

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 S.V. Grigorev

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

Saint Petersburg  
2021

# Оглавление

<b>Введение</b>	<b>4</b>
<b>1. Цель и задачи</b>	<b>6</b>
<b>2. Обзор предметной области</b>	<b>7</b>
2.1. Терминология . . . . .	7
2.2. Поиск путей с ограничениями . . . . .	8
2.3. Существующие решения . . . . .	8
2.4. Поиск путей с КС ограничениями через тензорное произведение . . . . .	9
2.5. Вычисления на GPGPU . . . . .	10
2.6. Библиотеки линейной алгебры для GPGPU . . . . .	12
<b>3. Архитектура библиотеки</b>	<b>14</b>
3.1. Компоненты библиотеки . . . . .	15
3.2. Последовательность обработки операций . . . . .	18
<b>4. Детали реализации</b>	<b>20</b>
4.1. Прimitives и операции . . . . .	20
4.2. Cuda-модуль . . . . .	21
4.3. Python-пакет . . . . .	21
4.4. Пример использования . . . . .	22
<b>5. Алгоритм поиска путей с КС ограничениями</b>	<b>24</b>
<b>6. Экспериментальное исследование</b>	<b>26</b>
6.1. Постановка экспериментов . . . . .	26
6.2. Результаты . . . . .	30
<b>7. Заключение</b>	<b>32</b>
<b>Список литературы</b>	<b>34</b>

# 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 [25] и статистический анализ кода [14].

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 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 современных графовых базах данных. Однако, полноценная поддержка запросов с КС ограничениями до сих пор не представлена. Существуют алгоритмы [3,6,7,16,19] для вычисления запросов с КС ограничениями, но потребуется еще время, прежде чем появиться полноценная высокопроизводительная реализация одного из алгоритмов, способная обрабатывать реальные графовые данные.

Работы Никиты Мишина и др. [12] 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 примитивов 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 линейной 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 соответствии с разработанной архитектурой
- Реализация алгоритма поиска путей с КС ограничениями через тензорное произведение с использованием разработанной библиотеки.
- Экспериментальное исследование библиотеки и реализации алгоритма.

## 2. Обзор предметной области

В данной секции представлена базовая теория запросов к графам с КС ограничениями, рассмотрены существующие алгоритмы выполнения данных запросов, с акцентом на перспективное направление алгоритмов, которые полагаются на операции разреженной линейной алгебры. Также в данной секции рассмотрен алгоритм поиска путей на основе тензорного произведения, а также представлены существующие инструменты для работы с примитивами разреженной линейной алгебры на GPGPU, и обоснована необходимость разработки нового подобного инструмента.

### 2.1. Терминология

*Ориентированный граф с метками на ребрах*  $\mathcal{G} = \langle V, E, L \rangle$  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.  $V = \{0 \dots |V| - 1\}$ .

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  $\pi$  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  $\pi = (v_0, l_0, v_1), \dots, (v_{n-1}, l_{n-1}, v_n)$  будем обозначать как  $\omega(\pi) = l_0 \dots l_{n-1}$ , что является конкатенацией меток вдоль этого пути  $\pi$ .

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

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

## 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  $\mathcal{L}$ , 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  $\omega(\pi) \in \mathcal{L}$ . Результат запроса обозначается как  $R = \{(v, u) \mid \exists v\pi u : \omega(\pi) \in \mathcal{L}\}$ .

Поиск путей в графе с семантикой **всех путей** — это поиск всех таких путей  $v\pi u$ , что  $\omega(\pi) \in \mathcal{L}$ . Результат запроса обозначается как  $\Pi = \{v\pi u \mid v\pi u : \omega(\pi) \in \mathcal{L}\}$ . Необходимо отметить, что множество  $\Pi$  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  $v\pi u : \omega(\pi) \in \mathcal{L}$  для каждой пары  $(v, u) \in R$ .

Поскольку язык  $\mathcal{L}$  может быть бесконечным, при составлении запросов используют не множество  $\mathcal{L}$  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 [30]. С того времени были представлены многие работы, в



которых так или иначе предлагалось решение данной проблемы.

