

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б08-мм

# Реализация 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

<b>Введение</b>	<b>3</b>
<b>1. Постановка задачи</b>	<b>5</b>
<b>2. Обзор</b>	<b>6</b>
2.1. Выбор библиотеки булевой линейной алгебры для GPU .	9
2.2. Библиотека cuBool . . . . .	10
2.3. Поэлементное умножение разреженных булевых матриц	11
<b>3. Обновление библиотеки cuBool и реализация алгоритма</b>	<b>13</b>
3.1. Отладка некорректной работы функциональности cuBool	14
3.2. Исправление ошибки в сложении матриц . . . . .	16
3.3. Реализация поэлементного умножения на инвертированную матрицу . . . . .	17
3.4. Реализация алгоритма на базе библиотеки cuBool . . . .	18
<b>4. Эксперимент</b>	<b>19</b>
4.1. Условия эксперимента . . . . .	19
<b>5. Заключение</b>	<b>23</b>
5.1. Дальнейшее направление исследований . . . . .	23
<b>Список литературы</b>	<b>24</b>

# Введение

В последние годы графовые базы данных набирают все большую популярность: они используются в алгоритмах для социальных сетей [9], в создании баз знаний [11], в медицине [5]. Хотя они и имеют ряд ограничений и недостатков, во многих задачах они показывают себя намного лучше, чем реляционные базы данных [7]. 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 39075:2024<sup>1</sup>.

В реализациях алгоритмов для решения этой задачи активно используются представление графов в виде разреженных матриц смежности, например в графовых базах данных MilleniumDB [6] и FalcorDB<sup>2</sup>. Также есть алгоритмы, представленные в работах [1] и [12], 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. Есть различные библиотеки, позволяющие работать с булевой линейной алгеброй на GPU:

---

<sup>1</sup>Стандарт ISO/IEC 39075:2024: <https://www.iso.org/standard/76120.html> (Дата посещения: 09.11.2024)

<sup>2</sup>Документация FalcorDB: <https://docs.falkordb.com/> (Дата посещения: 08.01.2025)

Spla [8], Cusp<sup>3</sup>, cuBool [13]. Библиотека cuBool [13] в тестах<sup>4</sup> производительности показывает себя лучше всего, но она не поддерживает последнюю версию вычислительного API CUDA и в ней нет некоторой функциональности, необходимой для реализации алгоритма.

В данной работе будет представлена адаптация алгоритма достижения с регулярными ограничениями для GPGPU вычислений и его реализация на вычислительном API CUDA для графических ускорителей.

---

<sup>3</sup>Репозитория Cusp на GitHub: <https://github.com/cusplibrary/cusplibrary> (Дата посещения: 08.01.2025)

<sup>4</sup>Тесты производительности cuBool: <https://github.com/SparseLinearAlgebra/cuBool?tab=readme-ov-file#performance> (Дата посещения: 08.01.2025)

# 1. Постановка задачи

Целью работы является адаптация алгоритма достижимости с регулярными ограничениями для GPGPU вычислений и его перенос на видеокарту с помощью библиотеки cuBool [13]. Для достижения поставленной выше цели необходимо выполнить следующие задачи.

1. Обновить библиотеку cuBool для поддержки последней версии CUDA.
2. Реализовать в cuBool недостающие для алгоритма операции с булевыми матрицами.
3. Реализовать алгоритм, используя обновленную библиотеку.
4. Провести эксперименты: найти узкие места в алгоритме и уменьшить среднее время исполнения, сравнить время работы реализации алгоритма на GPU с его реализацией на CPU и проанализировать результаты.

## 2. Обзор

У алгоритма достижимости с регулярными ограничениями есть довольно много реализаций, из популярных можно назвать реализации в больших графовых базах данных с открытым кодом MilleniumDB<sup>5</sup> и FalcorDB<sup>6</sup>.

Но для реализации на GPU был выбран алгоритм, предложенный Георгием Беляниным в работе [12]. Реализация этого алгоритма на CPU от автора работы представлена в GitHub репозитории<sup>7</sup>. Она реализованна на базе библиотеки линейной алгебры для разреженных матриц SuiteSparse [4] в форке репозитория LaGraph, где собраны алгоритмы на базе этой библиотеки.

Причины, по которым была выбрана именно она, перечислены ниже.

