

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б08-мм

Улучшение производительности алгоритма достижимости с регулярными ограничениями на графическом ускорителе

Козенко Дмитрий Сергеевич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Библиотека cuBool	8
2.2. Поэлементное сложение разреженных булевых матриц .	9
3. Анализ производительности и улучшения алгоритма	10
3.1. Организация репозитории cuBoolGraph	12
4. Эксперимент	13
4.1. Условия эксперимента	13
4.2. Анализ набора данных Wikidata	14
4.3. Анализ датасета grqbench	16
4.4. Анализ масштабируемости алгоритма	19
5. Заключение	21
Список литературы	22

Введение

В последние годы графовые базы данных набирают все большую популярность: они используются для анализа социальных сетей [7], в создании баз знаний [9], в медицине [4]. Хотя они и имеют ряд ограничений и недостатков, во многих задачах они показывают себя намного лучше, чем реляционные базы данных [6]. В них данные представляются как граф с метками на вершинах (которые представляют объекты) и на ребрах (которые представляют отношения между объектами). Одна из естественных задач, возникающая при работе с такими базами данных, — нахождение всех объектов, связанными определенными отношениями с заданным. Для решения ее надо искать в графе пути с определенными ограничениями. Если рассматривать всевозможные метки на ребрах как алфавит, а путь в графе как слова, то ограничения можно проклассифицировать формальными языками [2]. Будем рассматривать регулярные языки над этим алфавитом и использовать регулярные ограничения, а точнее их расширение — двухсторонние регулярные ограничения, позволяющие ходить по ребрам в обратном направлении. Запросы с двухсторонними регулярными ограничениями были реализованы в языке запросов к данным SPARQL, и добавлены в ISO стандарт языков запросов к графам в ISO 39075:2024¹.

В реализациях алгоритмов для решения этой задачи активно используются представление графов в виде разреженных матриц смежности, например в графовых базах данных MilleniumDB [5] и FalcorDB². Так же есть алгоритмы, представленные в работах [1] и [10], основанные на операциях линейной алгебры.

Операции линейной алгебры отлично ложатся на огромную вычислительную мощность видеокарт, и хочется использовать этот потенциал для улучшения времени работы алгоритма.

В предыдущей работе [11] был реализован алгоритм [10] для вычислений на GPU с помощью библиотеки cuBool [12]. Тестирование на

¹Стандарт ISO/IEC 39075:2024: <https://www.iso.org/standard/76120.html> (Дата доступа: 19.06.2025)

²Документация FalcorDB: <https://docs.falkordb.com/> (Дата доступа: 19.06.2025)

наборе данных Wikidatda [8] показало, что реализация на GPU³ отработала в 4 раза хуже реализации на CPU⁴.

В данной работе будут изучены причины, по которым реализация на GPU отставала по производительности от CPU и предложены изменения реализации на GPU, улучшающие ее производительность.

³[Код](#) реализации алгоритма на GPU, представленный в предыдущей работе (Дата доступа: 19.06.2025)

⁴[Код](#) реализации алгоритма на CPU (Дата доступа: 19.06.2025)

1 Постановка задачи

Целью работы является анализ результатов тестов производительности и изучение причин отставания реализации алгоритма достижимости с регулярными ограничениями на GPU от его реализации на CPU с дальнейшим изменениями, улучшающие производительность.

Для достижения поставленной выше цели необходимо выполнить следующие задачи.

1. Воспроизвести тестирование из прошлой работы, проанализировать результаты и найти узкие места алгоритма.
2. Улучшить производительность алгоритма.
3. Провести эксперименты с итоговой версией алгоритма.
4. Перенести свою ветку `update-cuda-12.6` в основную репозиторий `cuBool`, организовать репозиторий с алгоритмами, использующими `cuBool`, и добавить в нее свой алгоритм

2 Обзор

В предыдущей работе [11] используется алгоритм 1 достижимости в графе с регулярными ограничениями, его реализация на языке программирования C++, основанная на библиотеке cuBool [12], представлена в GitHub репозитории⁵.

Algorithm 1 Алгоритм достижимости в графе с регулярными ограничениями на основе поиска в ширину, выраженного с помощью операций матричного умножения

```

1: procedure BFSBASEDRPQ( $D = \langle Q, q_S, Q_F, \delta, \Sigma \rangle, G = \langle V, E, L \rangle, V_{src}$ )
2:    $\mathcal{D}^a \leftarrow$  Матрица смежности ДКА для символа  $a$ 
3:    $\mathcal{G}^a \leftarrow$  Матрица смежности графа для символа  $a$ 
4:    $\mathcal{F} \leftarrow$  Вектор финальных состояний автомата  $D$ 
5:    $M \leftarrow$  Матрица  $|Q| \times |V|$  ▷ Матрица обхода
6:    $P \leftarrow$  Матрица  $|Q| \times |V|$  ▷ Матрица с посещёнными состояниями
7:    $M' \leftarrow$  Матрица  $|Q| \times |V|$  с вершинами из  $V_{src}$ , соответствующими
    $q_S$ 
8:   while  $M'$  ненулевая do
9:      $M \leftarrow M'$ 
10:     $M' \leftarrow$  Пустая матрица
11:    for all  $a \in (\Sigma \cap L)$  do
12:      if  $a$  начинается не с  $\wedge$  then
13:         $M' \leftarrow M' + (\mathcal{D}^a)^T \times (M \times \langle \neg P \rangle \mathcal{G}^a)$  ▷ Умножение с
        маской
14:      else
15:         $M' \leftarrow M' + (\mathcal{D}^a)^T \times (M \times \langle \neg P \rangle (\mathcal{G}^a)^T)$ 
16:      end if
17:    end for
18:     $P \leftarrow P + M'$  ▷ Аккумуляция достигнутых состояний
19:     $\mathcal{P} \leftarrow \mathcal{P} + \mathcal{F} \times M'$  ▷ Поиск финальных состояний
20:  end while
21:  return  $\mathcal{P}$ 
22: end procedure

