Содержание

Введение

С момента первоначального создания логическое программирование было признано одной из парадигм с наибольшим потенциалом автоматизированного использования параллелизма. Ковальски в своей книге определяет распараллеливание как сильную сторону логического программирования [1]. Это стало началом интенсивного развития данной области.

За последние два десятилетия были проведены новые исследования, в которых изучалась роль новых параллельных архитектур в ускорении и развитии новых подходов логического вывода на основе графического процессора [2].

Целью проводимой работы является обзор методов и способов распараллеливания логического вывода.

1 Анализ предметной области

В данном разделе определяются основные определения, используемые при анализе, а также описывается актуальность исследуемой области.

1.1 Использование логического программирования

Математическая логика является формализацией человеческого мышления и представления знаний. В связи с целью автоматизации процесса логического вывода возникла новая парадигма — логическое программирование. В основе этой идеи лежит описание задачи совокупностью утверждений на некотором формальном логическом языке и получение решения с помощью вывода в некоторой формальной системе [3].

В широком смысле логическое программирование представляет собой семейство таких методов и решений проблем, в которых используются приемы логического вывода для манипулирования знаниями. В узком – использование исчисления предикатов первого порядка в качестве основы для описания предметной области с использованием дизъюнктов Хорна [4].

Машина логического вывода — реализованная программным или аппаратным образом высокоуровневая абстракция, выполняющая генерацию новых фактов и правил в соответствии с законами формальной логики [5]. Используемые в ней архитектурные решения определяют задачу логического вывода.

Известным языком программирования в данной парадигме является Prolog и его реализации [6]. Программа состоит из набора фактов, а также отношений, которые устанавливаются между ними. Существует некоторая база данных, в которой они хранятся вместе с задаваемыми правилами. Для начала выполнения работы программа должна иметь запрос. Он образуется из термов, каждый из которых должен иметь истинное значение.

Использование логического программирования привлекает разработчиков тем, что программы не содержат затрудняющие для понимания детали, а описывают то, что собой представляет результат решения; данная особенность позволяет проще осуществить проверку и убедиться в том, что реализуется требуемое.

1.2 Параллелизм

Ввиду того, что определенные этапы вычислений могут производиться параллельно, то должна быть заложена возможность распределения подзадач между несколькими исполнителями.

Логическое программирование предлагает некоторые возможности для неявного использования параллелизма. Выделяют два основных вида параллелизма: AND-параллелизм и OR-параллелизм [11].

Независимый OR-параллелизм основан на параллельном выполнении дизъюнкции предикатов, не имеющих общих переменных. Результат будет получен из первого выполнившегося терма. Это соответствует параллельному просмотру дерева поиска (OR-дерева). Пример OR-параллелизма представлен на рисунке 1.1.

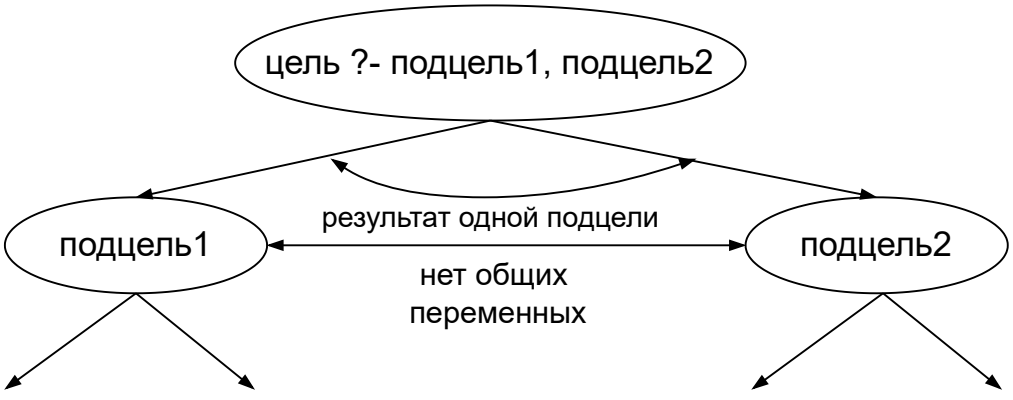

Рисунок 1.1 – Пример OR-параллелизма.

В основе идеи *независимого AND-параллелизма* лежит распараллеливание конъюнкции подцелей, которые не имеют одни и те же переменные. Такие термы не будут влиять на выполнение друг друга, устраняя необходимость введения какой-либо формы синхронизации во время параллельного выполнения. Данный способ проиллюстрирован на рисунке 1.2.