- Она основана на операциях булевой линейной алгебры и использует большие разреженные матрицы, а вычисления этих операций отлично подходят для огромной параллельной вычислительной мощности видеокарт.
- Есть поддержка двухсторонних запросов.
- В библиотеке SuiteParse ведется разработка поддержки вычислений на GPU, так что потенциально именно эта реализация может стать главным конкурентом по производительности.
- В работе [3] есть тесты производительности, которые говорят о том, что алгоритм работает быстрее своих конкурентов.

В работе можно найти следующий псевдокод алгоритма (1).

---

<sup>5</sup>Реализация RPQ в MilleniumDB: [https://github.com/MillenniumDB/MillenniumDB/tree/main/src/query/executor/binding\\_iter/paths/all\\_shortest\\_simple](https://github.com/MillenniumDB/MillenniumDB/tree/main/src/query/executor/binding_iter/paths/all_shortest_simple) (Дата посещения: 08.01.2025)

<sup>6</sup>Реализация RPQ в FalcorDB: [https://github.com/FalcorDB/FalcorDB/blob/master/src/algorithms/all\\_shortest\\_paths.c](https://github.com/FalcorDB/FalcorDB/blob/master/src/algorithms/all_shortest_paths.c) (Дата посещения: 08.01.2025)

<sup>7</sup>Реализация алгоритма RPQ: <https://github.com/SparseLinearAlgebra/LAGraph/tree/2-rpq> (Дата посещения: 08.01.2025)

---

**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 (существуют алгоритмы преобразования), состоящий из графа переходов  $Q$ , начальных ( $q_S$ ) и конечных ( $Q_F$ ) состояний, а так же алфавита  $\delta$  и языка  $\Sigma$  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 финальных состояний  $\mathcal{P}$  (дописываем в него, в какой вершине мы достигли какого-то финального состояния конечного автомата).



Результатом алгоритма является вектор финальных состояний  $\mathcal{P}$ .

## 2.1. Выбор библиотеки булевой линейной алгебры для GPU

Проанализировав алгоритм, можно составить список необходимых операций, который должна поддерживать искомая библиотека булевой линейной алгебры для GPU. Все перечисленные ниже операции подразумевают умножение и сложение над булевым полукольцом.

*Инвертированной матрицей* для матрицы  $A$  называется матрица такого же размера, у которой на месте 1 у матрицы  $A$  стоит 0, а на месте 0 — 1.

1. Перемножение булевых матриц.
2. Возможность транспонировать и сохранить матрицу.
3. Поэлементное умножение на инвертированную матрицу.
4. Поэлементное сложение матриц.
5. Операция получения количества ненулевых элементов.
6. Умножение вектора на матрицу.

Также очень важным фактором является формат хранения матриц: ожидается, что алгоритм должен уметь работать с графами, в которых порядка  $10^8$  вершин, и если хранить матрицу в виде двумерного массива, то это займет невозможное в нынешнее время количество памяти. При этом в алгоритме матрицы преимущественно разреженные, так что рассматривались библиотеки, использующие оптимальные структуры данных для хранения разреженных матриц.

Были найдены следующие библиотеки, подходящие по данному критерию: Spla [8], Cusp<sup>8</sup>, cuBool [13]. Сравнивая их по скорости работы и

---

<sup>8</sup>Репозитория Cusp на GitHub: <https://github.com/cusplibrary/cusplibrary> (Дата посещения: 08.01.2025)

легкости интеграции и отладки, была выбрана библиотека `cuBool` по следующим причинам.

- Скорость работы<sup>9</sup>. Это является ключевым требованием, так как хочется добиться максимальной производительности и обрабатывать графы из реального мира.
- Её легковесность и сравнительная простота: читабельность кода и возможность быстро разобраться в нем чрезвычайно важны при разработке, так как для достижения максимальной производительности придется отлаживать даже функции внутри самой библиотеки, и необходимо разбираться в том, как они работают. Также библиотека почти не использует сложные API для вычислений на видеокарте, вместо это в ней используется библиотека `thrust`<sup>10</sup>, которая хорошо задокументирована.
- Возможность обновлять ее и дописывать свой код.

## 2.2. 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.

Но при наличии большого набора операций нет одной, необходимой для алгоритма: поэлементного умножения на инвертированную матрицу.

---

<sup>9</sup>Тесты производительности `cuBool`: <https://github.com/SparseLinearAlgebra/cuBool?tab=readme-ov-file#performance> (Дата посещения: 08.01.2025)

<sup>10</sup>Библиотека `thrust`: <https://developer.nvidia.com/thrust> (Дата посещения: 08.01.2025)

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 CUDA<sup>11</sup> от Nvidia, а точнее говоря библиотеку thrust, написанную на CUDA самими разработчиками. Полная и доступная документация thrust<sup>12</sup> делает разработку на ней быстрой и приятной, а код становится легко читабельным.

