполнения (регистры каждого потока, указатель на текущую инструкцию и др.) *warp*'а хранится в блоке памяти на текущем *SM* то отсутствуют накладные расходы на переключение контекста между *warp*'ами. Это позволяет скрывать задержки, порождаемые, например, подсистемой памяти: если текущий исполняемый *warp* ожидает результата чтения из глобальной памяти, то планировщик *warp*'ов снимет с исполнения на CUDA ядрах текущий *warp* и поставит на исполнение другой.

Работа с подсистемой памяти является самой сложной при разработке на *GPGPU*. Именно она может оказаться узким горлышком при разработке высокопроизводительного ПО. Поэтому необходимо отдавать себе отчет в том, какой тип памяти используется, а так-же каким образом происходит обращение к памяти из разных потоков.

Иерархия памяти

Остановимся лишь на трех важных типах памяти.

- регистры — похожи на таковые у процессора. Каждый поток работает со своим набором регистром, регистры хранятся в блоке

памяти фиксированного размера на *SM*. Время доступа – порядка одного цикла.

- *shared memory* — память доступная всем потокам одного блока. Располагается в блоке памяти фиксированного размера на *SM*. Время доступа – порядка десятка циклов.

- *global memory* (видеопамять) — память доступная всем потокам из разных блоков. Самая медленная память, доступ к ней осуществляется транзакциями по *32*, *64* или *128* байт. Поэтому выгодно, чтобы потоки из одного *warp*'а обращаясь к данной памяти, обращались бы к последовательным элементам, чтобы уменьшить количество необходимых транзакций. Время доступа – порядка тысячи циклов.

Регистры и *shared memory* физически располагаются на *SM*, разделяя блок памяти размером порядка *64* килобайт, т.е. имеют весьма ограниченный ресурс памяти. В то время как *global memory* физически располагается вне *SM*, ее размер достигает порядка десятков гигабайт.

2 Реализация матричного алгоритма выполнения запросов

2.1 Архитектура решения

В качестве графового хранилища использовался RedisGraph[5]. Все внутренние операции над графами RedisGraph делегирует высокопроизводительной реализации GraphBLAS[6]. RedisGraph — первая графовая база данных, которая переводит запросы на языке Cypher[7] к операциям линейной алгебры, которые впоследствии применяются к матрицам смежности хранящихся графов. Благодаря удачно подобранному программному интерфейсу GraphBLAS [17] удается возможным выразить необходимые операции над графами, а так-же реализовать их эффективно.

Для возможности интеграции различных реализаций алгоритма в базу данных был выбран следующий программный интерфейс:

```
1 int context_free_path_quering(const Grammar *grammar, CfpqResponse *response,
2         const GrB_Matrix *relations, const char **relations_names,
3         size_t relations_count, size_t graph_size);
```

Реализуемый алгоритм получает на вход объекты библиотеки GraphBLAS, с которыми удобно работать, так как они описывают графы в терминах разреженных матриц смежности.

Абстрагирование данного интерфейса от внутренних структур RedisGraph позволило проводить тестирование реализованных алгоритмов на корректность с использованием библиотеки Googletest[8] без необходимости разворачивания экземпляра базы данных.

[5]RedisGraph – графовая база данных, страница проекта: `https://oss.redislabs.com/redisgraph/`. Дата посещения: 20.05.2020

[6]GraphBLAS – примитивы линейной алгебры для написания алгоритмов над графами, страница проекта: `http://graphblas.org/`. Дата посещения: 20.05.2020

[7]Cypher – язык описания запросов к данным в графовой модели, страница проекта: `https://www.opencypher.org/`. Дата посещения: 20.05.2020

[8]Googletest – фреймворк для тестирования, страница проекта: `https://github.com/google/googletest`. Дата посещения: 20.05.2020

2.2 Реализация алгоритма с реляционной семантикой

В матричном алгоритме выполнения запросов с реляционной семантикой (1) критической частью с точки зрения производительности являются операции в строке 14. Нет возможности хранить матрицы в плотном виде, так как для этого потребовалось бы $O(V^2)$ памяти и граф из 500000 вершин занимал бы уже 30ГБ видеопамяти, что превышает доступные, как правило, на практике объемы. С другой стороны, реальные данные разрежены, поэтому единственным возможным решением является переход к разреженным матрицам и выполнению операций сложения и умножения именно над разреженными матрицами.

Разреженные матрицы как правило используют $O(E + V)$ памяти для хранения. Любые способы представить разряженную матрицу используя $O(E + V)$ памяти вводят косвенную адресацию. Самыми распространенными способами хранения разреженных матриц являются *CSR (compressed sparse row)* и *COO (coordinate list)*, так как позволяют эффективно обходить значения в одной строке матрицы (но не колонке), так как они хранятся в памяти линейно.

Рассмотрим базовый алгоритм перемножения двух разреженных матриц, использующий возможность эффективного обращения к последовательным элементам одной строки, из работы Фреда Гаставсона [15]. Данную схему использует большинство высокопроизводительных библиотек.

Существует три ключевых вопроса в реализации алгоритма (4):

- как распределить нагрузку между вычислительными блоками, в алгоритме есть 3 места для введения параллелизма (строки 2,3,4 соответственно)
- как определить конечную структуру результирующей матрицы и сколько памяти необходимо выделить под её хранение
- какую структуру данных использовать для аккумулирования результатов операций в строках 7 и 9

Algorithm 4 Алгоритм для нахождения результата $C = A \cdot B$

```
1: C_{i,j} ← ⊥ for all i, j
2: for all A_{i*} in A do
3:     for all nonzero A_{ij} in Ai* do
4:         for all nonzero B_{jk} in Bj* do
5:             v_{ijk} ← A_{ij} B_{jk}
6:             if C_{ik} = ⊥ then
7:                 C_{ik} ← v_{ijk}
8:             else
9:                 C_{ik} ← C_{ik} + v_{ijk}
10:            end if
11:        end for
12:    end for
13: end for
```

На сегодняшний день существуют лишь две полноценные библиотеки для работы с разреженными матрицами на CUDA, это CUSP и Cusparse. Их функциональность не ограничивается одним лишь произведением разреженных матриц и сложением. Кроме того, существуют работы, в которых вводятся новые и оптимизируются старые подходы к задаче перемножения разреженных матриц. Однако, во всех таких работах рассматривается произведение матриц с одинарной и двойной точностью, т.е. задачи, когда элементами являются *4* и *8* байтные числа (*float, int, double*). Исходный код если и прикладывается к таким работам, то переиспользовать его практически невозможно, так как решается несколько иная задача, чем перемножение булевых матриц.

Далее будет рассказано о трех подходах к реализации данной критически важной части алгоритма выполнения запросов. Две из них используют существующее библиотеки, а последний подход — предложенный мной алгоритм на основе одной из недавно опубликованных работ [19].

