Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 17Б.11-мм

Орачев Егор Станиславович

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

Отчёт по преддипломной практике

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

Санкт-Петербург
2020

Оглавление

Введение **3**

1. Цель и задачи **5**

2. Обзор предметной области **6**

2.1. Терминология 6

2.2. Поиск путей с ограничениями 7

2.3. Существующие решения 7

2.4. Поиск путей с КС ограничениями через тензорное произведение 8

2.4.1. Рекурсивный автомат 9

2.4.2. Пересечение рекурсивного автомата и графа . . . 10

2.4.3. Описание алгоритма 11

2.5. Вычисления на GPGPU 12

2.6. Библиотеки линейной алгебры для GPGPU 13

3. Разработка библиотеки линейной алгебры на GPGPU **15**

3.1. Примитивы и операции 15

3.2. Описание реализации 16

3.2.1. Ядро библиотеки 16

3.2.2. Python модуль 17

3.3. Пример использования 18

4. Текущий прогресс **19**

Список литературы **21**

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 статический анализ кода [15].

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

мантику, требуются различные программные и алгоритмические средства для его выполнения.

Запросы с регулярными ограничениями изучены достаточно хорошо, языковая и программная поддержка выполнения подобных запросов присутствует в некоторых в современных графовых базах данных. Однако, полноценная поддержка запросов с КС ограничениями до сих пор не представлена. Существуют алгоритмы [3,6,7,17,19] для вычисления запросов с КС ограничениями, но потребуется еще время, прежде чем появиться полноценная высокпроизводительная реализация одного из алгоритмов, способная обрабатывать реальные графовые данные.

Работы Никиты Мишина и др. [13] и Арсения Терехова и др. [8] показывают, что реализация алгоритма Рустама Азимова [3], основанного на операциях линейной алгебры, с использованием GPGPU для выполнения наиболее вычислительно сложных частей алгоритма, дает *существенный* прирост в производительности.

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

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

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

1. Цель и задачи

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

- Реализация библиотеки примитивов линейной булевой алгебры для работы с разреженными данными на GPGPU
- Реализация интерфейса библиотеки для работы в среде с управляемыми ресурсами с целью прототипирования алгоритмов и использования существующей тестовой инфраструктуры [30] по работе с графовыми данными
- Реализация алгоритма поиска путей с КС ограничениями через тензорное произведение с использованием разработанной библиотеки
- Экспериментальное исследование реализованного алгоритма с использованием синтетических и реальных данных в тестовой инфраструктуре

2. Обзор предметной области

2.1. Терминология

В этой секции изложены основные определения и факты из теории графов и формальных языков, необходимые для понимания предметной области.

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

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

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

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

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

2.2. Поиск путей с ограничениями

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

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

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

Необходимо отметить, что множество Π может быть бесконечным, поэтому в качестве результата запроса предполагается не всё множество в явном виде, а некоторый *итератор*, который позволит последовательно извлекать все пути.

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

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

2.3. Существующие решения

Впервые проблема выполнения запросов с контекстно-свободными ограничениями была сформулирована в 1990 году в работе Михалиса Яннакакиса [29]. С того времени были представлены многие работы, в которых так или иначе предлагалось решение данной проблемы. Однако в недавнем исследовании Йохем Куиджперс и др. [14] на основе сравнения нескольких алгоритмов [3, 17, 26] для выполнения запросов с

контекстно-свободными ограничениями заключили, что существующие алгоритмы неприменимы для анализа реальных данных в силу того, что обработка таких данных занимает значительное время. Стоит отметить, что алгоритмы, используемые в статье, были реализованы на языке программирования *Java* и исполнялись в среде *JVM* в однопоточном режиме, что не является сколь-угодно производительным решением.

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

Алгоритм Рустама Азимова [8] способен выполнять запросы только в семантике одного пути. Поскольку в качестве формализма для представления грамматики КС запроса используется *ослабленная нормальная форма Хомского (ОНФХ)* [18], 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

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

Далее предлагается рассмотреть описание данного алгоритма и используемую им технику сведения вычислений к операциям линейной алгебры.