```

Список входных параметров перечислен ниже.

- Конечный автомат, построенный по регулярному ограничению

⁵Код реализации алгоритма на GPU, представленный в предыдущей работе (Дата доступа: 19.06.2025)

(существуют алгоритмы преобразования), состоящий из графа переходов Q , начальных (q_S) и конечных (Q_F) состояний, а так же алфавита δ и языка Σ на котором построено регулярное ограничение.

- Граф, состоящий из множества вершин V , множества ребер с метками E и множества допустимых меток L .
- Начальные вершины графа — V_{src} .

Граф декомпозируются по меткам (для каждой метки выделяется граф, состоящий из тех же вершин, что и начальный граф, но из ребер начального графа берутся только ребра с этой меткой), и создаётся множество матриц смежности для каждого получившегося графа $\mathcal{D}^a$. Такая же операция применяется и к графу конечного автомата $\mathcal{G}$.

Далее создаются матрицы, которые будут обновляться каждый шаг алгоритма и хранить текущее состояние: все достижимые вершины на данном шаге алгоритма. Это матрицы обхода M и M' , одна пустая (так как на первом же шаге цикла в нее будет записано другое значение), а вторая заполняется следующим образом: для каждого начального состояния конечного автомата все начальные вершины графа помечаются достижимыми. Так же создается матрица всех посещенных состояний P , чтобы в дальнейшем избежать лишних итераций цикла алгоритма.

Основная часть алгоритма — это обход поиском в ширину, который происходит с использованием умножения матриц. Он продолжается, пока матрица обхода с предыдущего шага M' ненулевая. Каждый шаг матрица обхода с предыдущего шага сохраняется, матрица обхода текущего шага заполняется нулями.

На каждом шагу обхода алгоритм проходит по всем исходящим из вершин, достигнутых на предыдущем шаге, ребрам для каждой метки, одновременно обходится и конечный автомат. Это происходит следующим образом: к матрице обхода состояния M' для каждой метки a аккумулируются состояния, получающиеся из вершин прошлого состояния, которые еще не были посещены (это достигается при помощи умножения на маску посещенных состояний P), проходя по ребру, если

оно есть и в графе (умножение на $\mathcal{G}^a$), и в графе конечного автомата (умножаем на $\mathcal{D}^a$ на результат прохода по ребру графа). Если по ребру требуется пройти в обратную сторону, то используется транспонированную матрицу графа $(\mathcal{G}^a)^T$ (доказательство корректности операции можно найти в работе в пункте 3.1).

После этого мы обновляем матрицу посещенных состояний P и вектор достижимых вершин $\mathcal{P}$ (дописываем в него, в какой вершине мы достигли какого-то финального состояния конечного автомата).

Результатом алгоритма является вектор достижимых вершин $\mathcal{P}$.

2.1 Библиотека cuBool

В этом разделе будет более подробно рассмотрена библиотека cuBool, ее возможности и недостатки.

В cuBool представлены почти все операции булевой линейной алгебры для векторов и матриц. Для матриц это транспонирование, сложение, умножение, как обычное, так и поэлементное, операция взятия столбца или строки как вектор, сложение всех строк, а также произведение Кронекера. Что очень важно, есть операция получения количества ненулевых элементов, и при этом она работает за $O(1)$. Для векторов же это сложение и умножение как вектора на матрицу, так и матрицы на вектор.

Все матрицы в cuBool разреженные, и используется следующий формат хранения: для каждой матрицы хранится множество пар индексов с ненулевыми значениями.

Для реализации данных операций данная библиотека использует вычислительное API CUDA⁶ от Nvidia, а точнее говоря библиотеку thrust, написанную на CUDA самими разработчиками. Полная и доступная документация thrust⁷ делает разработку на ней быстрой и приятной, а код становится простым для восприятия.

В cuBool есть поддержка операций, выполняемых на CPU без ис-

⁶API CUDA: <https://developer.nvidia.com/cuda-toolkit> (Дата доступа: 19.06.2025)

⁷Документация thrust: <https://nvidia.github.io/cccl/thrust/> (Дата доступа: 19.06.2025)

пользования параллельных вычислений, что, конечно, не дает желаемой производительности, но позволяет запускаться на платформах, на которых нет графического ускорителя.

Также в `cuBool` есть удобная система логгирования, сильно облегчающая отладку, а также слежение за выделенной памятью, что помогает избежать её утечек.