Реализация на основе библиотеки CUSP

Одна из реализаций использует библиотеку CUSP[9]. Для решения поставленной задачи достаточно наличия булевых матриц в библиотеке и реализованных операций над ними. CUSP предлагает более богатый интерфейс, так как библиотека использует шаблоны C++, то имеется возможность параметризовать тип данных, который находится в матрицах, так и операции "умножить" и "плюс". CUSP работает с матрицами в *COO* представлении, которое заключается в том, что каждое значение в матрице кодируется тройкой чисел — (колонка, столбец, значение). Но для булевых матриц напрашивается оптимизация следующего характера: домен значений состоит только из двух элементов, а значит *COO* представление можно сжать дополнительно на треть — храня лишь позиции элементов со значением *true*. Но в CUSP отсутствуют специализации алгоритмов и контейнеров для типа *bool*, которые бы осуществляли данную оптимизацию.

Для матричного произведения CUSP использует ESC алгоритм [5], который состоит из трёх фаз: Expansion (расширение), Sorting (сортировка), Compression (сжатие). На фазе расширения все промежуточные результаты, полученные в результате выполнения строки 5 алгоритма 4 сохраняются в отдельный массив. На втором шаге они должным образом сортируются и на последнем этапе аккумулируются результаты операций для одних и тех же позиций (i,j) в конечной матрице.

Суммирование матриц выполняется используя схожий алгоритм: элементы двух матриц объединяются в один массив, затем сортируются, и на последнем шаге аккумулируются результаты для одних и тех же позиций (i,j) в конечной матрице.

Как можно заметить, данный алгоритм требователен к количеству доступной памяти, так как разворачивает все произведения в новый массив. Так же, реализация из библиотеки CUSP не дает возможности уменьшить на треть использования памяти за счет отказа от массива значений для булевых матриц. Две данных особенности могут привести

[9]CUSP – шаблонная C++ библиотека для линейной алгебры, страница проекта: `https://cusplibrary.github.io/`. Дата посещения: 20.05.2020

к тому, что реализация матричного алгоритма на основе CUSP может испытывать проблемы с нехваткой памяти при росте количества элементов в матрицах.

Реализация на основе библитеки Cusparse

Другая реализация матричного алгоритма использует библиотеку Cusparse[10]. В Cusparse отсутсвуют методы для работы с булевыми матрицами, поддержана работа только с ограниченным набором типов данных: $float$, $double$, $cuComplex$, $cuDoubleComplex$. Но Cusparse использует двухфазный алгоритм для вычисления произведения разреженных матриц. На первой фазе вычисляется структура конечной матрицы, на второй фазе происходит заполнение матрицы конечными значениями. Программный интерфейс Cusparse разработан таким образом, что имеется возможность выполнить только первую фазу, а вторую фазу не выполнять. Не трудно заметить, что первая фаза решает проблему произведения булевых матриц.

Cusparse имеет закрытый исходный код, но удалось выяснить [9], что для выполнения первой фазы по нахождению конечной структуры используется идея хеширования. Для этого, для каждой строки результирующей матрицы заводится хеш-таблица, в которую добавляются элементы, полученные в результате строки 5 алгоритма 4. Использование идеи хеширования позволяет сразу агрегировать результаты и потенциально использовать меньшее количество памяти. С другой стороны, хеш-таблица может заполнится и ее придется расширять, что приведет к дополнительным накладным расходам. В случае больших размеров хеш-таблицы будут хранится в глобальной памяти, что может привести к потерям производительности в связи с тем, что будут выполнятся произвольные доступы к данным хеш-таблицы. Архитектура современных GPU предполагает, что доступ к глобальной памяти в пределах группы потоков будет последовательной, благодаря чему будет выполнено меньшее количество транзакций к глобальной памяти.

[10]Cusparse – библиотека от NVIDIA для работы с разреженными матрицами, страница проекта: `https://docs.nvidia.com/cuda/cusparse/index.html`. Дата посещения: 20.05.2020

Суммирование матриц выполняется используя ту же идею хеширования.

Во время выполнения матричного алгоритма, количество элементов в матрицах будет только возрастать, что может привести к образованию строк, в которых находится большое количество элементов, а как следствие к проблемам с производительностью при обращении к хеш-таблицам в глобальной памяти.

Реализация на основе алгоритма Nsparse

Для решения потенциальных проблем, которые могут возникнуть при использовании реализаций на основе CUSP и Cusparse, мной был разработан алгоритм, использующий идеи из работы [19]. Предложенная авторами Nsparse реализация алгоритма перемножения разреженных матриц показывала лучшие результаты относительно существующих алгоритмов, как с точки зрения памяти, так с точки зрения производительности. На Рис. 1 обозначены 6 основных шагов алгоритма. Ниже будет представлено описание каждого отдельного шага.

Рис. 1: Основные шаги алгоритма Nsparse

Отметим два пункта, которые были усовершенствованы в предложенном авторами алгоритме:

- предложенная авторами реализация решает задачу перемножения разреженных матриц с типами $float$ и $double$, в то время как для нашей задачи необходимы булевы матрицы
- предложенная авторами реализация допускает размещение хеш - таблиц в глобальной памяти и последующий неэффективный доступ к ним

Перейдём к непосредственному описанию конечной версии алгоритма, которая была реализована. Алгоритм решает задачу нахождения произведения двух разреженных булевых матриц: $C = A \times B$. Матрицы представлены в CSR формате, заметим, что массив значений можно не рассматривать в силу того, что матрицы булевы. Тогда, каждая матрица представляется двумя массивами:

- массив col — массив индексов столбцов, имеет размер равный количеству элементов в матрице
- массив rpt — массив индексации строк, имеет размер $n + 1$, где n есть число строк в матрице; элементы i-*ой* строчки находятся в массиве col на позициях с $rpt[i]$ до $rpt[i + 1]$

На *первом* шаге алгоритма для каждой строки результирующей матрицы C рассчитывается число промежуточных произведений используя алгоритм 5.

На *втором* шаге алгоритма строки разбиваются на группы в зависимости от количества промежуточных произведений для данной строки. Это нужно для того, чтобы более равномерно распределить вычислительные ресурсы для разных строк, в зависимости от того, сколько вычислений они порождают. Т.е. если строка порождает малое количество произведений, то ей необходимо малое количество shared memory для построения хеш-таблицы и меньшее число потоков.