2.4.1. Рекурсивный автомат

Для представления входной грамматики КС запроса алгоритм [7] использует *рекурсивный автомат*. Данный формализм является своего рода обобщением *недетерминированного конечного автомата* на случай КС языков. Для понимания того, как он устроен, обратимся к теории формальных языков.

Недетерминированный конечный автомат (НКА) $F = \langle \Sigma, Q, Q_s, Q_f, \delta \rangle$ это пятерка объектов, где:

- Σ конечное множество входных символов или алфавит
- Q конечное множество состояний
- $Q_s \subseteq Q$ множество стартовых состояний
- $Q_f \subseteq Q$ множество конечных состояний
- $\delta : \Sigma \times Q \to 2^Q$ функция переходов автомата

Язык, допускаемый автоматом F будем обозначать как $L(F)$. Любое регулярное выражение может быть преобразовано в соответствующий НКА [18].

Рекурсивный автомат (РА) $R = \langle M, m, \{C_i\}_{i \in M} \rangle$ это тройка объектов, где:

- M конечное множество меток компонентных НКА, называемых далее *модули*
- m метка стартового модуля
- $\{C_i\}$ множество модулей, где модуль $C_i = \langle \Sigma \cup M, Q_i, S_i, F_i, \delta_i \rangle$ состоит из:

– $\Sigma \cup M$ множество символов модуля, $\Sigma \cap M = \emptyset$

– Q_i конечное множество состояний модуля, $Q_i \cap Q_j = \emptyset, \forall i \neq j$

– $S_i \subseteq Q_i$ множество стартовых состояний модуля

– $F_i \subseteq Q_i$ множество конечных состояний модуля

– $\delta_i : (\Sigma \cup M) \times Q_i \to 2^{Q_i}$ функция переходов

Рекурсивный автомат ведет себя как набор НКА или модулей [2]. Эти модули очень сходны с НКА при обработке входных последовательностей символов, однако они способны обрабатывать дополнительные *рекурсивные вызовы* за счет неявного *стека вызовов*, который присутствует во время работы РА. С точки зрения прикладного программиста это похоже на рекурсивные вызовы одних функций из других с той разницей, что вместо функций здесь выступают модули РА.

Рекурсивные автоматы по своей вычислительной мощности эквивалентны автоматам на основе стека [2]. А поскольку подобный стековый автомат способен распознавать КС грамматику [18], рекурсивные автоматы эквивалентны КС грамматикам. Это позволяет корректно использовать РА для представления входной КС грамматики запроса.

2.4.2. Пересечение рекурсивного автомата и графа

Классический алгоритм [18] *пересечения* двух НКА F^1 и F^2 позволяет построить новый НКА F с таким свойством, что он допускает пересечение исходных регулярных языков, т.е. $L(F) = L(F^1) \cap L(F^2)$. Формально, для $F^1 = \langle \Sigma, Q^1, Q^1_S, Q^1_F, \delta^1 \rangle$ и $F^2 = \langle \Sigma, Q^2, Q^2_S, Q^2_F, \delta^2 \rangle$ строится новый НКА $F = \langle \Sigma, Q, Q_S, Q_F, \delta \rangle$, где:

- $Q = Q^1 \times Q^2$
- $Q_S = Q^1_S \times Q^2_S$
- $Q_F = Q^1_F \times Q^2_F$
- $\delta : \Sigma \times Q \to Q$ и $\delta(s, \langle q_1, q_2 \rangle) = \langle q'_1, q'_2 \rangle$, если $\delta^1(s, q_1) = q'_1$ и $\delta^2(s, q_2) = q'_2$

Интерпретируя ориентированный граф с метками как некоторый конечный автомат, в котором все вершины графа являются одновременно начальными и конечными состояниями автомата, а ребра графа — переходами между состояниями автомата, возможно пересечь этот граф и некоторый НКА, используя алгоритм пересечения, описанный выше. Однако, если представить граф и функцию переходов КА, тоже интерпретируемую как граф, в виде матриц смежности, можно использовать *произведение Кронекера* для построения функции переходов автомата пересечения.