Однако недавно Йохем Куиджперс и др. [13] на основе сравнения нескольких алгоритмов [3,16,26] для выполнения запросов с контекстно-свободными ограничениями заключили, что существующие алгоритмы неприменимы для практики из-за проблем с производительностью. Стоит отметить, что алгоритмы, используемые в статье, были реализованы на языке программирования *Java* и исполнялись в среде *JVM* в однопоточном режиме, что могло существенно повлиять на производительность представленных алгоритмов и, соответственно, выводы авторов.

Это подтверждают результаты работы Арсения Терехова и др. [8], в которой с использованием 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. Данное исследование показывает, что эффективная реализация алгоритма на GPU может сделать его применимым для анализа реальных данных.

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)* [17], 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 [17]. 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 в результирующий граф добавляются новые ребра.

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.

## 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 в SIMD (англ. single instruction, multiple data) вычислениях, когда необходимо обрабатывать большие массивы однотипных данных с фиксированным набором команд.

---

**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  $s \in \{0, \dots, \dim(\mathcal{M}_1) - 1\}$  do
7:     for  $S \in \text{getNonterminals}(R, s, s)$  do
8:       for  $i \in \{0, \dots, \dim(\mathcal{M}_2) - 1\}$  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:     $C_3 \leftarrow \text{transitiveClosure}(M'_3)$  ▷ 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:       $s, f \leftarrow \text{getStates}(C_3, i, j)$ 
20:       $x, y \leftarrow \text{getCoordinates}(C_3, i, j)$ 
21:      for  $S \in \text{getNonterminals}(R, s, f)$  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
```

---

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 [29], OpenGL [28], DirectX [11] как API для графических и неспециализированных вычислительных задач, а также OpenCL [24], Nvidia Cuda [21] 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 используются 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* [22] и *CUSP* [9] для платформы Nvidia и *clSPARSE* [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* имеют 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. [23], и специализировать его для булевых значений. Данный алгоритм эксплуатирует возможности видеокарт Nvidia и за счет 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

В данной секции представлена архитектура разработанной библиотеки, описание основных компонентов и модулей, а также показана последовательность обработки операций и концепция организации вычислений на GPU.