Algorithm 5 Алгоритм для подсчета количества промежуточных произведений в iой строке результирующей матрицы

1: **function** IntermediateProductsCount(i, A, B)
2: $n_{prod} \leftarrow 0$
3: **for** $j = rpt_A[i]\ to\ rpt_A[i+1]$ **do**
4: $n_{prod} \leftarrow n_{prod} + (rpt_B[col_A[j]+1] - rpt_B[col_A[j]])$
5: **end for**
6: **return** n_{prod}
7: **end function**

На *третьем* шаге для каждой строки строится хеш-таблица в быстрой shared memory. Так как нам известно полное количество промежуточных произведений для каждой строки, то можно предоставить хеш-таблицу с ёмкостью не менее чем это число, для того, чтобы все элементы уместились. В хеш-таблицу помещаются индексы колонок, которые должны присутствовать в результирующей матрице. В конечном итоге для каждой строки будет известно, сколько будет элементов в данной строке (это число как правило сильно меньше чем количество промежуточных произведений, так как во всех произведениях некоторые колонки повторяются). В качестве алгоритма хеширования используется лок-фри алгоритм на линейных пробах (листинг 6). Однако, может существовать набор строк, хеш-таблицы для которых не уместятся в shared memory. В этом случае авторы алгоритма размещают хеш-таблицы в глобальной памяти, что приводит к последующему неэффективному доступу к не последовательным элементам хеш-таблиц из разных потоков. Вместо этого, будет использован подход ESC [5] алгоритма, который заключается в том, что все промежуточное произведения сохраняются в массив в глобальной памяти, а затем сортируются.

На *четвертом* шаге становится доступна информация о количестве элементов в результирующей матрице из *третьего* шага. Используя эту информацию выделяется память под конечную матрицу.

На *пятом* шаге, имея информацию о количестве элементов для каждой строчки из результирующей матрицы (из шага *три*), строки вновь разбиваются на группы исходя из количества элементов в результиру-

Algorithm 6 Алгоритм вставки элемента в хеш-таблицу

```
1: function InsertIntoHashTable(key, table, table_size)
2:     hash ← (key · SCALE) % table_size
3:     while true do
4:         if table[hash] = key then
5:             return false
6:         else if table[hash] = ⊥ then
7:             old ← atomicCAS(table + hash, ⊥, key)
8:             if old = ⊥ then
9:                 return true
10:            end if
11:        else
12:            hash ← (hash + 1) % table_size
13:        end if
14:    end while
15: end function
```

щей матрице с той-же целью, что и в шаге номер *два*.

На *шестом* шаге для каждой строки вновь строится хеш-таблица, как на шаге *три*. Но теперь элементы из хеш-таблицы переносятся в выделенную глобальную память на шаге *четыре*. Предварительно они подвергаются сортировке. Для этого используется алгоритм битонической сортировки, так как он не требует дополнительной памяти, ресурс которой ограничен, так как сортируемые массив находится в shared memory.

Реализацию суммирования (объединения) булевых матриц была реализована на основе алгоритма MergePath [14]. Было решено отказаться от реализации объединения через алгоритм хеширования, так как это приводило бы к еще большим промежуточным размерам хеш-таблиц, которые могли бы не поместиться в shared memory. Алгоритм MergePath применим, так как индексы колонок в пределах строк отсортированы по возрастанию. Итоговый алгоритм объединения матриц $C = A + B$ состоит из двух фаз. На первой фазе происходит расчет количества элементов для каждой строки матрицы C, конечное количество элементов в строке матрицы C не равно сумме элементов в соответствующих строках матрицы A и B, так как индексы колонок могут повторят-

ся. На второй фазе, после выделения необходимого количества памяти для элементов, происходит заполнение результирующей строки результатом объединения.

2.3 Реализация алгоритма с возможностью восстановления пути

Матричный алгоритм выполнения запросов с возможностью восстановления пути состоит из двух фаз: фазы построения индекса (2) и фазы восстановления пути (3). Алгоритм восстановления пути (3) плохо подходит для реализации используя GPU, так как во время восстановления пути для конкретной пары вершин, будет происходить доступ к произвольным элементам памяти, в рамках всех матриц. Соответственно, если рассмотреть какой-либо *warp*, то все потоки в пределах данного *warp*'а будут обращаться к произвольным участкам памяти, тем самым создавая большое число транзакций. Так-же, невозможно использовать *shared memory* для кеширования каких либо данных, чтобы уменьшить количество обращений к глобальной памяти.

Однако, можно реализовать с использованием GPU алгоритм построения индекса (2), так как задача схожа с таковой для алгоритма с реляционной семантикой. Но теперь матрица состоит не из булевых значений, а из элементов *PathIndex* (8), для которых определены бинарные операции умножения (9) и сложения (10).

Для решения данной задачи необходима библиотека для работы с разреженными матрицами и возможностью параметризовать полукольцо. В результате исследования, были выявлены три библиотеки, которые позволяют решить данную задачу: CUSP, GBTL-CUDA [13], GraphBLAST [23].

GBTL-CUDA [13] использует для матричных операций CUSP, поэтому не имеет смысла рассматривать GBTL-CUDA, так как она делегирует выполнение алгоритмов CUSP. Библиотека GraphBLAST [23] находится в активной стадии разработки и там всё еще нет средств для решения поставленной задачи.

В итоге было предложено две реализации, одна использует библиотеку CUSP, а вторая — реализация, которая использует результаты алгоритма с реляционной семантикой.

Реализация на основе библиотеки CUSP

Определив структуру $PathIndex(8)$ в виде структуры на языке программирования C++ , а так-же операции "плюс" (10) и "умножить" (9) в виде функторов и инстанциировав данными типами шаблонный алгоритм произведения матриц из CUSP, удается возможным получить алгоритм для построения индекса.

Реализация основанная на использовании существующего результата алгоритма с реляционной семантикой

Заметим, что конечная структура матриц, необходимых для восстановления пути, будет совпадать с таковой у конечных матриц для решения задачи с реляционной семантикой. Имея информацию о конечной структуре матриц, создадим матрицы для построения индекса восстановления пути. Например, если для какого-то нетерминала A булева матрица в CSR представлении выглядела так:

$$rpt = [0, 2, 3, 6, 8]$$
$$col = [1, 3, 0, 0, 1, 3, 1, 2]$$

То матрица для восстановления пути для соответствующего нетерминала будет выглядеть так:

$$rpt = [0, 2, 3, 6, 8]$$
$$col = [1, 3, 0, 0, 1, 3, 1, 2]$$
$$val = [\bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot]$$

Таким образом инициализируются матрицы для каждого нетерминала. Важно отметить, что если нам известно число элементов и распределение их по строкам в матрице, полученной для каждого нетерминала в

результате выполнения алгоритма с реляционной семантикой, то в матрице для соответствующего нетерминала, которую мы будем заполнять при построении индекса для восстановления пути, количество элементов и распределение по строкам останется неизменным.