Произведение Кронекера для двух матриц A и B размера $m_1 \times n_1$ и $m_2 \times n_2$ с поэлементной операцией умножения $\cdot$ дает матрицу $M = A \otimes B$ размером $m_1 * m_2 \times n_1 * n_2$, где $M[u * m_2 + v, p * n_2 + q] = A[u, p] \cdot B[v, q]$.

Поскольку РА состоит из набора модулей, которые по своей структуре не сильно отличаются от классических НКА, это дает идею для применения похожего алгоритма пересечения РА и графа, с той разницей, что процесс пересечения будет итеративным и будет включать в себя транзитивное замыкание, чтобы учесть *рекурсивные вызовы*, присутствующие в РА.

2.4.3. Описание алгоритма

В листинге 1 представлен псевдокод алгоритма [7]. 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 и компактным, так как всю сложность выполнения он перекладывает на операции линейной алгебры, которые должны быть реализованы в сторонних высокопроизводительных библиотеках.

Listing 1 Поиск путей через произведение Кронекера

```
1:  function KroneckerProductBasedCFPQ(G, 𝒢)
2:      R ← Рекурсивный автомат для грамматики G
3:      ℳ1 ← Матрица переходов R в булевой форме
4:      ℳ2 ← Матрица смежности 𝒢 в булевой форме
5:      C3 ← Пустая матрица
6:      for s ∈ {0, ..., dim(ℳ1) − 1} do
7:          for S ∈ getNonterminals(R, s, s) do
8:              for i ∈ {0, ..., dim(ℳ2) − 1} do
9:                  M2^S[i, i] ← {1}
10:             end for
11:         end for
12:     end for
13:     while Матрица смежности ℳ2 изменяется do
14:         ℳ3 ← ℳ1 ⊗ ℳ2                          ▷ Вычисление произведения Кронекера
15:         M3' ← ⋁_{M3^a ∈ ℳ3} M3^a                ▷ Слияние матриц в одну булеву матрицу достижимости
16:         C3 ← transitiveClosure(M3')             ▷ Транзитивное замыкание для учета рекурсивных вызовов
17:         n × n ← dim(M3)
18:         for (i, j) | C[i, j] ≠ 0 do
19:             s, f ← getStates(C3, i, j)
20:             x, y ← getCoordinates(C3, i, j)
21:             for S ∈ getNonterminals(R, s, f) do
22:                 M2^S[x, y] ← {1}
23:             end for
24:         end for
25:     end while
26:     return ℳ2, C3
27: end function
```

2.5. Вычисления на GPGPU

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

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

На данный момент существует несколько промышленных стандартов для создания программ, использующих графический процессор, одними из которых являются Vulkan [28], OpenGL [27], Direct3D [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 используется результаты исследования [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. В качестве такой библиотеки для выполнения матричных операций на центральном процессоре авторы исследований [7,8] использовали *SuiteSparse* [10,12]. Это эталонная реализация стандарта *GraphBLAS* [16], который был разработан как некоторый инструмент для реализации алгоритмов обработки графов на языке линейной алгебры.

Экспериментальное исследование [13] по реализации алгоритма Рустама Азимова [3] на GPGPU с использованием операций над плотными булевыми матрицами показало, что вычисление на графическом процессоре дает значительный прирост производительности при обработке синтетических данных и данных среднего размера. Однако реальные графовые данные насчитывают порядка 10^5 – 10^9 вершин и являются сильно разреженными, т.е. количество ребер в графе сравнимо с коли-

чеством вершин, поэтому плотные матрицы не подходят для представления такого типа данных.

Библиотеки *cuSPARSE* [22] и *CUSP* [5] для платформы NVIDIA предоставляют функциональность для работы с разреженными данными, однако они имеют специализацию только для стандартных типов, 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, библиотека *CUSP* имеет открытый исходный код и свободную лицензию, однако используемый ею алгоритм умножения разреженных матриц *слишком* требователен к ресурсам памяти, что делает его неприменимым для обработки данных большого размера.