Структура библиотеки и ее конечная функциональность в основном определяется следующими высокоуровневыми требованиями, которые продиктованы как конечными вычислительными задачами на GPGPU, так и наличием существующей инфраструктуры для осуществления экспериментов [32].

- *Поддержка вычислений на Cuda-устройстве.* В качестве GPGPU технологии необходимо использовать Cuda, поскольку в библиотеке необходимо повторно использовать уже существующий код для работы с разреженными матрицами из исследования Арсения Терехова и др. [8].
- *Поддержка вычислений на CPU.* Пользователь сможет запустить вычисления с использованием библиотеки, даже если его компьютер не оснащен Cuda-устройством.
- *C-совместимый API для работы с библиотекой.* Прimitives и операции библиотеки доступны в среде с неуправляемыми ресурсами для вызова библиотечных функций без накладных расходов, что критически важно для обработки больших массивов данных.
- *Python-пакет для работы с библиотекой.* Прimitives и операции будут доступны в высокоуровневой среде с управляемыми ресурсами, что снизит порог вхождения и упростит процесс прототипирования алгоритмов.
- *Поддержка логирования, функций для отладки конечных пользовательских алгоритмов.* Конечные пользовательские алгоритмы могут иметь нетривиальную структуру, поэтому требуются встроенные в библиотеку инструменты для отладки.

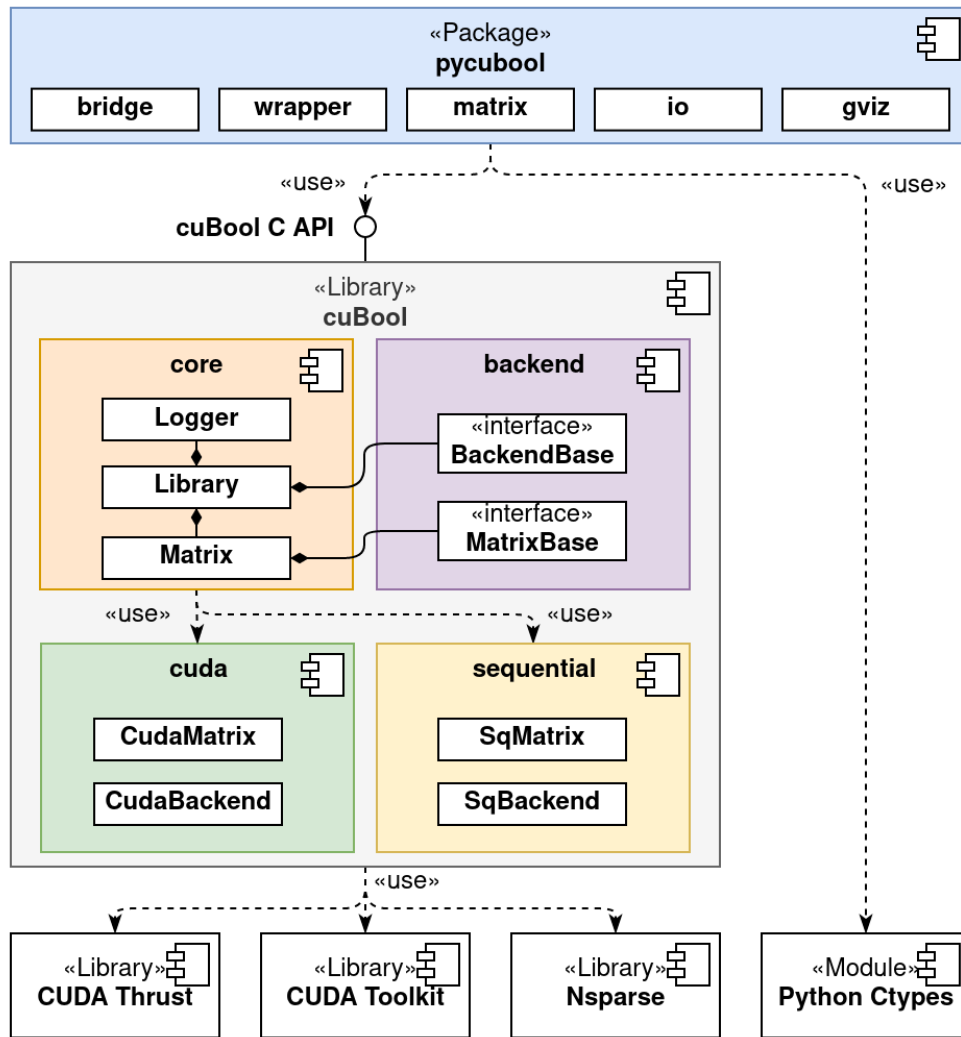


Рис. 1: Архитектура разработанной библиотеки

### 3.1. Компоненты библиотеки

Архитектура библиотеки представлена на рис. 1. Библиотека называется **cuBool**, а Python-пакет для нее — **pycubool**. Библиотека имеет модульную структуру, интерфейс представлен C-совместимым API, что отвечает требованиям проекта. Сторонние зависимости проекта обозначены белым цветом и находятся на нижнем уровне диаграммы. Далее предлагается детально ознакомиться с основными компонентами библиотеки.

## Core

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

Класс **Matrix** является прокси-классом, который осуществляет 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 логирование в выбранный пользователем текстовый файл в процессе использования функций библиотеки, а также позволяет профилировать операции и также сохранять время их выполнения в текстовом виде.

## 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, специфичных 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-устройстве. **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 компьютере пользователя имеется Cuda-устройство. Однако пользователь всегда может выбрать **Sequential** вычисления, если это требуется.

Для работы с технологией Cuda используется как **Cuda Toolkit**, так и библиотека **Cuda Thrust**, которая упрощает работу с высокоуровневыми операциями на GPU. Данные инструменты по умолчанию поставляются в Cuda 10 и выше. В качестве алгоритмов сложения и умножения матрицы повторно используются результаты исследования Арсения Терехова и др. [8], которые были выделены в отдельную библиотеку **Nsparse**.