В алгоритме 4 в строках 7 и 9 происходит обновление результата конечной матрицы. В нашем случае — элементы матрицы это объекты $PathIndex$, а значит, обновление не может быть выполнено атомарно, так как не существует атомарных операций над 16 байтами. Но если мы хотим максимально распараллелить данный алгоритм, то нужно допускать ситуацию, что один и тот-же элемент матрицы может обновляться из нескольких потоков. Существует два решения данной проблемы: использование примитивов синхронизации или использование дополнительной памяти для агрегации результата. Оказывается, можно реализовать операции в строках 7 и 9 атомарно. Для этого рассмотрим операцию "плюс" (10) для структуры $PathIndex$. Заметим, что она определена только для таких PI_1 и PI_2, у которых $PI_1.rigth = PI_2.rigth$ и $PI_1.left = PI_2.left$. Как следствие, $left$ и $rigth$ в результате остаются такими-же. Для любого элемента $PathIndex$, который не равен $\perp$ и находится на позиции (i, j) в матрице, справедливо: $PathIndex_{(i,j)} = (i, j, middle, height)$. Значит, индексы (i, j) можно вынести из структуры $PathIndex$. Но они всё еще необходимы для выполнения операции "умножить" (9). Чтобы решить эту проблему, мой алгоритм будет передавать данной бинарной операции контекст – индекс j левого операнда или индекс i правого (так как они равны).

Теперь структура $PathIndex$ состоит лишь из двух 32 битных полей — $middle$ и $height$. Чтобы реализовать атомарную операцию "плюс" (10) достаточно упаковать два данных поля в одно 64 битное число. В старших битах которого будет находится $height$, во младших $middle$ и использовать операцию $atomicMin$. В качестве значения для $\perp$ можно использовать максимальное 64 битное число.

Умея атомарно обновлять значения в матрице C, при вычислении $C = A \cdot B$, мы реализовали операцию $C = C + A \cdot B$, так как матрица C имеет известную структуру. Рассмотрим алгоритм (7), который позво-

ляет понять как именно происходит распараллеливание операций при обновлении одной строки матрицы C.

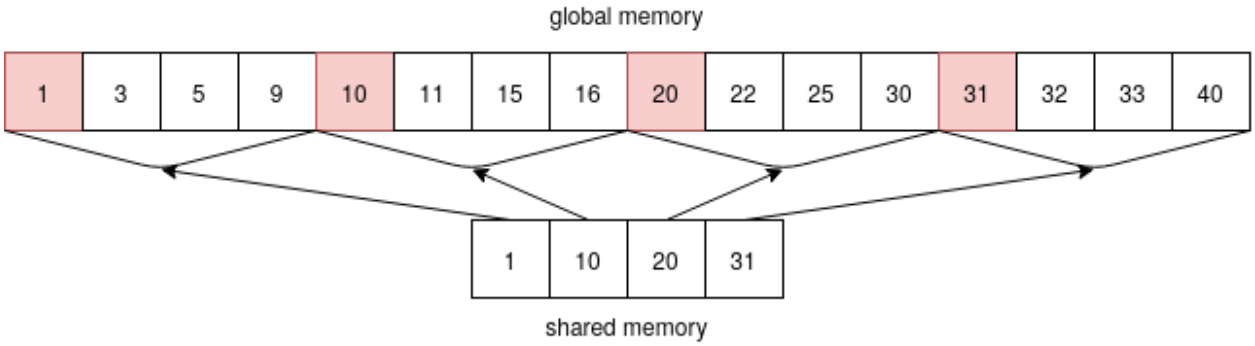
Algorithm 7 Заполнение результата для iой строки матрицы C

1: $i \leftarrow blockIdx$
2: $warp_{id} \leftarrow threadIdx / warpSize$
3: $thread_{id} \leftarrow threadIdx \% warpSize$
4: **for** $j \leftarrow rpt_A[i] + warp_{id}$ *to* $rpt_A[i+1]$ **do**
5: $\quad a_{col} \leftarrow col_A[j]$
6: $\quad a_{val} \leftarrow val_A[j]$
7: $\quad$ **for** $k \leftarrow rpt_B[a_{col}] + thread_{id}$ *to* $rpt_B[a_{col}+1]$ **do**
8: $\quad\quad b_{col} \leftarrow col_B[k]$
9: $\quad\quad b_{val} \leftarrow val_B[k]$
10: $\quad\quad v \leftarrow Mult(a_{val}, b_{val}, context = a_{col})$
11: $\quad\quad index \leftarrow find(col_C + rpt_C[i], col_C + rpt_C[i+1], b_{col})$
12: $\quad\quad Add(val_C + rpt_C[i] + index, v)$
13: $\quad$ **end for**
14: **end for**

Данная схема распараллеливания позволяет эффективно обращаться к элементам матрицы B во внутреннем цикле, так как в строках 8 и 9 происходят последовательные чтения элементов из массивов col_B и val_B в рамках одного $warp$'а. Но в строке 11 происходит поиск элемента в глобальной памяти, что занимает линейное время. Заметим, что поиск можно заменить на бинарный поиск, так как значения индексов колонок отсортированы в каждой строке в CSR представлении.

Можно еще улучшить производительность, уменьшив количество обращений в глобальную память. Можно выгрузить весь массив $col_C[rpt_C[i] : rpt_C[i+1]]$ в shared memory. Тогда выгрузив массив лишь один раз в shared memory, потом будет возможность делать бинарный поиск в быстрой памяти. Но памяти shared memory может не хватить для всего массива в случае его большого размера. Тогда можно сохранять не каждый элемент в shared memory из массива $col_C[rpt_C[i] : rpt_C[i+1]]$, а только каждый k-ый. В этом случае поиск проходит в два этапа: сначала с помощью бинарного поиска ищется необходимый интервал в shared memory. А затем уже на этом интервале с помощью бинарного поиска ищется элемент в глобальной памяти (см. Рис 2).

Рис. 2: Пример оптимизации с сохранением каждого 4-го элемента



Перед запуском алгоритма производится разбиение строк по группам по количеству элементов в строках для того, чтобы более сбалансированно распределить вычислительные ресурсы. Например, если в данной строке матрицы C максимально возможно 128 элементов, то нет смысла выделять в shared memory массив сильно большего размера.

В итоге, для построения индекса был реализован алгоритм использующий вышеописанные идеи. Из достоинств можно отметить то, что алгоритм способен переиспользовать результаты полученные алгоритмом с реляционной семантикой, а так-же отсутствие промежуточных выделений памяти.

2.4 Работа алгоритма в условиях недостаточного объема видеопамяти

Объем доступной памяти в современных GPU существенно меньше чем в объем CPU RAM. Предложенная мной реализация алгоритма с реляционной семантикой на основе алгоритма Nsparse, по заявлениям авторов, эффективна с точки зрения памяти, но могут возникнуть ситуации, когда все промежуточные структуры не уместятся в память GPU. Для решения этой проблемы я использовал механизм унифици-

рованного адресного пространства (CUDA unified memory [11]).

Обращение к страницам памяти, которые физически не находятся в GPU вызывают page fault и драйвер видеокарты подкачивает данную страницу в память GPU. В случае, если на GPU нет места для подкачиваемой страницы, вымещается другая страница по принципу LRU кеша. В одной из последних работ [6] показано, что использование механизма Unified memory не дает больших накладных расходов при решение BLAS[12] задач.