В работе Арсения Терехова и др. [8] была предпринята попытка самостоятеьно реализовать алгоритм умножения разреженных матриц *Nsparse* [23] и специализировать его для булевых значений. Данный алгоритм эксплуатирует возможности видеокарт от NVIDIA и за счет бóльшего времени на обработку 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*.

Поэтому было принято решение расширить реализацию матричных операций из работы [8], дополнив ее требуемыми для алгоритма [7] примитивами, и оформить решение в виде самостоятельной библиотеки, которая позволила бы в дальнейшем решать сходные вычислительные задачи.

3. Разработка библиотеки линейной алгебры на GPGPU

Разработка библиотеки *Cubool* [9], предоставляющей примитивы линейной булевой алгебры для работы с разреженными данными, осуществляется в рамках исследовательского проекта лаборатории языковых инструментов JetBrains Research.

3.1. 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 работы [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 для реализации алгоритмов [3, 7] представлены ниже:

- Создание матрицы M размера $m \times n$
- Удаление матрицы M и освобождение занятых ею ресурсов
- Заполнение матрицы M списком значений (i, j), где i и j обозначают строку и столбец ненулевого значения
- Чтение из матрицы M списка значений (i, j), где i и j обозначают строку и столбец ненулевого значения
- Сложение матриц $M + N$
- Умножение матриц $M * N$

- Произведение Кронекера для двух матриц $M \otimes N$

3.2. Описание реализации

Архитектура разработанной библиотеки представлена на рис. 1. В качестве языка программирования для написания реализации библиотеки выбран C++, так как он предоставляет механизмы для ручного управления ресурсами, а также позволяет работать с CUDA исполняемым кодом. Исходный код компилируется в разделяемую библиотеку **libcubool.so**, которая может быть динамически загружена позже в конечное пользовательское приложение. В качестве целевой платформы ля исполнения поддерживается семейство операционных систем на базе ядра Linux.

3.2.1. Ядро библиотеки

Интерфейс библиотеки написан на языке C, что дает возможность использовать данную библиотеку в C компилируемых приложениях или экспортировать ее функции в среды с управляемыми ресурсами через механизмы исполнения внешнего кода.

Реализация библиотеки представлена классом **Instance**, который поддерживает глобальное состояние и в момент работы предоставляет контекст выполнения для всех операций библиотеки. Класс **CsrMatrix** представляет разреженную матрицу булевых значений в формате CSR. Данный класс реализует интерфейс **MatrixBase** и предоставляет доступ ко всем ранее перечисленным операциям.

Пакет **Kernels** предоставляет доступ к операциям линейной алгебры, написанным на языке CUDA C++. В качестве основы для реализации операций умножения и сложения матриц использовалась библиотека **Nsparse** [8], которая была доработана, чтобы добавить возможность динамически конфигурировать механизмы использования видеопамяти. Для реализации произведения Кронекера использовались примитивы библиотеки **Thrust**, которая позволяет оперировать данными