## Sequential

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

Данный вычислительный модуль используется по умолчанию на устройствах без Cuda-устройства. 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.

## Py cubool

Python-пакет предоставляет доступ к примитивам и операциям библиотеки в языковой среде Python. Модуль **matrix** предоставляет доступ к классу матрицы и основным операциям, доступным 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 матрицы в текстовом формате, а также экспортировать набор матриц в виде графа в формате GraphViz, что может быть полезно для отладки пользовательских алгоритмов.

## 3.2. Последовательность обработки операций

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

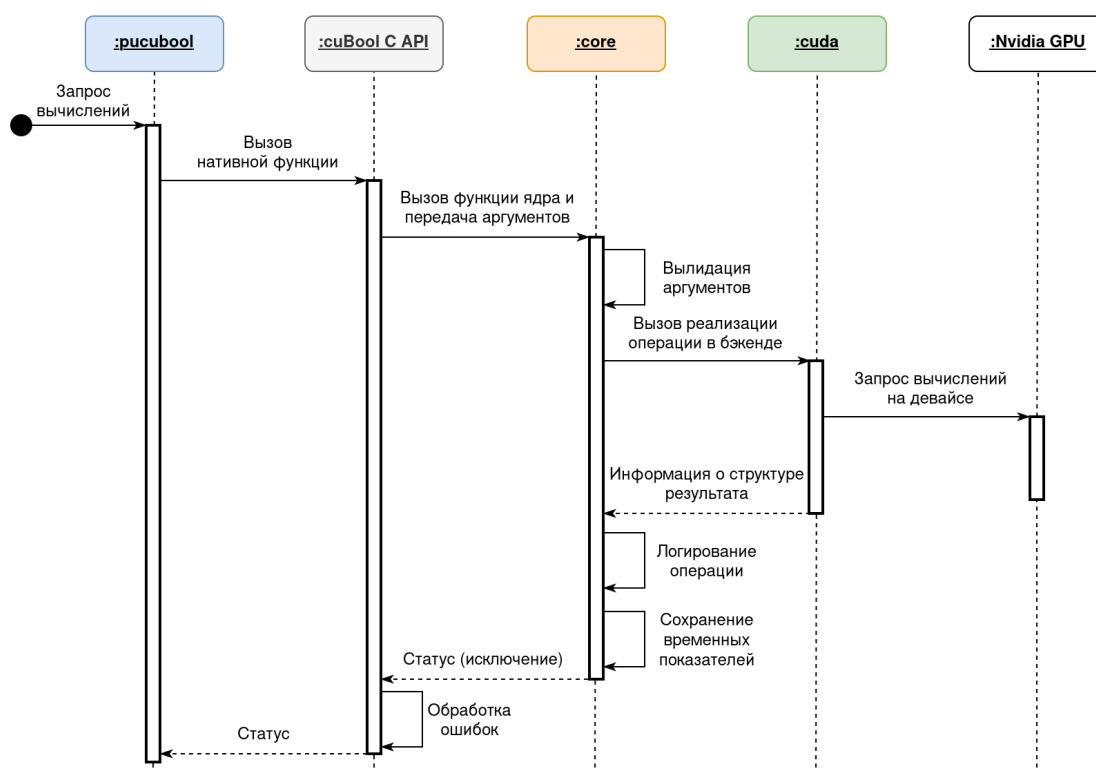


Рис. 2: Последовательность выполнения вычислительной матричной операции на Nvidia GPU с использованием pycubool

Пользовательский Python-код инициирует выполнение операции над матрицей 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, осуществляет валидацию аргументов, а также определяет допустимость выполнения

операции. Далее вызов передается непосредственно вычислительному модулю **cuda**, который осуществляет подготовку и непосредственный запуск вычислений на стороне **Nvidia GPU**.

Когда вычисления завершаются, **cuda**-модуль обновляет состояние матриц в соответствии с полученными результатами. Модуль **core** осуществят 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** осуществляет финальную обработку исключения (если таковое возникло), и возвращает числовой идентификатор 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.

## 4. Детали реализации

В данной секции изложены основные детали в реализации библиотеки примитивов разреженной линейной булевой алгебры *cuBool*.

Разработка библиотеки 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. [23], 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  $M = N[i : k, j : t]$
- Редуцирование матрицы к вектору  $V = \text{reduceToColumn}(M)$
- 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* [20], которые выполняются на графическом процессоре. **Thrust** поставляется совместно с инструментами CUDA-разработки и не требует настройки дополнительных зависимостей.

## 4.3. Python-пакет

Все примитивы и операции библиотеки **cuBool** доступны внутри Python-пакета **ruscubool**. 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 **ruscubool** 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

---

### Listing 2 Пример вычисления транзитивного замыкания с использованием cuBool C API

---