В cuBool есть поддержка операций, выполняемых на CPU без использования 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.

Еще одной проблемой стало отсутствие поддержки в последние годы: последний раз cuBool был собран компилятором gcc версии 8 и протестирован на CUDA версии 10.1 (версия датируется 28 июля 2019 года). При тестировании на компьютере с последней версией CUDA 12.6 (от 14 августа 2024 года) и gcc 14.2 возникло большое количество ошибок компиляции.

## 2.3. Поэлементное умножение разреженных булевых матриц

Так как в cuBool не хватает реализации умножения на инвертированную матрицу, необходимо изучить, как реализовать эту операцию, учитывая формат хранения разреженных матриц в cuBool.

Дан простой пример поэлементного умножения матриц.

---

<sup>11</sup>API CUDA: <https://developer.nvidia.com/cuda-toolkit> (Дата посещения: 08.01.2025)

<sup>12</sup>Документация thrust: <https://nvidia.github.io/cccl/thrust/> (Дата посещения: 08.01.2025)

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

Пусть  $C = A * B$ . Для каждого элемента итоговой матрицы довольно легко понять, какое значение у него будет. Он равен единице тогда и только тогда, когда у каждой перемножаемой матрицы соответствующие элементы тоже равны единицам.

$$C_{ij} = 1 \iff \begin{cases} A_{ij} = 1 \\ B_{ij} = 1 \end{cases}$$

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

Теперь легко можно вывести правило для поэлементного умножения на инвертированную матрицу. Все единицу у матрицы  $B$  переходят в нули, а нули – в единицу.

$$C_{ij} = 1 \iff \begin{cases} A_{ij} = 1 \\ B_{ij} = 0 \end{cases} \iff \begin{cases} A_{ij} = 1 \\ B_{ij} \neq 1 \end{cases}$$

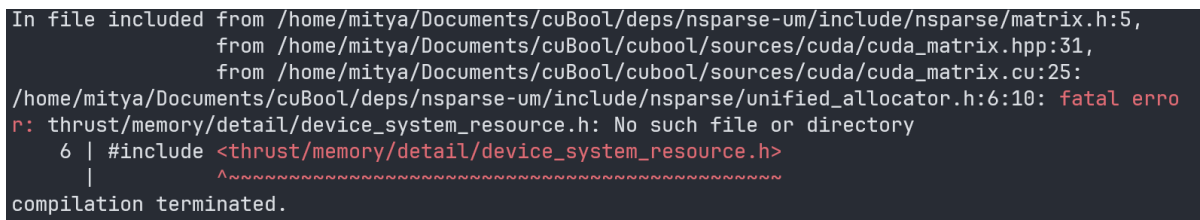
Для получения итоговой матрицы нам нужно узнать все индексы, в которых у элемента первой матрицы значение единица, а у элемента второй матрицы значение 0. Для получения этого множества индексов надо вычесть из множества индексов с ненулевыми элементами первой матрицы множество индексов с ненулевыми элементами второй матрицы.

### 3. Обновление библиотеки cuBool и реализация алгоритма

При первой попытке скомпилировать cuBool (использовались указанные в репозитории команды<sup>13</sup> для сборки) возникло большое количество ошибок компиляции.

Была ошибка компиляции связанная с отсутствием включения заголовочного файла стандартной библиотеки C++ limits, появившаяся вследствие обновления версии компилятора, следственно и стандартной библиотеки, и скорее всего в gсс-8 он включался в другой заголовочный файл и был доступен без явного его подключения.