Для использования Unified memory я предоставил аллокатор памяти, который делегирует запросы на выделение памяти системному вызову *cudaMallocManaged*. С помощью механизма шаблонов C++ данным аллокатором были параметризованы все контейнеры данных, которые используются в реализации.

[11]Unified memory - документация `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd`. Дата посещения: 20.05.2020

[12]Basic Linear Algebra Subprograms — базовые подпрограммы линейной алгебры

3 Экспериментальное исследование

3.1 Описание набора данных для экспериментов

Для экспериментального исследования использовались как синтетически сгенерированные графы так и графы содержащие реальные данные.

Синтетические графы

Для возможности более детального анализа алгоритмов использовался синтетический сгенерированный набор графов. Использовалась модель генерирующая масштабно-инвариантные сети, основанная на методе, предложенным Альберт-Ласло Барабаши [2]. Данная модель моделирует топологию графов из реального мира (социальные, биологические и др. графы). Граф инициализируется кликой из k вершин. На каждом следующем шаге в граф добавляется новая вершина, с k исходящими ребрами. Конечные вершины для ребер выбираются случайно: вероятность того, что вершина будет конечной пропорциональна степени вершины. Процесс повторяется до тех пор, пока в графе не будет n вершин. Дополнительно, каждое ребра помечается случайной меткой из алфавита: $\{a, b, c, d\}$. Для обозначения таких графов будет использоваться нотация $G(n, k)$.

Реальные графы

Для исследования практической применимости алгоритмов использовались реальные данные. Характеристики графов приведены в таблице 1.

- Онтология *Geospecies* из работы [11] — гетерогенный граф содержащий таксономическую иерархию и географическую информацию о видах животных.
- Самые большие RDF графы из набора данных $CFPQ\_data$ [10]:

Таблица 1: Характеристики графов

Name	#V	#E
pathways	6238	37196
go-hierarchy	45007	1960436
enzyme	48815	219390
eclass_514en	239111	1047454
go	272770	1068622
geospecies	450609	2311461
taxonomy-hierarchy	2217333	71167908
taxonomy	5938883	20199337

Конфигурация, на которой проводилось тестирование производительности имела следующие характеристики: Intel core i7-6700 CPU, 3.4GHz, DDR4 64Gb, Geforce GTX 1070 GDDR5 8Gb.

Все замеры производительности проводились 5 раз, выбиралось среднее значение по выборке. Предварительно совершался запуск алгоритмов, который не учитывался в результатах, чтобы проинициализировать все библиотеки (например, на первую инициализацию Cusparse уходит примерно 300мс).

3.2 Анализ производительности выполнения запросов с реляционной семантикой

Рассмотрим влияние размера и плотности синтетических графов на время выполнение запроса. В качестве грамматики используется язык $\{a^n b^m c^m d^n \mid n > 0, m > 0\}$. Грамматика в нормальной форме Хомского

соответствующая данному языку:

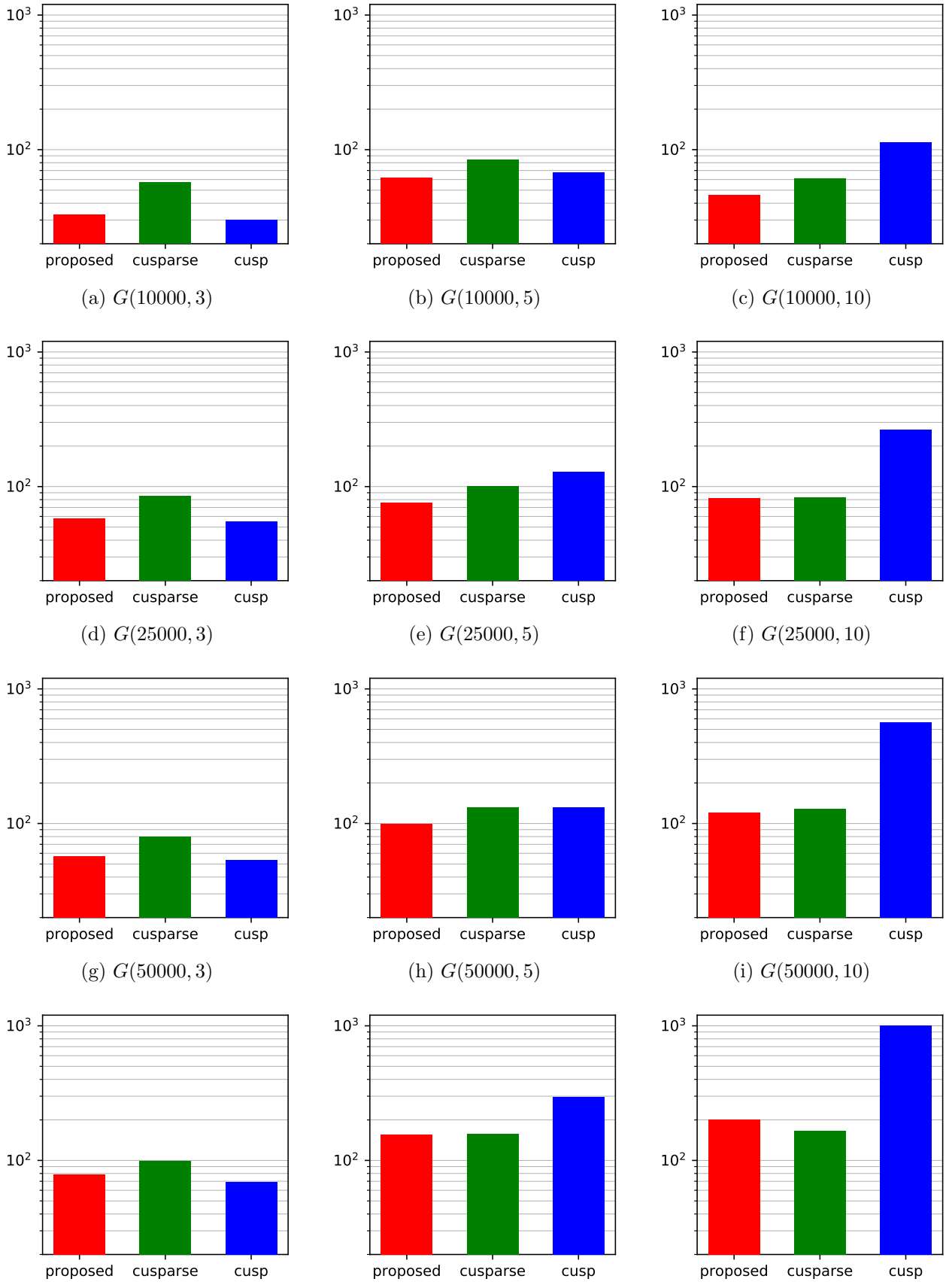
$$S \to A\,S_1 \qquad S \to A\,S_2$$
$$S_1 \to S\,D \qquad S_2 \to X\,D$$
$$X \to B\,C \qquad X \to B\,X_1$$
$$X_1 \to X\,C \qquad A \to a$$
$$B \to b \qquad C \to c$$
$$D \to d$$

Можно заметить (Рис. 3), что поведение предложенного алгоритма на основе $Nsparse$ и алгоритма, использующего $Cusparse$ похожи, так как оба используют идею хеширования. В это же время, алгоритм, использующий $CUSP$ тратит больше время на выполнения на более плотных графе. Однако, данные графы отличаются тем, что распределение элементов по строкам в них равномерное, и алгоритмы на основе хеширования хорошо справляются, так как не возникает ситуация появления больших хеш-таблиц.