2.2 Поэлементное сложение разреженных булевых матриц

В ходе изучения производительности алгоритма было обнаружено, что операция сложения матриц занимает много времени, поэтому необходимо изучить, как можно реализовать эту операцию.

Пусть $C = A + B$. Рассмотрим пример сложения булевых матриц.

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Для каждого элемента итоговой матрицы легко понять, какое значение у него будет. На примере видно, что элемент матрицы равен единице, если хотя бы у одной из складываемых матриц соответствующий элемент тоже равен единице.

$$C_{ij} = 1 \iff A_{ij} = 1 \vee B_{ij} = 1$$

Один из оптимальных по памяти форматов представления разреженных матриц это множество пар индексов, в которых элементы отличны от 0. И чтобы вычислить результат поэлементного сложения надо узнать все пары индексов, которые есть хотя бы в одной из исходных матриц, то есть нужно объединить множества индексов с ненулевыми элементами матриц.

3 Анализ производительности и улучшения алгоритма

В начале работы был воспроизведен эксперимент из предыдущей работы [11] — проведены замеры производительности реализаций на GPU и CPU на наборе данных Wikidata⁸ [8]. Время выполнения всех первых 520 запросов следующее: для CPU — 98.3 секунды, для GPU — 273.3 секунды.

Далее к реализации на GPU был применён анализатор производительности Intel Vtune⁹, был составлен FlameGraph. Распределение времени работы по функциям предоставлено в таблице 1.

Таблица 1: Распределение времени работы внутри алгоритма

Функция cuBool	Время работы, сек	Часть в %
cuBool_Matrix_EWiseAdd	164.2	60.1%
cuBool_MxM	70.7	25.9%
cuBool_Matrix_Transpose	30.5	11.2%
cuBool_Matrix_EWiseMulInverted	6.0	2.2%
cuBool_Matrix_Free	1.7	0.6%

То, что функция сложения матриц занимает больше всего времени, очень необычно. Если посмотреть на псевдокод алгоритма 1, то можно заметить, что операция сложения применяется не чаще операции поэлементного умножения на инвертированную матрицу, а асимптотически операции очень похожи, это видно из сравнения пункта 2.2 и пункта 3.3 из работы [11]. Но функция сложения занимает в разы больше времени относительно функции поэлементного умножения на инвертированную матрицу.

Была изучена реализация функции сложения матриц¹⁰ в cuBool. Было обнаружено, что она использует библиотеку nsparse [3]. Для проверки гипотезы, что эта реализация не дает достаточной производи-

⁸База знаний WIKIDATA: https://www.wikidata.org/wiki/Wikidata:Database_download (Дата доступа: 19.06.2025)

⁹Анализатор vtune: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html> (Дата доступа: 19.06.2025)

¹⁰Код предыдущей реализации сложения матриц в cuBool (Дата доступа: 19.06.2025)

Листинг 1.: Ошибочное выставление флагов компиляции

```
target_compile_options(cubool PRIVATE $<$<AND:$<CONFIG:Debug>,  
                                     $<COMPILE_LANGUAGE:CXX>>: -O2>)  
target_compile_options(cubool PRIVATE $<$<AND:$<CONFIG:Release>,  
                                     $<COMPILE_LANGUAGE:CXX>>: -O0>)
```

сти, функция сложения была реализована аналогично функции умножения на инвертированную матрицу. Для сохранения старой функциональности была добавлена опция для выбора реализации функции сложения — CUBOOL_USE_NSPARSE_MERGE_FUNCTOR.

При применении vtune для версии с новой функцией сложения матриц, видно значительное ускорение в 2.34 раза, теперь время выполнения всех запросов составляет 116.6 секунд, а время работы самой функции сложения матриц сократилось с 164.2 секунд до 10.5, то есть ускорение в 15.6 раз.

Так же была обнаружена ошибка в файле CmakeLists.txt библиотеки cuBool (Листинг 1) — перепутаны флаги оптимизаций для компилятора в конфигурациях Debug и Release. В Debug используются оптимизации компилятора, а в Release — нет. Было убрано явное выставление флагов, так как хорошей практикой при написании CmakeLists считается не выставлять явно флаги компиляции, если нет необходимости.

Это изменение не повлияло на время выполнения запросов (так как эти опции не относятся к ядру CUDA, в которых происходят сами вычисления), зато сильно повлияло на время загрузки запросов: функция преобразования матриц в формат CSR cuBool_Matrix_Build стала работать в 8.3 раз быстрее — вместо 5686.4 секунд 686.3 секунд.

Так же после анализа времени работы алгоритма на CPU было обнаружено, что транспонирование матриц не занимает времени, в то время как в реализации для GPU оно занимало около 25%. Было замечено, что в бенчмарке для CPU транспонированные матрицы уже передаются в алгоритм. После этого была добавлена такая возможность и в бенчмарк для реализации для GPU.

3.1 Организация репозитории cuBoolGraph

Была организована GitHub репозиторий cuBoolGraph¹¹ в ветке `add-rpq`. Была выбрана следующая структура файлов (Листинг 2).