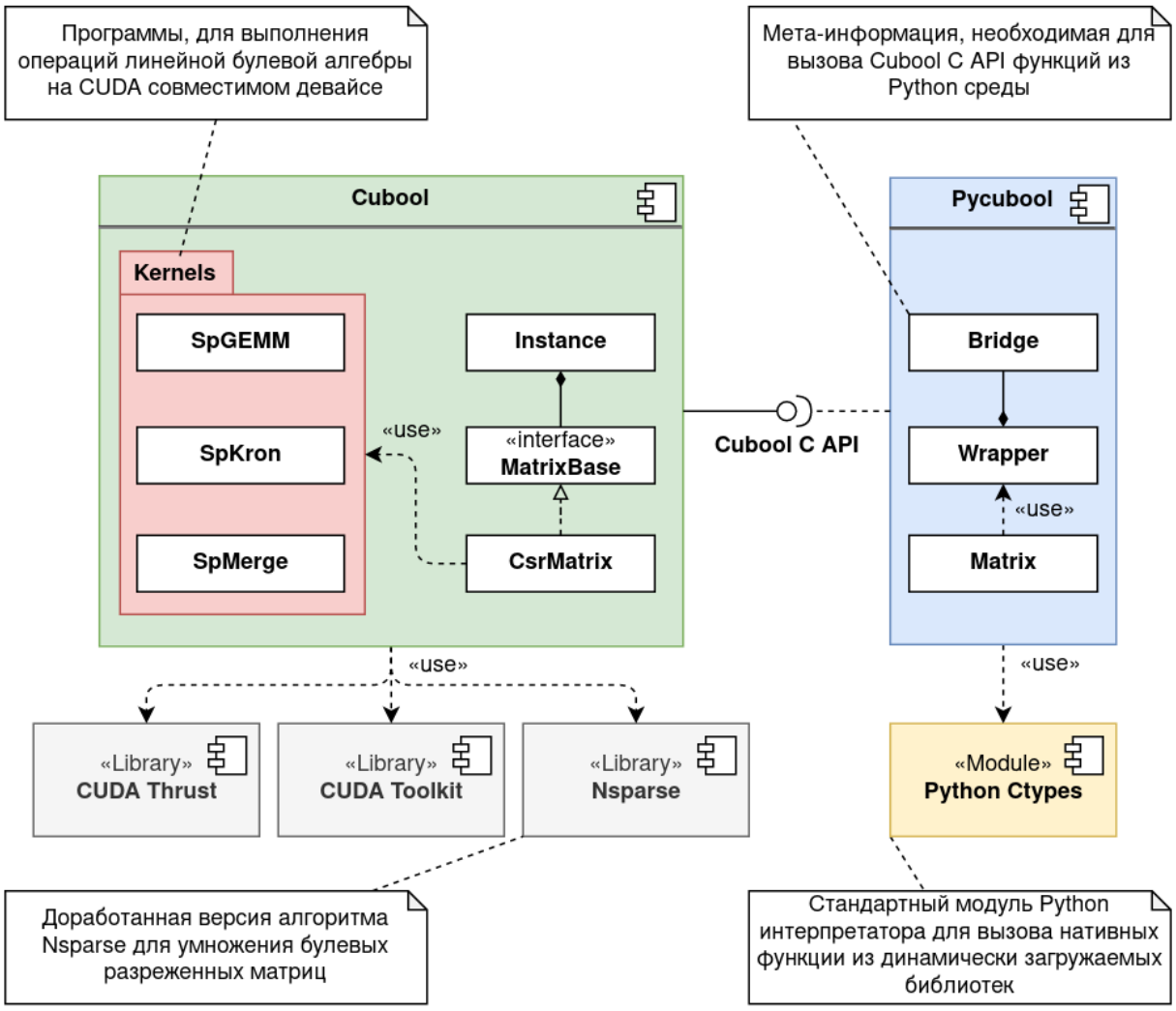

Рис. 1: Архитектура разработанной библиотеки

в терминах высокоуровневых операций *свертки*, *отображения* и *префиксной суммы* [20], которые выполняются на графическом процессоре.

3.2.2. Python модуль

Для работы с примитивами библиотеки на языке Python был разработан модуль **Pycubool**. Данный инструмент позволит более широкому кругу программистов работать с библиотекой, так как инфраструктура языка Python предоставляет широкий набор утилит для обработки данных, а также поддерживает механизмы автоматического управления ресурсами. Кроме это, использование библиотеки в Python среде позволит переиспользовать существующее решение [30] для работы с

графовыми данными, которое было получено при реализации алгоритмов [7, 8].

В качестве инструмента для вызова нативных методов, находящихся в скомпилированной библиотеке libcubool.so, используется модуль **Ctypes**, так как он поставляется вместе с инфраструктурой Python и не требует настройки сторонних зависимостей.

Класс **Bridge** хранит мета-информацию о методах и примитивах, импортируемых из нативной библиотеки. Класс **Wrapper** поддерживает глобальное состояние библиотеки и является точкой входа при импортировании модуля **Pycubool** в исполняемую среду. Класс **Matrix** предоставляет операции для работы с разреженными матрицами.