Остальные же ошибки были связаны с исходными файлами CUDA, собирающимися компилятором nvcc. Многие были связаны с отсутствием включения необходимых заголовочных файлов библиотеки thrust, видимо когда-то необходимые функции включались через другие заголовочные файлы. Были ошибки, связанные и с тем, что некоторые файлы перестали существовать, так как они были в detail части thrust (Рис. 1).



```
In file included from /home/mitya/Documents/cuBool/deps/nsparse-um/include/nsparse/matrix.h:5,
                 from /home/mitya/Documents/cuBool/cubool/sources/cuda/cuda_matrix.hpp:31,
                 from /home/mitya/Documents/cuBool/cubool/sources/cuda/cuda_matrix.cu:25:
/home/mitya/Documents/cuBool/deps/nsparse-um/include/nsparse/unified_allocator.h:6:10: fatal error
r: thrust/memory/detail/device_system_resource.h: No such file or directory
 6 | #include <thrust/memory/detail/device_system_resource.h>
    |           ^~~~~~
compilation terminated.
```

Рис. 1: Ошибки компиляции из-за отсутствия файлов из detail

Были же ошибки намного более сложные, они выглядели следующим образом (Рис. 2). Вывод компилятора ничего не говорит о причинах ошибок, он только указывает на отсутствие некоторых синтаксических конструкций, таких как точка с запятой и фигурные скобки. Возможно, это связано с тем, что после стадии препроцессинга получился синтаксически некорректный код, и компилятор не может его распознать как код на C++.

---

<sup>13</sup>Команды сборки cuBool: <https://github.com/SparseLinearAlgebra/cuBool?tab=readme-ov-file#get-the-source-code-and-run> (Дата посещения: 08.01.2025)

```

15 /usr/local/cuda-12.6/bin/../../targets/x86_64-linux/include/cub/detail/device_synchronize.cuh(37):
error: expected a ";"
16     namespace detail
17     ^
18
19 /usr/local/cuda-12.6/bin/../../targets/x86_64-linux/include/cub/detail/device_synchronize.cuh(44):
error: this pragma must immediately precede a declaration
20     #pragma nv_exec_check_disable
21     ^
22
23 /usr/local/cuda-12.6/bin/../../targets/x86_64-linux/include/cub/detail/device_synchronize.cuh(70):
error: expected a declaration
24     }
25     ^
26
27 /usr/include/c++/13/istream(86): error: expected a declaration
28     public:
29     ^

```

Рис. 2: Ошибки компиляции исходных файлов CUDA

Можно заметить, что эти ошибки компиляции возникают не в исходных файлах самой библиотеки, а в части Cuda Development Toolkit — Cub, как можно увидеть на Рис. 2, эти файлы находятся в системе и устанавливаются пакетным менеджером вместе с CUDA. После более подробного изучения cuBool, было обнаружено, что внутри нее тоже есть cub, хотя в последних версиях CUDA он устанавливается вместе с ней. Тогда было удалено включение локального cub в проект, и это помогло скомпилировать библиотеку.

### 3.1. Отладка некорректной работы функциональности cuBool

После компиляции встала задача починить тесты, так как прошло только 101 из 139 тестов. Причины падения тестов были непонятны с первого взгляда, при этом результат некоторых менялся от запуска к запуску, и иногда они даже проходили. Во время отладки было обнаружено, что не работает должным образом функция `thrust::set_intersection`. При пересечении двух массивов, в которых есть одинаковые элементы, она в ответе выдавала пустой массив. Для отладки был создан отдельный проект с простым кодом, использующий эту функцию (Листинг 1).

Этот код работал абсолютно корректно, как от него и ожидалось: в

**Листинг 1: Код для отладки `thrust::set_intersection`. Здесь `print_vector` — простая вспомогательная функция, печатающая элементы вектора**

```
void test() {
    thrust::device_vector<uint32_t> v {2, 7};
    thrust::device_vector<uint32_t> u {1, 2, 3, 4, 5, 6, 7, 8, 9};
    thrust::device_vector<uint32_t> w(2);

    auto end = thrust::set_intersection(v.begin(), v.end(),
                                       u.begin(), u.end(),
                                       w.begin());
    auto size = thrust::distance(w.begin(), end);

    print_vector(v);
    print_vector(u);
    std::cout << "size = " << size << std::endl;
    print_vector(w);
}
```

в лежали элементы 2 и 7. Но когда этот код был вставлен в `siBool` и вызван в одном из тестов, в ответе опять был пустой массив. После того, как один и тот же код работает по-разному в разных проектах осталось изучать только окружение программы и систему сборки. Окружение было одинаковым, все запускалось с одной машины с одной корневой директорией. Осталось только смотреть на опции сборки.

Более глубокое и подробное изучение `CMake` у `siBool` дало результат. Была найдена опция (Листинг 2), без добавления которой код стал работать корректно и все тесты успешно прошли. Эта опция отвечает за параллельную компиляцию<sup>14</sup> исходных файлов `CUDA`. Поиск причины ошибки в интернете не дал результатов, так что планируются дальнейшее самостоятельное изучение.