Листинг 2.: Структура файлов cuBoolGraph

```
cuBoolGraph
├── deps
│   ├── cuBool
│   ├── fast_matrix_market
│   └── googletest
├── include
├── src
├── tests
│   └── test_data
```

В директории `src` находятся исходные файлы алгоритмов, в `include` — их заголовочные файлы. Также есть директория с тестами `tests`. В директории `deps` лежат внешние зависимости. Это, естественно, библиотека `cuBool`, а также зависимости для тестирования — `googletests`¹² и `fast_matrix_market`¹³ для загрузки матриц в формате `matrix market`. Все зависимости присоединяются к библиотеке при помощи `git submodule`. Также был настроен CI, в котором запускаются тесты.

¹¹Репозиторий cuBoolGraph: <https://github.com/SparseLinearAlgebra/cuBoolGraph> (Дата доступа: 19.06.2025)

¹²GitHub репозиторий googletests: <https://github.com/google/googletest> (Дата доступа: 19.06.2025)

¹³GitHub репозиторий fast_matrix_market: https://github.com/alugowski/fast_matrix_market (Дата доступа: 19.06.2025)

4 Эксперимент

Для дальнейшего анализа производительности алгоритма были поставлены следующие задачи.

- Замерить время работы реализаций на CPU и GPU и проанализировать типы запросов, работающих на GPU медленнее, чем на CPU.
- Изучить масштабируемость реализации для GPU, сравнив время работы на двух разных по мощности тестовых стендах.

4.1 Условия эксперимента

Эксперименты проводились на тестовом стенде со следующей конфигурацией.

- **CPU:** Intel i5-10400f, 6 физических ядер, 12 потоков, максимальная частота 4.3 GHz
- **GPU:** Nvidia RTX 3050, 8 Gb VRAM, 1777 MHz
- **RAM:** 16 Gb, 3200 MHz, DDR4
- **OS:** Manjaro 6.14
- **Компиляторы:** nvcc 12.8, gcc 15.1

Измерялось только время работы алгоритма, загрузка входных данных в оперативную память и видеопамять не учитывалась.

Код бенчмарка находится в репозитории GitHub¹⁴.

В качестве данных была выбрана база знаний WIKIDATA¹⁵ [8], на которой ставился эксперимент в предыдущей работе [11] и набор данных RPQ-bench¹⁶.

¹⁴Код бенчмарка для реализации на GPU: <https://github.com/mitya-y/rpq> (Дата доступа: 19.06.2025)

¹⁵База знаний WIKIDATA: https://www.wikidata.org/wiki/Wikidata:Database_download (Дата доступа: 19.06.2025)

¹⁶Набор данных RPQ-bench (Дата доступа: 19.06.2025)

Данные	Время GPU, s	Время CPU, s	Ускорение GPU
Wikidata	115.73	136.30	1.18
RPQ-bench	185.28	55.19	0.30

Таблица 2: Результаты сравнения CPU и GPU

4.2 Анализ набора данных Wikidata

Было проведено 15 измерений. Погрешность не учитывалась, так как стандартное отклонение не превышает 3%. Результаты измерения суммарного времени исполнения всех запросов представлены в таблице 2. Видно, что на Wikidata реализация на GPU выигрывает в производительности на 18%, но проигрывает более чем в 3 раза на новом наборе данных RPQ-bench.

Для визуализации результатов было составлено облако точек (Рис. 1) с временем выполнения запросов на CPU и GPU. Видно, что среднее время выполнения запроса на GPU меньше, но медиана выше, так как на относительно маленьких запросах реализация на GPU, кроме самих вычислений, тратит значительную часть времени на синхронизацию.

Так же есть отдельные запросы, на которых алгоритм на GPU отработывает намного хуже. Ниже представлена таблица 3 с четырьмя запросами, на которых самый значительный проигрыш.

Таблица 3: 4 запроса, на которых GPU сильно проигрывает

№ запроса	Тип запроса	Замедление	Время GPU, s	Время CPU, s
132	$(a^* b^*)^+$	1.58	11.28	7.16
443	a^*	2.27	2.61	1.15
207	$(a \mid b)^+$	2.52	17.34	6.88
67	$(\hat{a} a)^+$	43.05	9.04	0.21