Рассмотрим результаты работа алгоритмов на реальных графах. В качестве запросов будут использоваться различные вариации *same generation query*, так как это важный пример запроса из реального мира, который контекстно-свободный но не регулярный.

Для графов из набора данных $CFPQ\_data$ будут использоваться два запроса над отношениями $subClassOf$ и $type$.
Грамматика G_1:

$$S \to subClassOf^{-1}\,S\,subClassOf \qquad S \to type^{-1}\,S\,type$$
$$S \to subClassOf^{-1}\,subClassOf \qquad S \to type^{-1}\,type$$



(a) $G(10000, 3)$ (b) $G(10000, 5)$ (c) $G(10000, 10)$

(d) $G(25000, 3)$ (e) $G(25000, 5)$ (f) $G(25000, 10)$

(g) $G(50000, 3)$ (h) $G(50000, 5)$ (i) $G(50000, 10)$

(j) $G(100000, 3)$ (k) $G(100000, 5)$ (l) $G(100000, 10)$

Рис. 3: Время работы в миллисекундах для синтетических графов с разными параметрами

Грамматика G_2:

$$S \to subClassOf^{-1}\, S\, subClassOf \qquad\qquad S \to subClassOf$$

Для онтологии *Geospecies* используется грамматика из оригинальной работы [11].
Грамматика *Geo*:

$$S \to broaderTransitive\, S\, broaderTransitive^{-1}$$
$$S \to broaderTransitive\, broaderTransitive^{-1}$$

Результаты представлены в таблицах 2, 3, 4. В таблицах представлено число итераций, размер ответа (количество пар вершин, для которых существует путь выводимый из стартового нетерминала грамматики) и время работы в миллисекундах.

Таблица 2: Время выполнения запроса G_1 с реляционной семантикой

граф	#итераций	#ответа	proposed	cusp	cusparse
pathways	2	884	15	13	31
go-hierarchy	2	588976	78	240	600
enzyme	2	396	26	12	43
eclass_514en	2	90994	69	42	4254
go	12	272770	209	338	1698
taxonomy-hie	2	6977582	1718	OOM	2047243
taxonomy	2	191291	859	805	7470

По результатам экспериментов можно отметить, что реализация на основе библиотеки *CUSP* плохо показала себя на графах и грамматиках, которые приводят к большому размеру ответа. Это можно объяснить тем, что количество элементов в матрицах растет, а значит растет и число промежуточных произведений, которые *CUSP* хранит в глобальной памяти. Реализация на основе библиотеки *Cusparse* не имеет таких проблем как *CUSP*, однако время выполнения запросов существенно выше, по сравнению с алгоритмом на основе библиотеки *CUSP*

Таблица 3: Время выполнения запроса G_2 с реляционной семантикой

граф	#итераций	#ответа	proposed	cusp	cusparse
pathways	2	3117	4	4	5
go-hierarchy	3	738937	67	257	558
enzyme	2	8163	5	5	8
eclass_514en	3	96163	35	37	2409
go	13	334850	125	260	708
taxonomy-hie	3	36722046	2741	OOM	1445181
taxonomy	2	2217345	349	581	392

Таблица 4: Время выполнения запроса *Geo* с реляционной семантикой

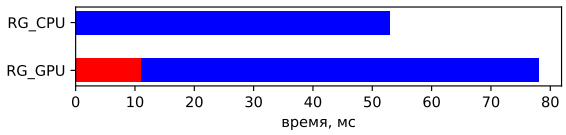
граф	#итераций	#ответа	proposed	cusp	cusparse
geospecies	7	226669749	1026	OOM	5287

и предложенной реализацией на основе алгоритма $Nsparse$.

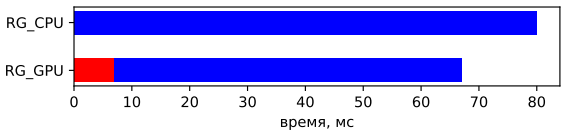
Проведем сравнение предложенной реализации на основе алгоритма $Nsparse$ (обозн. RG_GPU) с реализацией алгоритма использующей CPU на основе $GraphBLAS$ (обозн. RG_CPU), так как данная библиотека используется в RedisGraph для реализации функциональности графовой базы данных и с реализацией $AnnGram_{rel}$ (Intel Xeon E5-4610, 500GB RAM) из работы [11], которая была встроена в Neo4j (обозн. N4J_CPU). На Рис. 4 представлено сравнение реализаций, дополнительно было отмечено время, затраченное на передачу данных между GPU и CPU (красным).

RG_GPU реализация оказалась существенно быстрее чем RG_CPU на графах с самым большим размером ответа (*geospecies*, *taxonomy* – *hierarchy*) — RG_GPU оказалась быстрее от 3 до 8 раз.

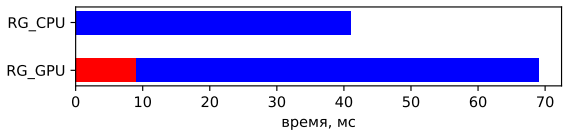
В итоге, предложенный алгоритм на основе Nsparse показал себя лучше, чем другие две $GPGPU$ реализации. Он не обладает проблемой, связанной с нехваткой памяти, и на подавляющем большинстве графов показывает лучшую производительность.