3.3. Пример использования

В качестве примера рассмотрим проблему вычисления *транзитивного замыкания* (англ. transitive closure) для некоторого ориентированного графа без меток $\mathcal{G} = \langle V, E \rangle$. Результатом вычисления транзитивного замыкания является новый в 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 глобальное состояние библиотеки, матрицу смежности исходного графа, а также указатель на идентификатор, который необходимо использовать при сохранении результирующей матрицы смежности графа после транзитивного замыкания.

В листинге 3 представлен похожий фрагмент кода, однако он уже решает поставленную в задачу на языке Python. Здесь в качестве входного аргумента используется матрица смежности графа, в качестве результата возвращается матрица смежности графа после транзитивного замыкания. Передача состояния библиотеки здесь не требуется, так как он неявно передается во все вызовы нативных методов.

Listing 2 Пример вычисления транзитивного замыкания с использованием Cubool C API

```
CuBoolStatus TransitiveClosure(CuBoolInstance Inst, CuBoolMatrix A, CuBoolMatrix* T) {
    CuBool_Matrix_Duplicate(Inst, A, T);         /** Копируем матрицу смежности A */

    CuBoolSize_t total = 0;
    CuBoolSize_t current;
    CuBool_Matrix_Nvals(Inst, *T, &current);     /** Количество ненулевых значений */

    while (current != total) {                   /** Пока результат меняется  */
        total = current;
        CuBool_MxM(Inst, *T, *T, *T);            /** T += T * T */
        CuBool_Matrix_Nvals(Inst, *T, &current);
    }

    return CUBOOL_STATUS_SUCCESS;
}
```

Listing 3 Пример вычисления транзитивного замыкания с использованием Pycubool

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

    while total != t.nvals:          # Пока результат меняется
        total = t.nvals
        pycubool.mxm(t, t, t)        # t += t * t

    return t
```

4. Текущий прогресс

Прогресс в работе на данный момент:

- Выбран технологический стек для реализации библиотеки матричных примитивов: C/C++ для реализации интерфейса библиотеки и ее функциональности, CMake для сборки проекта, NVIDIA CUDA Toolkit 10 для написания кода, исполняемого на CUDA-совместимой видеокарте, NVIDIA Thrust для автоматизации работы с ресурсами GPU.

- Создан репозиторий проекта [9], настроена автоматическая сборка с использованием инструментария *Github Actions*. Добавлено описание проекта и инструкция для сборки.

- Создан C совместимый интерфейс для работы с примитивами библиотеки, а также добавлена непосредственно реализация интерфейса: создание и удаление матриц, запись и чтение значений матрицы, операции умножения, сложения, произведение Кронекера.

- Добавлен набор *unit*-тестов для проверки корректности работы операций на основе сравнения с эталонной реализации тестируемых операций на центральном процессоре.

- На языке Python с использованием библиотеки Ctypes реализован базовый уровень абстракции, необходимый для использования функции библиотеки матричных операций в тестовой инфраструктуре, которая также реализована на языке Python.

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] CUSP: A C++ Templated Sparse Matrix Library // Github. — 2020. — Access mode: https://github.com/cusplibrary/cusplibrary (online; accessed: 09.12.2020).

[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] CuBool: Linear Boolean algebra primitives written in NVIDIA

CUDA // Github. — 2020. — Access mode: https://github.com/JetBrains-Research/cuBool (online; accessed: 08.12.2020).

[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] Dr. Timothy Alden Davis. SuiteSparse: a suite of sparse matrix software // SuiteSparse website. — 2020. — Access mode: https://people.engr.tamu.edu/davis/suitesparse.html (online; accessed: 08.12.2020).

[13] 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.

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

[15] 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.

[16] GraphBLAS Graph Linear Algebra API // graphblas. — 2020. — Access mode: https://graphblas.github.io/ (online; accessed: 09.12.2020).

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

[18] 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.

[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 Developer 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/cusparse/index.html (online; accessed: 09.12.2020).

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

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

[25] Quantifying variances in comparative RNA secondary structure prediction / 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] The Khronos Working Group. OpenGL 4.4 Specification // Khronos Registry. — 2014. — Access mode: https://www.khronos.org/registry/OpenGL/specs/gl/glspec44.core.pdf (online; accessed: 08.12.2020).

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

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

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