Было проверено как изменилось время компиляции библиотеки после удаления опции. Компилятором для `C++` файлов был выбран `gsc` версии 14.2, для файлов `CUDA` — `nvcc` версии 12.6.68. Компиляторам была передана опция `-j1` для теста времени работы в однопоточной ком-

---

<sup>14</sup>Документация `cmake`: [https://cmake.org/cmake/help/latest/prop\\_tgt/CUDA\\_SEPARABLE\\_COMPILATION.html](https://cmake.org/cmake/help/latest/prop_tgt/CUDA_SEPARABLE_COMPILATION.html) (Дата посещения: 08.01.2025)

## Листинг 2: Опция CMake, из-за которой thrust работал некорректно

```
set_target_properties(cubool PROPERTIES CUDA_SEPARABLE_COMPILATION ON)
```

## Листинг 3: Команды для замера времени компиляции

```
cmake -DCMAKE_BUILD_TYPE=Release -DCUBOOL_BUILD_TESTS=ON -B build -S .  
time cmake --build build -j1
```

пиляции. Использовались команды компиляции, указанные в листинге 3. Для замера времени использовалась команда Unix `time`, итоговое время считалось как сумма работы `user` и `system` частей.

Время компиляции без опции составило 113.53, с опцией 115.33. Разница находится в пределах погрешности, итоговое время компиляции осталось таким же.

## 3.2. Исправление ошибки в сложении матриц

В ходе разработки алгоритма была обнаружена ошибка в сложении матриц (это можно заметить на упрощенном коде сложения матриц из библиотеки, представленным в листинге 4). Переменная `cols` должна быть проинициализирована как `a.m_cols`, а не `a.m_rows`, так как при сложении матриц одинакового размера (это условие уже проверенно выше) размер итоговой матрицы должен быть таким же, как и у исходных матриц.

Эта ошибка была допущена вследствие того, что в тестах `cuBool` не хватает проверки размерности итоговой матрицы, так что в дальнейшем стоит добавить эту проверку в каждый тест.



#### Листинг 4: Функция сложения, в которой была обнаружена ошибка

```
class SpMergeFunctor {
public:
    // type definitions
    MatrixType operator()(const MatrixType& a, const MatrixType& b) {
        assert(a.m_rows == b.m_rows);
        assert(a.m_cols == b.m_cols);
        IndexType rows = a.m_rows;
        IndexType cols = a.m_cols;
        // addition GPU operations
        return MatrixType(std::move(col_result), std::move(rpt_result),
                           rows, cols, vals);
    }
    // class fields
};
```

### 3.3. Реализация поэлементного умножения на инвертированную матрицу

В cuBool все матрицы, в силу своей разреженности, хранятся как множество пар индексов. Для поиска пересечений множества индексов используется функция `thrust::set_intersection`. Тогда для поиска разности множеств индексов отлично подходит функция `thrust::set_difference`.

Для поддержки операции поэлементного умножения на инвертированную матрицу в cuBool были реализованы следующие функции, классы и файлы.

- C-API интерфейсная функция `cuBool_Matrix_EWiseMulInverted`.
- Файл `cuBool_Matrix_EWiseMultInverted.cpp` с реализацией C-API функции.
- Абстрактный виртуальный метод `eWiseMultInverted` в классе `MatrixBase`.
- Реализация метода `eWiseMultInverted` в наследнике `MatrixBase` `Matrix`.

- CUDA файл `cuda_matrix_ewisemult_inverted.cu` с реализацией метода `eWiseMultInverted` в наследнике `MatrixBase CudaMatrix`.
- CUDA kernel `spewisemultinverted.cuh` с классом `SpVectorEwiseMultInverted`, в котором реализованы код вычислений на GPU.
- Тесты для C-API интерфейсной функции.

Все изменения, которые были сделаны в `cuBool` можно найти в форке GitHub репозитории<sup>15</sup> на ветке `update-cuda-12.6`.

### 3.4. Реализация алгоритма на базе библиотеки `cuBool`

С помощью обновленной версии `cuBool` алгоритм достижимости с регулярными ограничениями был реализован на GPU. Реализация опирается на работу [12] и реализацию алгоритма в `LaGraph`<sup>16</sup>.

Были написаны тесты для проверки корректности работы алгоритма. В качестве входных данных был взят пример из работы [12] и данные, на котором тестировалась реализация алгоритма на CPU.