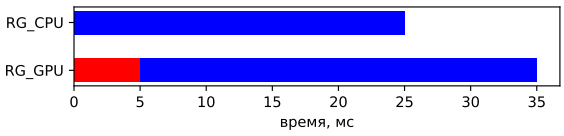
(a) go-hierarchy(G_1)



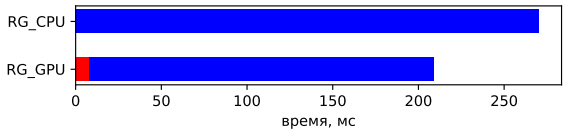
(b) go-hierarchy(G_2)



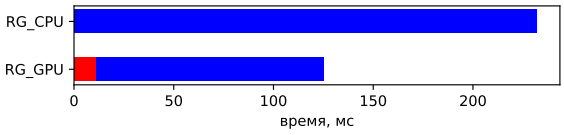
(c) eclass514en(G_1)



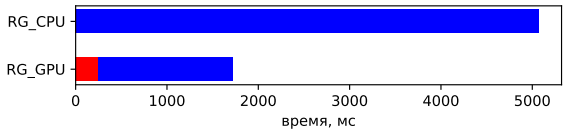
(d) eclass514en(G_2)



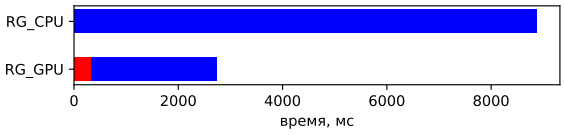
(e) go(G_1)



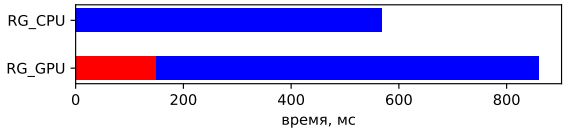
(f) go(G_2)



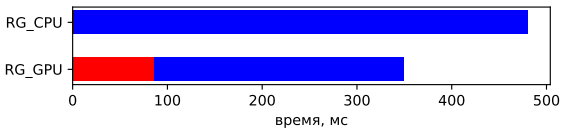
(g) taxonomy-hierarchy(G_1)



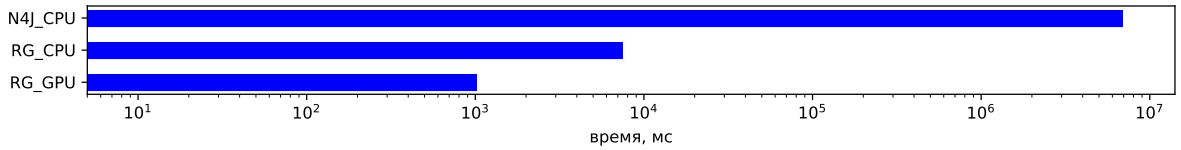
(h) taxonomy-hierarchy(G_2)



(i) taxonomy(G_1)



(j) taxonomy(G_2)



(k) geospecies(Geo)

Рис. 4: Время работы алгоритма выполнения контекстно-свободных запросов с реляционной семантикой для трёх реализаций. Синим цветом выделено время работы алгоритма. Красным цветом выделено время затраченное на обмен данными между CPU и GPU.

3.3 Анализ производительности выполнения запросов с возможностью восстановления пути

Анализ времени построения индекса для восстановления пути проведем на реальных данных. В таблицах 5, 6, 7 приведено время работы предложенной реализации и реализации на основе библиотеки $CUSP$.

Таблица 5: Время построения индекса для восстановления пути для запроса G_1

граф	#итераций	#ответа	proposed	cusp
pathways	2	884	29	15
go-hierarchy	2	588976	185	355
enzyme	2	396	54	17
eclass_514en	2	90994	279	57
go	12	272770	494	475
taxonomy-hie	2	6977582	3081	OOM
taxonomy	2	191291	2073	1131

Таблица 6: Время построения индекса для восстановления пути для запроса G_2

граф	#итераций	#ответа	proposed	cusp
pathways	2	3117	8	7
go-hierarchy	3	738937	171	394
enzyme	2	8163	13	8
eclass_514en	3	96163	100	47
go	13	334850	206	371
taxonomy-hie	3	36722046	4699	OOM
taxonomy	2	2217345	695	818

Таблица 7: Время построения индекса для восстановления пути для запроса Geo

граф	#итераций	#ответа	proposed	cusp
geospecies	7	226669749	1495	OOM

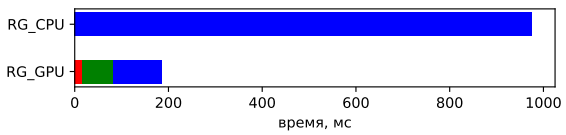
Можно заметить, что время работы алгоритма для построения индекса на основе библиотеки $CUSP$ увеличилось в среднем на 40%. При

этом, список графов, которые не были обработаны по причине нехватки памяти, остался прежним. Время работы предложенного алгоритма для построения индекса увеличилось на 100-200%, относительно предложенной реализации основанной на алгоритме $Nsparse$. Данную разницу можно заметно уменьшить, если на первом этапе алгоритма не с нуля решать задачу построения индекса достижимости (реляционная семантика), а переиспользовать существующий результат при возможности.

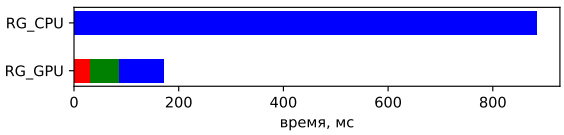
Проведем сравнение предложенной реализации (обозн. RG_GPU) с реализацией алгоритма использующей CPU на основе $GraphBLAS$ (обозн. RG_CPU). На Рис. 5 представлено сравнение реализаций, дополнительно было отмечено время, затраченное на передачу данных между GPU и CPU (красным), так-же время потраченное предложенным алгоритмом на решение задачи с реляционной семантикой (зеленым), которая является первой фазой алгоритма.

На всех запросах RG_GPU реализация показала существенной прирост по сравнению с RG_CPU (от 1.5 до 10 раз), за исключением двух экспериментов.

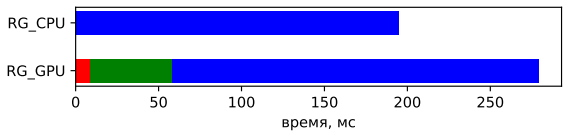
В итоге, предложенный двух-фазный алгоритм для построения индекса показал время работы, сопоставимое с временем работы реализации на основе CUSP, и при этом не испытывал проблем с нехваткой памяти.



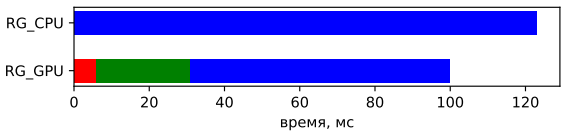
(a) go-hierarchy(G_1)



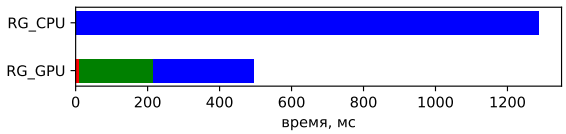
(b) go-hierarchy(G_2)



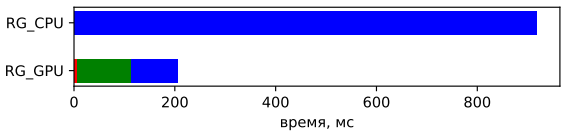
(c) eclass514en(G_1)



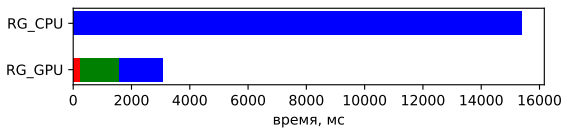
(d) eclass514en(G_2)