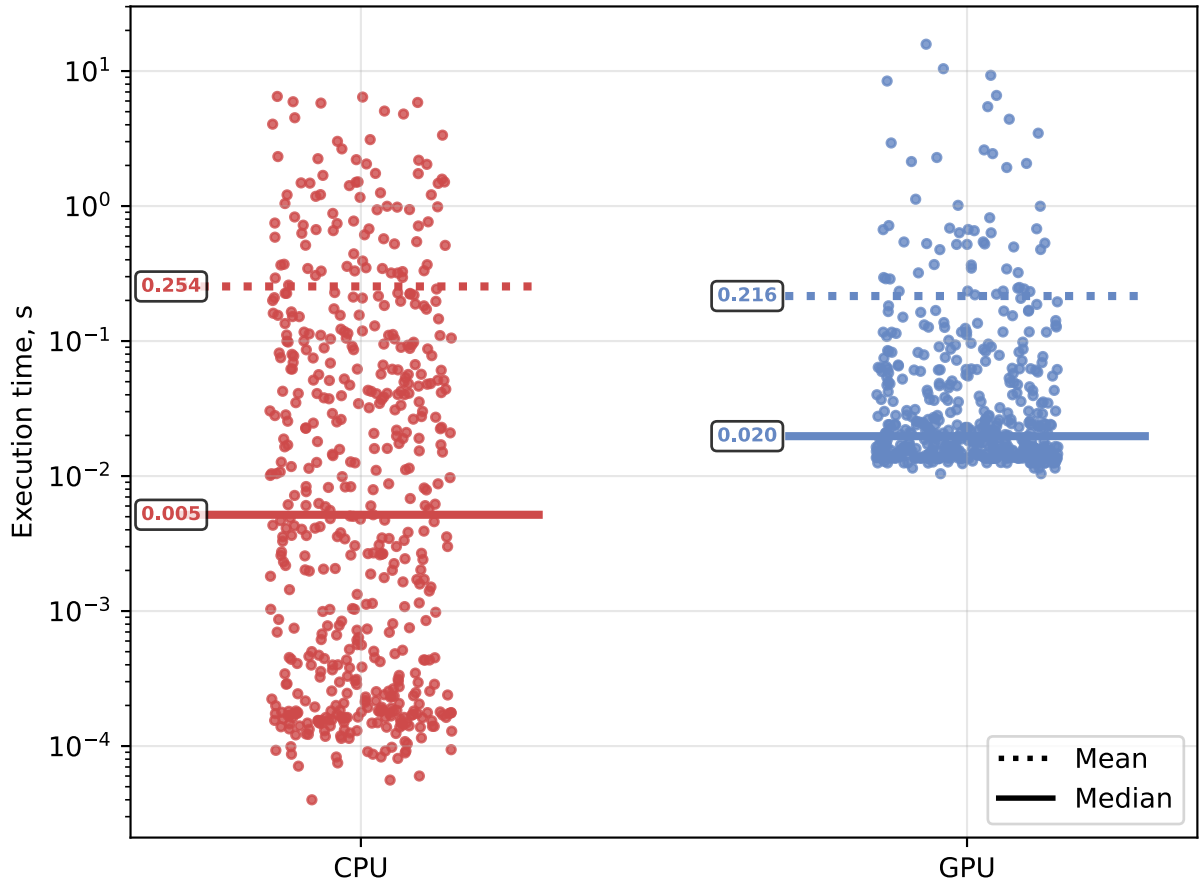


Рис. 1: Облако точек с временем выполнения запросов на GPU и CPU

Для трех запросов из таблицы 3 была обнаружена следующая закономерность — у них очень большое количество ответов относительно соседних запросов, это можно заметить на Рис. 2. Для 207 и 443 запроса видно, что у них количество ответов сильно превышает количество ответов у их соседей. Рядом же с 132-ым запросом есть 124-ый, у которого количество ответов даже больше, а работает он быстрее — 6.60 секунд против 10.41 и вызывает замедление всего на 1.13, хотя все равно входит в пятерку самых долгих запросов. Для сравнения двух этих запросов было проанализировано количество итераций цикла — 23 итерации у 124-го против 40 итераций у 132-го. Вероятно, столько времени 132-ой запрос тратит именно в дополнительных относительно 124-ого запроса итерациях.

Для 67-го запроса эта закономерность не наблюдается. Был исследован тип этого запроса — он представляется в виде $(\hat{a} a)^+$, где $\hat{a}$ обозна-

чает разворот ребра. Но есть и другие запросы такого типа, например с 656 по 659, но в них значительного замедления не наблюдается.

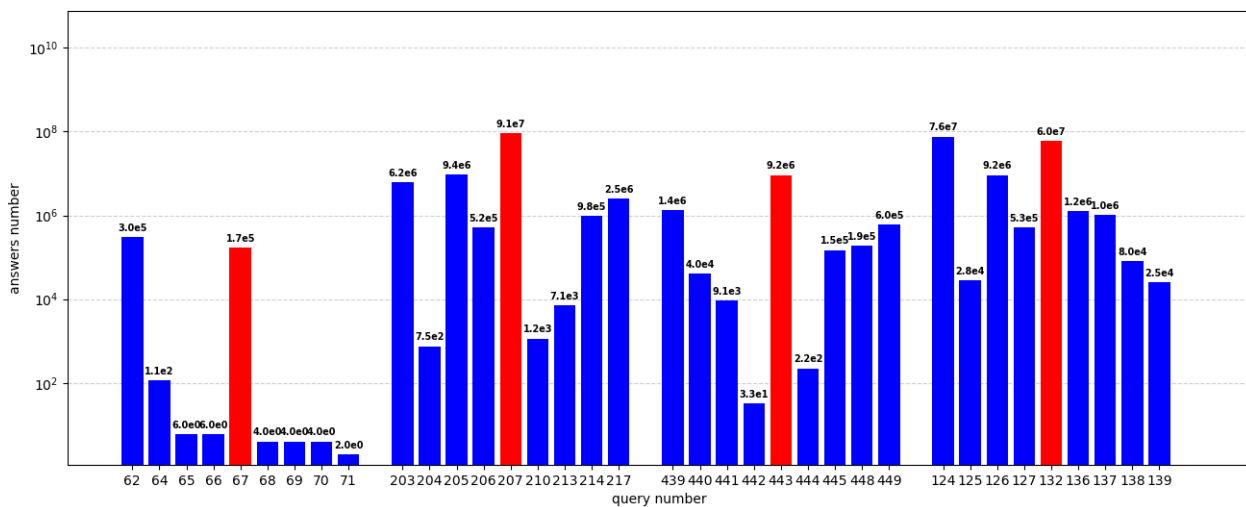


Рис. 2: Столбчатая диаграмма, показывающая количество ответов у худших по ускорению и ближайших к ним запросов. Худшие запросы выделены красным цветом.