```
1 #include <cubool/cubool.h>
2
3 cuBool_Status TransitiveClosure(cuBool_Matrix A, cuBool_Matrix* T) {
4     cuBool_Matrix_Duplicate(A, T);                /* Копируем матрицу смежности A */
5
6     cuBool_Index total = 0;
7     cuBool_Index current;
8     cuBool_Matrix_Nvals(*T, &current);            /* Количество ненулевых значений */
9
10    while (current != total) {                      /* Пока результат меняется */
11        total = current;
12        cuBool_MxM(*T, *T, *T, CUBOOL_HINT_ACCUMULATE); /* T += T x T */
13        cuBool_Matrix_Nvals(*T, &current);
14    }
15
16    return CUBOOL_STATUS_SUCCESS;
17 }
```

---

---

### Listing 3 Пример вычисления транзитивного замыкания с использованием пакета русcubool

---

```
1 import русcubool
2
3 def transitive_closure(a: русcubool.Matrix):
4     t = a.duplicate()                             # Копируем матрицу смежности A
5     total = 0                                     # Количество ненулевых значений результата
6
7     while total != t.nvals:                       # Пока результат меняется
8         total = t.nvals
9         t.mxM(t, out=t, accumulate=True)         # t += t x t
10
11    return t
```

---

В качестве примера рассмотрим проблему вычисления *транзитивного замыкания* (англ. transitive closure) для некоторого ориентирован-

ного 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.

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 задачу на языке Python. Здесь в качестве входного аргумента используется матрица смежности графа, в качестве результата возвращается матрица смежности графа после транзитивного замыкания.

## 5. Алгоритм поиска путей с КС ограничениями

В данной секции представлены основные детали реализации алгоритма [7] поиска путей с КС ограничениями через тензорное произведение на GPGPU.

Алгоритм реализован с использованием пакета **puscubool**. 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 **puscubool** реализован классический матричный алгоритм Рустама Азимова [3], требуемый для корректного сравнения производительности с алгоритмом на основе тензорного произведения.

Реализации упомянутых алгоритмов доступны в рамках открытого проекта **CFPQ PyAlgo** [32]. Данный проект предоставляет инфраструктуру для осуществления замеров производительности, а также для загрузки и конвертации данных, требуемых для экспериментов. Архитектура данного стенда представлена на рис. 3. Тестовый стенд позволяет использовать различные библиотеки и инструменты для реализации в модуле **Algorithms** алгоритмов поиска путей с КС ограничениями. Также стенд предоставляет утилиты для загрузки входных графов в модуле **Graph**, для загрузки грамматик и преобразования их в требуемый формат в модуле **Grammar** соответственно. Пакет **CFPQ-Data** [15] используется для загрузки из локального или удаленного хранилища актуальной версии набора данных для замеров.

В инфраструктуре уже доступны базовые реализации алгоритма Рустама Азимова [3] и алгоритма на основе тензорного произведения для вычислений на CPU. В качестве основы эти реализации используют



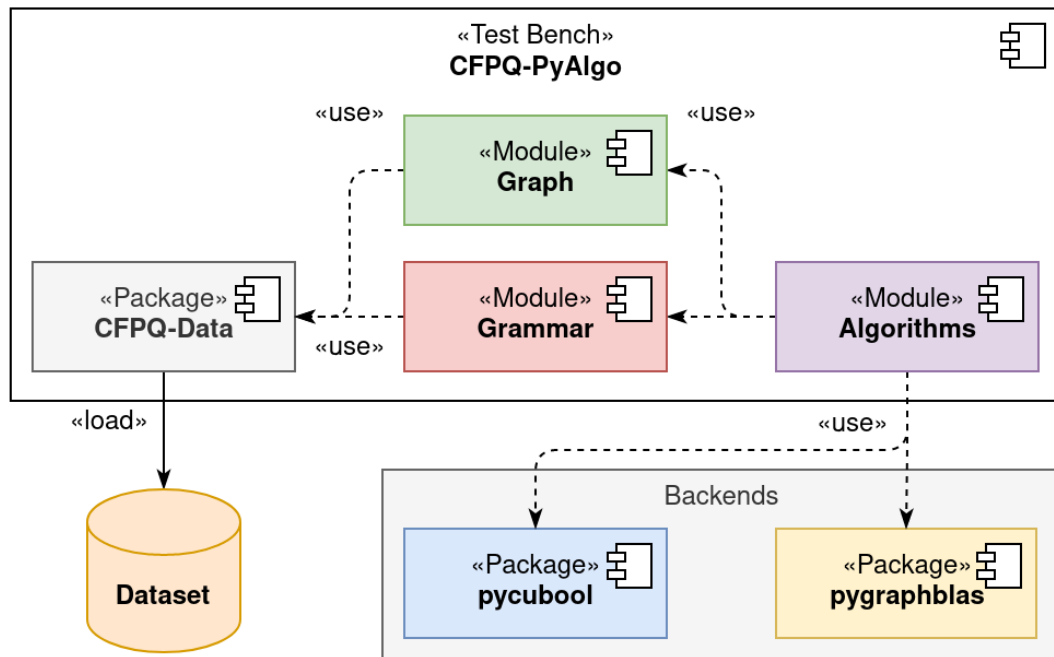


Рис. 3: Архитектура стенда для тестирования алгоритмов

пакет **pygraphblas**, который предоставляет доступ к примитивам разреженной линейной алгебры из стандарта GraphBLAS API [18] и его эталонной реализации SuiteSparse [10]. Данные реализации алгоритмов также используются для проведения экспериментального исследования.

## 6. Экспериментальное исследование

В данной секции представлены результаты экспериментального исследования как разработанной библиотеки, так и реализации алгоритма поиска путей с КС ограничениями. Основная задача исследования: оценить производительность полученных артефактов в задачах анализа данных близких к реальным, и сравнить полученные показатели с другими схожими решениями в области.