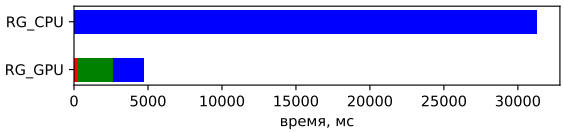
(e) go(G_1)



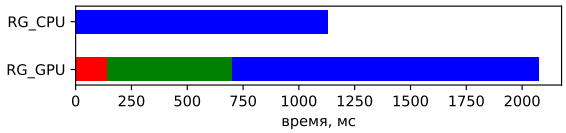
(f) go(G_2)



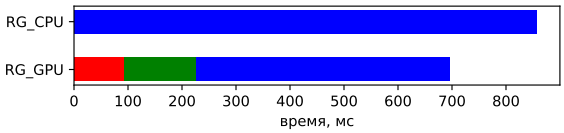
(g) taxonomy-hierarchy(G_1)



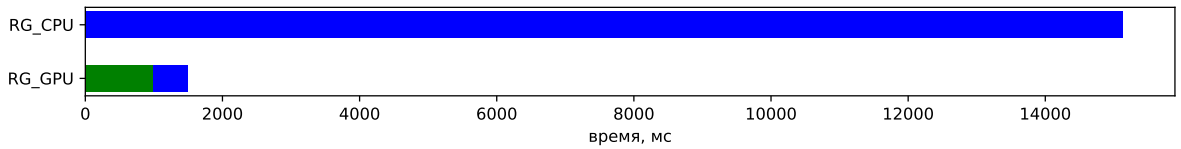
(h) taxonomy-hierarchy(G_2)



(i) taxonomy(G_1)



(j) taxonomy(G_2)



(k) geospecies(Geo)

Рис. 5: Время работы алгоритма построения индекса для восстановления пути. Синим цветом выделено время работы алгоритма. Зеленым цветом выделено время затраченное GPU реализацией на решение задачи достижимости. Красным цветом выделено время затраченное на обмен данными между CPU и GPU.

Заключение

В рамках данной работы была разработана реализация матричного алгоритма [3] выполнения контекстно-свободных запросов для графовой базы данных.

- Разработана реализация алгоритма выполнения контекстно-свободных запросов с реляционной семантикой, использующая $GPGPU$.
- Разработана реализация алгоритма построения индекса для восстановления пути, использующая $GPGPU$.
- Разработанные реализации алгоритмов корректно ведут себя в условиях недостаточного объема видеопамяти.
- Предложенные реализации были исследованы на самых больших графах из набора данных $CFPQ\_data$ [10], в котором содержаться реальные данные. Предложенные реализации оказались первыми реализациями алгоритма выполнения контекстно-свободных запросов, использующими $GPGPU$, способными работать с графами с числом вершин порядка нескольких миллионов.

Полученные результаты позволяют с уверенностью сказать, что использование $GPGPU$ может дать ощутимый прирост производительности (до 10 раз) при выполнении контекстно-свободных запросов на графах из реального мира. Причем как для запросов с реляционной семантикой, так и для построения индекса для восстановления пути.

Полученные результаты делают актуальными дальнейшие исследования алгоритмов выполнения контекстно-свободных запросов, направленными как на улучшение существующих реализаций, так и на полноценную интеграцию в графовые базы данных.

Результаты, полученные в ходе исследования, были изложены в статье, которая была принята на конференцию GRADES-NDA[13] 2020.

[13]Graph Data Management Experiences & Systems and Network Data Analytics, страница конференции: `https://gradesnda.github.io/`

Список литературы

[1] Abiteboul Serge, Hull Richard, Vianu Victor. Foundations of Databases: The Logical Level. — 1st edition. — USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN: 0201537710.

[2] Albert Réka, Barabási Albert-László. Statistical mechanics of complex networks // Rev. Mod. Phys. — 2002. — Jan. — Vol. 74. — P. 47–97. — Access mode: https://link.aps.org/doi/10.1103/RevModPhys.74.47.

[3] Azimov Rustam, Grigorev Semyon. Context-Free Path Querying by Matrix Multiplication // Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA '18. — New York, NY, USA : Association for Computing Machinery, 2018. — Access mode: https://doi.org/10.1145/3210259.3210264.

[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] Bell Nathan, Dalton Steven, Olson Luke. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods // SIAM Journal on Scientific Computing. — 2012. — 01. — Vol. 34.

[6] Chien Steven Wei Der, Peng Ivy Bo, Markidis Stefano. Performance Evaluation of Advanced Features in CUDA Unified Memory // 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC). — 2019. — P. 50–57.

[7] Context-Free Path Queries on RDF Graphs / Xiaowang Zhang, Zhiyong Feng, Xin Wang et al. // The Semantic Web – ISWC 2016 /

Ed. by Paul Groth, Elena Simperl, Alasdair Gray et al. — Cham : Springer International Publishing, 2016. — P. 632–648.

[8] Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication / Azimov Rustam, Grigorev Semyon, Khoroshev Artem, Terekhov Arseniy. — 2020.

[9] Demouth J. Sparse matrix-matrix multiplication on the gpu. — 2012.

[10] Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication / Nikita Mishin, Iaroslav Sokolov, Egor Spirin et al. // Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA'19. — New York, NY, USA : Association for Computing Machinery, 2019. — Access mode: `https://doi.org/10.1145/3327964.3328503`.

[11] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker. — 2019. — 07. — P. 121–132.

[12] Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis / Qirun Zhang, Michael R. Lyu, Hao Yuan, Zhendong Su // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 435–446. — Access mode: `https://doi.org/10.1145/2491956.2462159`.

[13] GBTL-CUDA: Graph Algorithms and Primitives for GPUs / P. Zhang, M. Zalewski, A. Lumsdaine et al. // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2016. — P. 912–920.

[14] Green Oded, Mccoll Rob, Bader David. GPU merge path: a GPU merging algorithm. — 2014. — 11.

[15] Gustavson Fred G. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition // ACM Trans. Math. Softw. — 1978. — Sep. — Vol. 4, no. 3. — P. 250–269. — Access mode: https://doi.org/10.1145/355791.355796.

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

[17] Mathematical foundations of the GraphBLAS / J. Kepner, P. Aaltonen, D. Bader et al. // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — P. 1–9.

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

[19] Nagasaka Y., Nukada A., Matsuoka S. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU // 2017 46th International Conference on Parallel Processing (ICPP). — 2017. — P. 101–110.

[20] Peter DeVries. 2009. The GeoSpecies Knowledge Base ontology. — http://rdf.geospecies.org/geospecies.rdf.gz.

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

[22] 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.

[23] Yang Carl, Buluç Aydin, Owens John Douglas. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU // ArXiv. — 2019. — Vol. abs/1908.01407.

[24] 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.