4.3 Анализ датасета `rpqbench`

Для дальнейшего тестирования алгоритма был использован набор данных `RPQ-bench`¹⁷.

Был создан набор данных на 10^7 вершин и протестировано 20 типов запросов, для каждого типа было создано по 1000 запросов, то есть общее количество — 20000.

Полное время работы на CPU составило 55 секунд, а на GPU — 185.

Были проанализированы типы запросов: для каждого типа посчитано общее время выполнения, результаты представлены в таблице 4. Можно заметить, что все типы запросов работают на GPU дольше, чем на CPU. Но соотношение 0.1 секунды на CPU и 3 секунды на GPU можно объяснить тем, что запросы этих типов достаточно малы, и их не стоит запускать на GPU, так как расходы на синхронизацию сильно превышают вычислительные расходы. Интересно поведение GPU на типах запросов 11, 12 и 15, так как они значительно превышают остальные по времени работы.

¹⁷Набор данных `RPQ-bench` (Дата доступа: 19.06.2025)

Таблица 4: Типы запросов grqbench

№	Шаблон	Стартовая	Конечная	GPU, s	CPU, s	Замедление
1	abc	Любая	Фикс.	3.61	0.10	34.40
2	abc	Фикс.	Любая	2.45	0.065	37.82
3	$(abc) \mid (cdd)$	Фикс.	Любая	4.31	0.13	33.30
4	$(abc) \mid (cdd)$	Любая	Фикс.	5.64	0.15	37.41
5	d^*	Фикс.	Любая	2.68	0.072	37.15
6	d^*	Любая	Фикс.	2.39	0.061	39.07
7	d^*e	Фикс.	Любая	4.14	0.12	35.73
8	d^*e	Любая	Фикс.	2.13	0.052	41.32
9	d^+	Фикс.	Любая	2.61	0.07	39.19
10	d^+	Любая	Фикс.	2.32	0.06	38.36
11	$(ab)^*$	Фикс.	Любая	30.60	0.19	164.18
12	$(ab)^*$	Любая	Фикс.	10.59	0.98	10.86
13	$fg(d \mid e)$	Фикс.	Любая	38.54	16.49	2.34
14	$fg(d \mid e)$	Любая	Фикс.	3.31	0.09	37.41
15	$fg(d \mid e)^*$	Фикс.	Любая	45.92	35.34	1.30
16	$fg(d \mid e)^*$	Любая	Фикс.	7.11	0.18	38.75
17	$(c \mid g)(d \mid e)$	Фикс.	Любая	4.57	0.33	13.76
18	$(c \mid g)(d \mid e)$	Любая	Фикс.	3.63	0.10	37.67
19	$(c \mid g)(d \mid e)^*$	Фикс.	Любая	5.15	0.52	9.85
20	$(c \mid g)(d \mid e)^*$	Любая	Фикс.	3.58	0.09	38.35

На 15-ом типе CPU тоже работает долго, и в итоге замедление не очень значительное, его изучение с целью выявления слабых мест реализации на GPU не очень интересно. 11-ый и 12-ый тип запросов имеют одинаковый шаблон, различается только направление обхода графа — в 12-ом типе все ребра графа развернуты в обратную сторону. Шаблон простой — 2 ребра подряд под звездочкой Клини, и появилась гипотеза, что проблема в большом количестве итераций цикла поиска в глубину на стороне CPU, которая требует постоянной синхронизации с GPU. Для всех запросов было посчитано количество итераций и составлен поточечный график (Рис. 3), который подтвердил эту гипотезу.