Так же был настроен CI в репозитории алгоритма на GitHub в котором происходит компиляция и запуск тестов, в которых используются непараллельные версии функций `cuBool` для CPU. Также применяется анализатор стиля кода при загрузке изменений.

Результаты работы можно найти в репозитории GitHub<sup>17</sup>.

---

<sup>15</sup>Форк репозитории `cuBool`: <https://github.com/mitya-y/cuBool/> (Дата посещения: 08.01.2025)

<sup>16</sup>Репозиторий `LaGraph`: <https://github.com/SparseLinearAlgebra/LAGraph/tree/2-rpq> (Дата посещения: 08.01.2025)

<sup>17</sup>Репозитория с алгоритмом: <https://github.com/mitya-y/rpq> (Дата посещения: 08.01.2025)

## 4. Эксперимент

После реализации алгоритма было необходимо протестировать его работоспособность на данных из реального мира и сравнить его производительность с реализацией на CPU, чтобы понять, что дал перенос алгоритма для вычислений на видеокарте.

В качестве данных была выбрана база знаний WIKIDATA<sup>18</sup> [10], на которой тестировалась производительность реализации на CPU в работе [12], так как это один из самых разнообразных и больших графов доступных публично и вместе с ним доступны наборы реальных регулярных запросов. Код<sup>19</sup> для теста производительности выложен автором алгоритма на GitHub.

### 4.1. Условия эксперимента

Эксперименты проводились на тестовом стенде со следующей конфигурацией.

- **CPU:** Intel i5-10400f, 6 физических, 12 логических ядер, максимальная частота 4.3 GHz
- **GPU:** Nvidia RTX 3050, 8 Gb VRAM, 1552 MHz
- **RAM:** 16 Gb, 3200 MHz, DDR4
- **OS:** Ubuntu 24.04
- **Компиляторы:** nvcc 12.6, gcc 14.2

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

Тестирование проводилось на 520 запросах, среди которых встречаются двусторонние. Размеры графа порядка  $10^8$ , что позволяет протестировать алгоритм на графе большого размера.

---

<sup>18</sup>База знаний WIKIDATA: [https://www.wikidata.org/wiki/Wikidata:Database\\_download](https://www.wikidata.org/wiki/Wikidata:Database_download) (Дата доступа 11.12.24)

<sup>19</sup>Код для теста производительности алгоритма: <https://github.com/georgiy-belyanin/RPQ-bench> (Дата посещения: 08.01.2025)

Погрешности измерения не указаны, так как они достаточно малы. Для каждого алгоритма все запросы были проведены по 10 раз, и стандартное отклонение только у 1 запроса для GPU превысило 1%, для измерений на CPU у 6 запросов, но ни одно стандартное отклонение не превысило 8%.