### 6.1. 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, под управлением ОС Ubuntu 20.04.

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

#### Исследовательские вопросы

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

- В1:** Какова производительность отдельных операций реализованной библиотеки примитивов разреженной линейной булевой алгебры на GPGPU по сравнению с существующими аналогами?
- В2:** Какова производительность реализованного алгоритма поиска путей через тензорное произведение на GPGPU по сравнению с существующими аналогами, также полагающимися на примитивы линейной алгебры?

B1 направлен на определение эффективности отдельных матричных операций в реализованной библиотеке. В качестве таких операций выступают *матричное умножение* и *матричное сложение* в булевом полукольце, как наиболее распространенные 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 качестве таких библиотек были выбраны CUSP и cuSPARSE для Nvidia Cuda, clSPARSE для OpenCL, и SuiteSparse для CPU. CUSP предоставляет реализацию операций, основанную на шаблонах для параметризации используемого типа данных, однако библиотека не делает каких-либо дополнительных оптимизаций конкретно для булевых значений. cuSPARSE и clSPARSE предоставляют операции только для основных типов данных с плавающей запятой. Однако данное ограничение можно обойти, если интерпретировать ненулевые значения как *true*. Библиотека SuiteSparse является эталонной реализацией GraphBLAS API и имеет встроенное булево полукольцо для вычислений.

B2 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. Также для сравнения будут использованы реализации этих алгоритмов для вычислений на CPU, чтобы оценить вклад GPGPU вычислений в ускорение работы алгоритмов. В качестве тестовой инфраструктуры используется стенд, описанный в разделе 5.

## Набор данных

Для замеров производительности отдельных операций реализованной библиотеки были выбраны 10 различных квадратных матриц из известной коллекции университета Флориды [27] для проверки эффективности алгоритмов, реализующих операции над разреженными мат-

Таблица 1: Разреженные матричные данные

Матрица $M$	Кол-во Строк $R$	Nnz $M$	Nnz/ $R$	Max Nnz/ $R$	Nnz $M^2$	Nnz $M + M^2$
wing	62,032	243,088	3.9	4	714,200	917,178
luxembourg_osm	114,599	239,332	2.0	6	393,261	632,185
amazon0312	400,727	3,200,400	7.9	10	14,390,544	14,968,909
amazon-2008	735,323	5,158,388	7.0	10	25,366,745	26,402,678
web-Google	916,428	5,105,039	5.5	456	29,710,164	30,811,855
roadNet-PA	1,090,920	3,083,796	2.8	9	7,238,920	9,931,528
roadNet-TX	1,393,383	3,843,320	2.7	12	8,903,897	12,264,987
belgium_osm	1,441,295	3,099,940	2.1	10	5,323,073	8,408,599
roadNet-CA	1,971,281	5,533,214	2.8	12	12,908,450	17,743,342
netherlands_osm	2,216,688	4,882,476	2.2	7	8,755,758	13,626,132

рицами. 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 (соответствует числу столбцов), а также количество ненулевых элементов 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 артефактов имитирует шаг транзитивного замыкания. Эффективное вычисление этого шага во многом определяет производительность конечных пользовательских алгоритмов на графах.

Для замеров производительности алгоритмов поиска путей с КС ограничениями используется коллекция графовых данных лаборатории языковых инструментов JetBrains Research [15], которая использовалась в ряде работ [3,7,8,12] для подобных экспериментов. Данная коллекция содержит RDF данные, а также графы программ, полученные из различных модулей ядра Linux (*arch*, *crypto*, *drivers*, *fs*). Информация о графах представлена в таблице 2. В таблице приведено название графа, количество вершин и ребер, а также количество ребер с метками *sco* (*subClassOf*), *type*, *bt* (*broaderTransitive*), *a* и *b*. В верхней секции таблицы расположены RDF графы, а в нижней — графы программ. Для анализа RDF данных используются *same-generations queries*  $G_1$  (выражение 1),  $G_2$  (выражение 2), а также  $G_{geo}$  (выражение 3), представленная