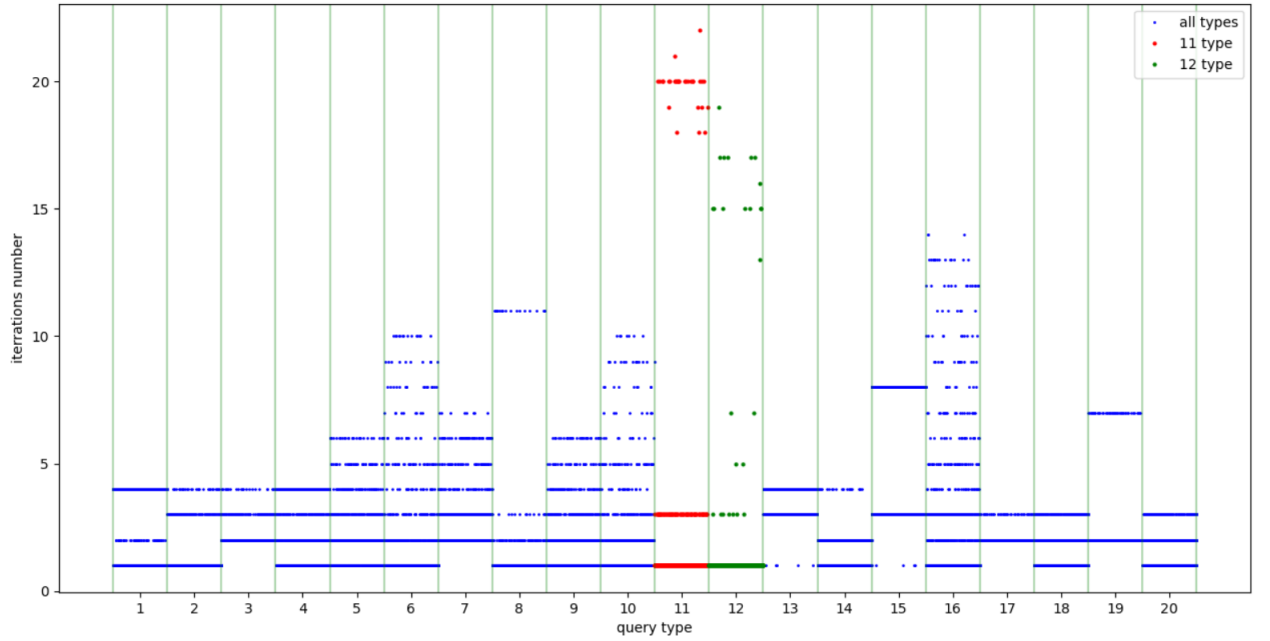


Рис. 3: Количество итераций в каждом типе запроса

Анализ производительности в vtune показал (результаты представлены в таблице 5), что время сложения и поэлементного умножения на инвертированную матрицу в реализации на GPU меньше, чем время этих операций на CPU (в LAGraph обе эти операции производятся через функцию `GRB_assign`), время извлечения финальных вершин (функции умножения вектора на матрицу) больше в 3 раза, а вот время исполнения функции умножения в cuBool — 178 секунд против 20 секунд в LAGraph — почти в 8 раз больше. Можно сделать вывод, что для реализации на GPU узким местом является умножение матриц, то есть библиотека `nsparse` [3]. Со сложением матриц она уже показала, что не является оптимальным решением, возможно и произведение матриц тоже реализовано не оптимально.

Таблица 5: Время работы каждой операции на наборе данных grqbench в реализациях на GPU и CPU

Операция	GPU, s	CPU, s
MxM	148.96 (80.11%)	20.37 (37.13%)
VxM	21.19 (11.49%)	8.08 (14.73%)
cuBool_EwiseAdd	6.55 (3.52%)	—
cuBool_EwiseMulInverted	9.23 (4.96%)	—
GrB_Matrix_assign	—	26.22 (47.79%)

4.4 Анализ масштабируемости алгоритма

Для дальнейшего анализа реализации на GPU были проведены измерения с целью сравнения с предыдущими результатами на новом тестовом стенде со следующей конфигурацией.

- **CPU:** AMD Ryzen 9 7900X, 12 физических ядер, 24 потоков, максимальная частота 5.733 GHz
- **GPU:** Nvidia RTX 3090, 24 Gb VRAM, 1692 MHz
- **RAM:** 128 Gb, 3600 MHz, DDR5
- **OS:** Ubuntu 24.04 LTS
- **Компиляторы:** nvcc 12.0, gcc 13.3

Теоретическая мощность RTX 3090 почти в 4 раза больше, чем у RTX 3050 — 35.6 TFLOS против 9.2 TFLOPS, пропускная способность так же больше в 4 раза — 936 Гб/с против 224 Гб/с. Фактическая производительность, оценённая бенчмарком Passmark¹⁸, приблизительно в 2 раза больше. Характеристики видеокарт взяты с сайта techpowerup^{19,20}.

Значит, если алгоритм хорошо масштабируем, то на RTX 3090 он должен работать хотя бы в 2 раза быстрее относительно на RTX 3050.

Так же, учитывая мощность RTX 3090, был создан набор данных grqbench на 10^8 вершин.

Результаты замеров представлены в таблице 6.

¹⁸[Результаты](#) бенчмарка Passmark для RTX 3050 и RTX 3090 (Дата доступа: 19.06.2025)

¹⁹[Характеристики](#) RTX 3050 (Дата доступа: 19.06.2025)

²⁰[Характеристики](#) RTX 3090 (Дата доступа: 19.06.2025)

Таблица 6: Результаты замеров на RTX 3050 и RTX 3090

Набор данных	RTX 3050, s	RTX 3090, s	Ускорение
wikidata	116.50	100.99	1.15
rpqbench 10^7	185.28	131.46	1.41
rpqbench 10^8	318.86	300.17	1.06

Интересно, что самое незначительное ускорение наблюдается на самом крупном наборе данных, хотя ожидалось обратное поведение.