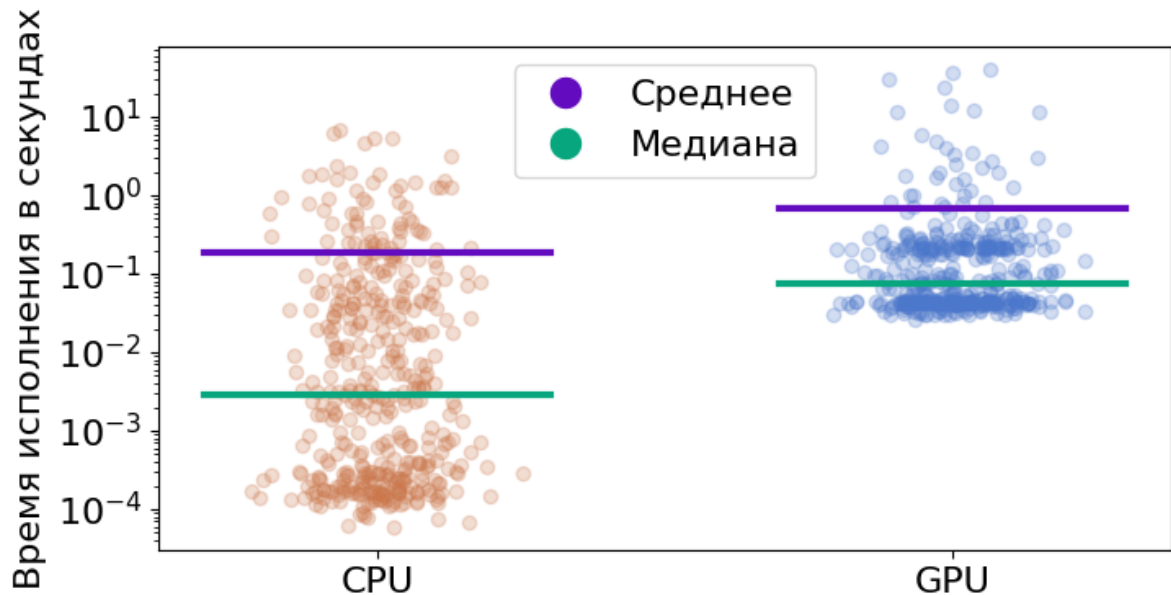


Рис. 3: Облако распределения времени исполнения запросов

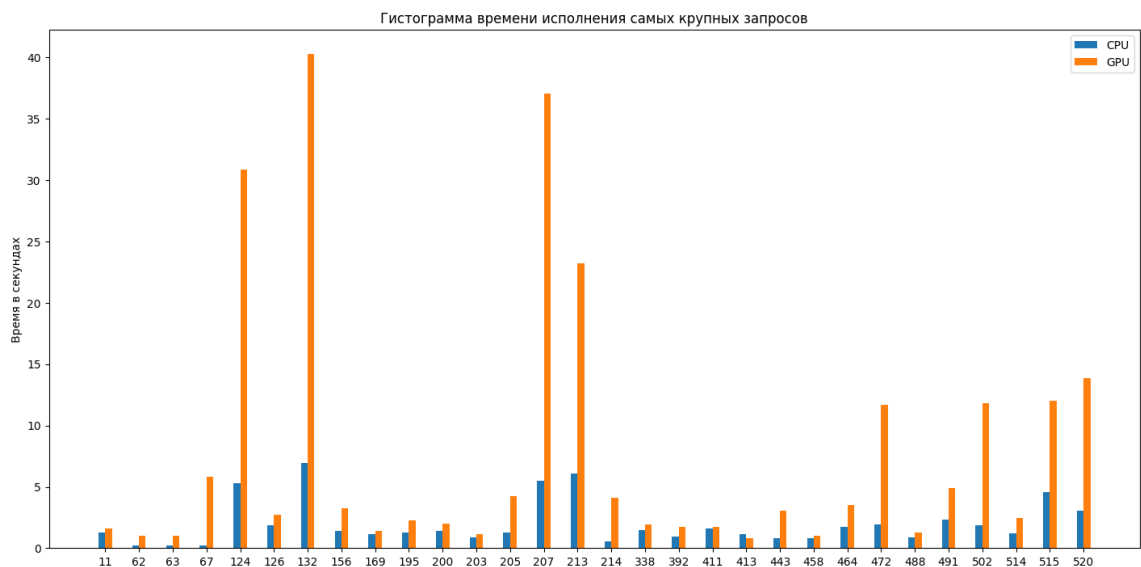


Рис. 4: Гистограмма времени исполнения самых крупных запросов (заявивших более 1 секунды)

Было построено облако распределения времени работы (Рис. 3) реализаций на CPU и GPU на разных запросах. Заметно, что даже маленькие запросы реализация на GPU исполняет с некоторой задержкой. Так же была построена гистограмма для выборки самых крупных запросов, в которых время работы одной из реализации больше 1 секунды (Рис. 4). Можно заметить, что есть отдельные запросы, на которых реализация на GPU работает намного хуже реализации на CPU. В дальнейшем следует подробнее изучить эти запросы и выявить закономерности.

Было посчитано среднее замедление, оно равняется 3.59. Этот показатель говорит о том, что текущая реализация алгоритма на GPU проигрывает реализации на CPU. Но есть и 20 запросов, на которых реализация на GPU выигрывает.

Для дальнейших исследований для каждого запроса было посчитано ускорение, взята медиана и посчитано среднее арифметическое: среднее арифметическое равняется 0.232, а медиана 0.0379. То, что медиана ускорений примерно в 8 раз меньше среднего показывает, что запросы делятся на те, на которых реализация на GPU проигрывает сильно, и те, на которых проигрыш не так заметен. Это также можно увидеть на графике (Рис. 3).