Таблица 2: RDF графы и графы программ для КС запросов

Граф $\mathcal{G}$	$ V $	$ E $	Кол-во sco	Кол-во type	Кол-во bt	Кол-во a	Кол-во d
eclass_514en	239 111	523 727	90 512	72 517	—	—	—
enzyme	48 815	109 695	8 163	14 989	—	—	—
geospecies	450 609	2 201 532	0	89 062	20 867	—	—
go	272 770	534 311	90 512	58 483	—	—	—
go-hierarchy	45 007	980 218	490 109	0	—	—	—
taxonomy	5 728 398	14 922 125	2 112 637	2 508 635	—	—	—
arch	3 448 422	5 940 484	—	—	—	671 295	2 298 947
crypto	3 464 970	5 976 774	—	—	—	678 408	2 309 979
drivers	4 273 803	7 415 538	—	—	—	858 568	2 849 201
fs	4 177 416	7 218 746	—	—	—	824 430	2 784 943

в исследовании Йохема Куиджперса и др. [13] для анализа *geospecies* RDF. Графы программ используются для анализа указателей. Данная проблема сведена к запросам с КС ограничениями с использованием грамматики  $G_{ma}$  (выражение 4), предложенной в исследовании Чжэн Синь и др. [31].

$$S \rightarrow \overline{subClassOf} \ S \ subClassOf \mid \overline{type} \ S \ type \mid \overline{subClassOf} \ subClassOf \mid \overline{type} \ type \quad (1)$$

$$S \rightarrow \overline{subClassOf} \ S \ subClassOf \mid \overline{subClassOf} \quad (2)$$

$$S \rightarrow \overline{broaderTransitive} \ S \ \overline{broaderTransitive} \mid \overline{broaderTransitive} \ \overline{broaderTransitive} \quad (3)$$

$$S \rightarrow \bar{d} \ V \ d \quad (4)$$

$$V \rightarrow ((S?)\bar{a})^*(S?)(a(S?))^*$$

## Метрики

Для ответа на поставленные исследовательские вопросы в качестве метрик производительности используется время, требуемое для выполнения операции, а также пиковое количество потребляемой RAM или

VRAM (в зависимости от платформы инструмента) в момент вычисления. Показатели времени усреднены по 10 запускам. Отклонение показателей составляет не более 10% для каждого отдельного эксперимента. Предварительно совершался не учитывающийся в замерах запуск, чтобы инициализировать начальное состояние тестируемых библиотек. Показатели потребления RAM получены с помощью дополнительных функций библиотеки C для семейства ОС на базе ядра Linux. Показатели потребления VRAM получены с помощью инструмента *nvidia-smi*, который с точностью до 1 миллисекунды позволяет отслеживать количество потребляемой памяти процессом ОС на стороне видеокарты.

## 6.2. Результаты

*B1: Какова производительность отдельных операций реализованной библиотеки примитивов разреженной линейной булевой алгебры на GPGPU по сравнению с существующими аналогами?*

Результаты эксперимента по сравнению производительности матричного произведения представлены в таблице 3. Реализованная библиотека *cuBool* показывает лучшие результаты по сравнению с другими библиотеками. Используемый 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 4. 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*.

Таблица 3: Матричное умножение (время (t) в миллисекундах, память (m) в мегабайтах, отклонение в пределах 10%)

Матрица $M$	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 4: Поэлементное матричное сложение (время (t) в миллисекундах, память (m) в мегабайтах, отклонение в пределах 10%)

Матрица $M$	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

*В2: Какова производительность реализованного алгоритма поиска путей через тензорное произведение на GPGPU по сравнению с существующими аналогами, также полагающимися на примитивы линейной алгебры?*