При помощи vtune было проанализировано, какую часть работы исполнения всех запросов на каждом наборе данных составляют функции cuBool, результаты представлены в таблице 7. Можно заметить, что самую значительную часть составляет операция умножения. Именно в ней заключается проблема масштабирования — она составляет большую часть времени, порядка 90%, и работает на RTX 3090 примерно столько же, сколько и на RTX 3050, на остальных же функциях можно заметить значительное ускорение работы.

Таблица 7: Время работы каждой операции на RTX 3050 и RTX 3090

Wikidata			
Операция	RTX 3050, s	RTX 3090, s	Ускорение
MxM	92.99 (76.42%)	80.27 (79.99%)	1.16
VxM	6.33 (5.20%)	1.67 (1.66%)	3.79
EwiseAdd	16.90 (13.89%)	12.88 (12.84%)	1.31
EwiseMulInverted	10.13 (8.32%)	5.53 (5.51%)	1.83
rpqbench на 10^7 вершин			
Операция	RTX 3050, s	RTX 3090, s	Ускорение
MxM	148.96 (80.11%)	112.09 (84.55%)	1.33
VxM	21.19 (11.49%)	8.58 (6.47%)	2.47
EwiseAdd	6.55 (3.52%)	4.86 (3.67%)	1.35
EwiseMulInverted	9.23 (4.96%)	7.04 (5.31%)	1.31
rpqbench на 10^8 вершин			
Операция	RTX 3050, s	RTX 3090, s	Ускорение
MxM	274.71 (85.00%)	269.33 (90.27%)	1.02
VxM	22.65 (7.00%)	11.22 (3.76%)	2.02
EwiseAdd	15.28 (4.73%)	11.67 (3.91%)	1.31
EwiseMulInverted	10.57 (4.27%)	6.15 (2.06%)	1.72

5 Заключение

В ходе работы были достигнуты следующие результаты.

- Были найдены узкие места в алгоритме и исправлены ошибки, вследствие чего производительность выросла.
- Было проведено сравнение финальной версии реализации на GPU с реализацией на CPU, показавшее, что на крупных запросах GPU может дать выигрыш, а на маленьких предпочтительнее использовать CPU.
- Было проведено исследование масштабируемости алгоритма, который выявил узкое место — операция умножения матриц.
- Изменения cuBool были перенесены в основной репозиторий, а исходный код в cuBoolGraph²¹

²¹GitHub репозиторий cuBoolGraph: <https://github.com/SparseLinearAlgebra/cuBoolGraph> (Дата доступа: 19.06.2025)

Список литературы

- [1] Arroyuelo Diego, Gómez-Brandón Adrián, Navarro Gonzalo. Evaluating Regular Path Queries on Compressed Adjacency Matrices. — 2024. — 2307.14930.
- [2] Barrett Chris, Jacob Riko, Marathe Madhav. Formal-Language-Constrained Path Problems // [SIAM J. Comput.](#) — 2000. — 01. — Vol. 30. — P. 809–837.
- [3] [Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication](#) / Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, Semyon Grigorev // Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA'20. — New York, NY, USA : Association for Computing Machinery, 2020. — 12 p. — URL: <https://doi.org/10.1145/3398682.3399163>.
- [4] Hall RJ Murray CW Verdonk ML. The Fragment Network: A Chemistry Recommendation Engine Built Using a Graph Database. — URL: <https://pubmed.ncbi.nlm.nih.gov/28712298/> (дата обращения: 9 ноября 2024 г.).
- [5] Vrgoc Domagoj, Rojas Carlos, Angles Renzo et al. MillenniumDB: A Persistent, Open-Source, Graph Database. — 2021. — [2111.01540](#).
- [6] Pokorný Jaroslav. [Graph Databases: Their Power and Limitations](#). — Vol. 9339. — 2015. — 09. — P. 58–69.
- [7] Time-varying Social Networks in a Graph Database: A Neo4J Use Case, in First International Workshop on Graph Data Management Experiences and Systems GRADES '13 (ACM, New York / C. Cattuto, Marco Quaggitto, Andre Panisson, A. Averbuch // USA. — 2013. — 06. — Vol. 1.

- [8] Vrandečić Denny, Krötzsch Markus. Wikidata: a free collaborative knowledgebase // *Commun. ACM*. — 2014. — sep. — Vol. 57, no. 10. — P. 78–85. — URL: <https://doi.org/10.1145/2629489>.
- [9] Zhang Zuopeng Justin. Graph Databases for Knowledge Management // *IT Professional*. — 2017. — Vol. 19, no. 6. — P. 26–32.
- [10] Белянин Георгий Олегович. Оптимизация алгоритма решения задачи достижимости с ограничениями в виде регулярных языков. — URL: https://se.math.spbu.ru/thesis/texts/Beljanin_Georgij_Olegovich_Spring_practice_2nd_year_2024_text.pdf (дата обращения: 9 ноября 2024 г.).
- [11] Козенко Дмитрий Сергеевич. Реализация алгоритма достижимости с регулярными ограничениями на графическом ускорителе.
- [12] Орачев Егор Станиславович. Linear Boolean algebra library primitives and operations for work with sparse matrices written on the NVIDIA CUDA platform. — URL: https://se.math.spbu.ru/thesis/texts/Orachev_Egor_Stanislavovich_Bachelor_Thesis_2020_text.pdf (дата обращения: 9 ноября 2024 г.).