Также были проанализированы лучшие и худшие показатели ускорения: взято среднее арифметическое по 5 лучшим (Таблица 1) и 5 худшим (Таблица 2). У лучших этот показатель равняется 1.584, что говорит о том, что есть запросы, на которых реализация на GPU показывает себя значительно лучше реализации на CPU. Худший показатель же крайне мал: 0.000603. Это говорит о том, что есть отдельный тип запросов, с которыми реализация на GPU крайне плохо работает. Можно заметить, что время работы у алгоритма на CPU крайне мало, и возможно такая большая разница из-за того, что сама работа с GPU занимает всегда какое-то время, которое незначительно в более крупных запросах, но в таких малых превышает время вычислений.

Результаты эксперимента показали, что реализация алгоритма на GPU хорошо справляется с данными из реального мира, но при этом проигрывает в производительности реализации на CPU. Изучение сред-

Номер запроса	Время GPU	Время CPU	Ускорение
353	0.182	0.263	1.444
421	0.294	0.441	1.501
416	0.431	0.704	1.633
346	0.277	0.46	1.658
311	0.179	0.3	1.682

Таблица 1: 5 лучших запросов

Номер запроса	Время GPU	Время CPU	Ускорение
429	0.170	0.000081	0.000476
511	0.200	0.000111	0.000553
484	0.202	0.000127	0.000627
433	0.175	0.000118	0.000671
442	0.174	0.000119	0.000683

Таблица 2: 5 худших запросов

него замедления показало, что среднее время исполнения запроса того же порядка, что и время исполнения запроса на CPU, а на некоторых запросах реализация на GPU показала себя даже лучше, следовательно реализацию на GPU можно использовать для распределения нагрузки на графические ускорители при многочисленных запросах. Анализ типов запросов, на которых реализация на GPU показывает себя лучше реализации на CPU или наоборот сильно ей проигрывает не был проведен, он планируется в дальнейших исследованиях алгоритма.

## 5. Заключение

В ходе работы над реализацией алгоритма достижимости с регулярными ограничениями на графическом ускорителе были достигнуты следующие результаты.

- Библиотека cuBool была обновлена до последней версии CUDA 12.6.
- В cuBool реализована операция поэлементного умножения на инвертированную матрицу.
- При помощи обновленной версии cuBool алгоритм достижимости с регулярными ограничениями был реализован для запуска на GPU.
- Был проведен эксперимент, который показал, что текущая реализация на GPU проигрывает в производительности реализации на CPU: абсолютное среднее замедление равно 3.59.

Исходный код доступен в GitHub репозитории <https://github.com/mitya-y/rpq>.

### 5.1. Дальнейшее направление исследований

В дальнейшем работа над реализацией алгоритма достижимости с регулярными ограничениями на GPU может быть продолжена в следующих направлениях.

- Изучение на каких типах запросов реализация на GPU показывает себя хуже всего и лучше всего и дальнейший анализ для нахождения слабых мест алгоритма для вычислений на видеокарте.
- Профилирование и отладка производительности GPU при помощи специальных инструментов для поиска самых нагруженных частей алгоритма.

## 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] Belyanin Georgiy, Grigoriev Semyon. Single-Source Regular Path Querying in Terms of Linear Algebra. — 2024. — [2412.10287](#).
- [4] 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. — 25 p. — URL: <https://doi.org/10.1145/3322125>.
- [5] 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.).
- [6] Vrgoc Domagoj, Rojas Carlos, Angles Renzo et al. MillenniumDB: A Persistent, Open-Source, Graph Database. — 2021. — [2111.01540](#).
- [7] Pokorný Jaroslav. [Graph Databases: Their Power and Limitations](#). — Vol. 9339. — 2015. — 09. — P. 58–69.
- [8] SPbLA: The Library of GPGPU-powered Sparse Boolean Linear Algebra Operations / Egor Orachev, Maria Karpenko, Pavel Alimov, Semyon Grigorev // [Journal of Open Source Software](#). — 2022. — 08. — Vol. 7. — P. 3743.
- [9] 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 Quaggiotto, Andre Panisson, A. Averbuch // USA. — 2013. — 06. — Vol. 1.

- [10] 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>.
- [11] Zhang Zuopeng Justin. Graph Databases for Knowledge Management // [IT Professional](#). — 2017. — Vol. 19, no. 6. — P. 26–32.
- [12] 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](https://se.math.spbu.ru/thesis/texts/Beljanin_Georgij_Olegovich_Spring_practice_2nd_year_2024_text.pdf) (дата обращения: 9 ноября 2024 г.).
- [13] 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](https://se.math.spbu.ru/thesis/texts/Orachev_Egor_Stanislavovich_Bachelor_Thesis_2020_text.pdf) (дата обращения: 9 ноября 2024 г.).