## 7. 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 русubool, который оступен для скачивания через пакетный менеджер PyPI.
- С использованием русubool 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 реализованной библиотеки и алгоритма. 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. GPGPU Реализация алгоритма поиска путей с КС ограничениями показывает ускорение до 6 раз по сравнению с CPU версией, что делает ее более применимой для реального анализа данных.



На основе результатов, полученных в данном исследовании, была написана статья, принятая на конференцию GrAPL 2021<sup>1</sup>.

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/пусcubool/>.

---

<sup>1</sup>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. — IWOCCL '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] Davis Timothy A. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra // ACM Trans. Math. Softw. — 2019. — Dec. — Vol. 45, no. 4. — Access mode: <https://doi.org/10.1145/3322125>.
- [11] 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).
- [12] 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.
- [13] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaeker // 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>.
- [14] 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>.
- [15] Graphs and grammars for Context-Free Path Querying algorithms evaluation // Github. — 2021. — Access mode: [https://github.com/JetBrains-Research/CFPQ\\_Data](https://github.com/JetBrains-Research/CFPQ_Data) (online; accessed: 11.03.2021).
- [16] Hellings Jelle. Path Results for Context-free Grammar Queries on Graphs. — 2015. — 02.

- [17] 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.
- [18] Mathematical foundations of the GraphBLAS / J. Kepner, P. Aalto-nen, D. Bader et al. // 2016 IEEE High Performance Extreme Com-puting Conference (HPEC). — 2016. — Sep. — P. 1–9.
- [19] Medeiros Ciro, Musicante Martin, Costa Umberto. An Algorithm for Context-Free Path Queries over Graph Databases. — 2020. — 04.
- [20] NVIDIA. CUDA Thrust // NVIDIA Developer Zone. — 2020. — Access mode: <https://docs.nvidia.com/cuda/thrust/index.html> (online; accessed: 16.12.2020).
- [21] NVIDIA. CUDA Toolkit Documentation // NVIDIA Devel-oper Zone. — 2020. — Access mode: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (online; accessed: 01.12.2020).
- [22] NVIDIA. cuSPARSE reference guide // NVIDIA Developer Zone. — 2020. — Access mode: <https://docs.nvidia.com/cuda/cuspars/index.html> (online; accessed: 09.12.2020).
- [23] Nagasaka Yusuke, Nukada Akira, Matsuoka Satoshi. High-Performance and Memory-Saving Sparse General Matrix-Matrix Mul-tiplication for NVIDIA Pascal GPU. — 2017. — 08. — P. 101–110.
- [24] OpenCL: Open Standard for Parallel Programming of Heterogeneous Systems // Khronos website. — 2020. — Access mode: <https://www.khronos.org/opencvl/> (online; accessed: 08.12.2020).
- [25] Quantifying variances in comparative RNA secondary structure pre-diction / James Anderson, Adám Novák, Zsuzsanna Sükösd et al. // BMC bioinformatics. — 2013. — 05. — Vol. 14. — P. 149.

- [26] 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.
- [27] 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).
- [28] 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).
- [29] 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).
- [30] Yannakakis Mihalīs. 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>.
- [31] Zheng Xin, Rugina Radu. Demand-driven Alias Analysis for C // SIGPLAN Not. — 2008. — Jan. — Vol. 43, no. 1. — P. 197–208. — Access mode: <http://doi.acm.org/10.1145/1328897.1328464>.
- [32] A collection of CFPQ algorithms implemented in PyGraph-BLAS // Github. — 2020. — Access mode: [https://github.com/JetBrains-Research/CFPQ\\_PyAlgo](https://github.com/JetBrains-Research/CFPQ_PyAlgo) (online; accessed: 16.12.2020).