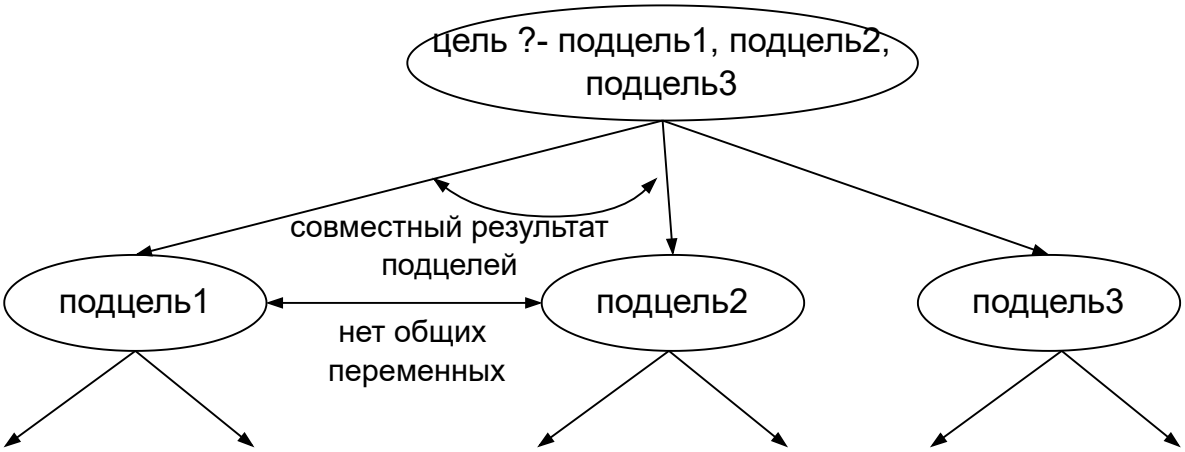

Рисунок 1.2 – Пример AND-параллелизма.

Зависимый AND-параллелизм возникает, когда задается конъюнктивная цель вида $p(X)$ & $q(X)$ — две подцели выполняются параллельно, но при этом используют общую переменную.

Развитие идей Prolog привело к созданию таких языков, как Datalog [7], Yedalog [8] и др. *Datalog* — декларативный язык программирования, который синтаксически является подмножеством Пролога. Программа Datalog состоит из конечного числа фактов и правил. Они задаются с помощью атомарных формул, которые состоят из предикатов с аргументами (листинг 1.1). В *зависимом OR-параллелизме* выполняется дизъюнкция цели с общей переменной.

Листинг 1.1 – Пример описания отношений с использованием Datalog

```
father relational table
- - - - - - - - - - - -
harry    john
john     david

FACTS
father(harry, john).
father(john, david).

RULE
grandfather(Z, X) :- father(Y, X), father(Z, Y).
```

Полученные в последние двадцать пять лет результаты в теории параллельного логического вывода (В. Ю. Мельцов, А. И. Симонов [9]; М. Н. Томчук, Д. А. Страбыкин, Е. В. Агалаков [10] и др.) позволяют вплотную подойти к реализации методов и машин логического вывода на высокопроизводительных многопроцессорных системах как с общей, так и с распределенной памятью.

1.3 Вывод

Описано введение в предметную область логического вывода. Приведены определения, которые в дальнейшем будут использованы при обзоре способов использования параллелизма.

2 Классификация существующих решений

В данном разделе представлены существующие идеи, заложенные в основе параллельного логического программирования, и описан предложенный способ реализации на графическом процессоре.

2.1 Использование неявного параллелизма

Фундаментальная идея, лежащая в основе неявного параллелизма, основывается на том, чтобы сделать его прозрачным для программиста, автоматически выполняя параллельно некоторые шаги, относящиеся к операционной семантике языка [11].

2.1.1 AND-параллелизм

Оригинальные реализации независимого AND-параллелизма основывались на низкоуровневых действиях с базовой машиной – служит вариантом расширения абстрактной машины Уоррена [12]. Такой низкоуровневый подход необходим для снижения накладных расходов на выполнение, связанное с параллелизмом; но эти реализации довольно трудно поддерживать с улучшением технологии последовательного выполнения.

Методы зависимого AND-параллелизма нашли применение в системах логического программирования, которые развились из Prolog, таких как Mercury [13] и EAM (Extended Andorra Model) [14].

Упрощенную архитектуру для зависимого AND-параллелизма можно найти в параллельной реализации *Mercury* [15]. Предлагаемая архитектура использует многопоточную реализацию для создания нескольких «рабочих», которые одновременно выполняют некоторые подцели. Позже это было расширено до реализации, которая поддерживает связь между подцелями. Предложенная система заменяет использование единой централизованной очереди задач набором локальных рабочих очередей, аналогичным иным AND-параллельным реализациям Пролога. Другим компонентом этой системы является использование анализа затрат для выявления перспективных подцелей во время компиляции для параллельного выполнения.

EAM. Расширенная модель Андорры позволяет выполнять подцели одновременно до тех пор, пока они детерминированы или пока они не запрашивают недетерминированную привязку внешней переменной. Когда присутствует недетерминизм, вычисление может разделиться в форме OR-параллелизма

2.1.2 OR-параллелизм

С. Коста, И. Дутра предложили реализацию OR-параллелизма в многопоточной версии YAP Prolog [16], которая опирается на модель копирования стека. Позже совместно с Р. Роша они распространили этот подход на системы, состоящие из кластеров многоядерных процессоров, решая проблему сочетания архитектур общей и распределенной памяти [17]. Ими была предложена многоуровневая модель, основанная на два уровня «работников»: «одиночные работники» и «группы работников», а также возможность использовать различные стратегии планирования для распределения работы между командами и между «работниками» внутри команды. Команда «работников» формируется теми «работниками», которые совместно используют одно и то же адресное пространство памяти.

2.2 Использование GPU

Графические процессоры (GPU) — это устройства с параллельной структурой, изначально разработанные для поддержки компьютерной графики и обработки изображений. В последнее время использование таких многоядерных систем стало широко распространенным в приложениях, которые требуют объемных вычислительных мощностей. Архитектура GPU отличается от CPU (рисунок 2.1).

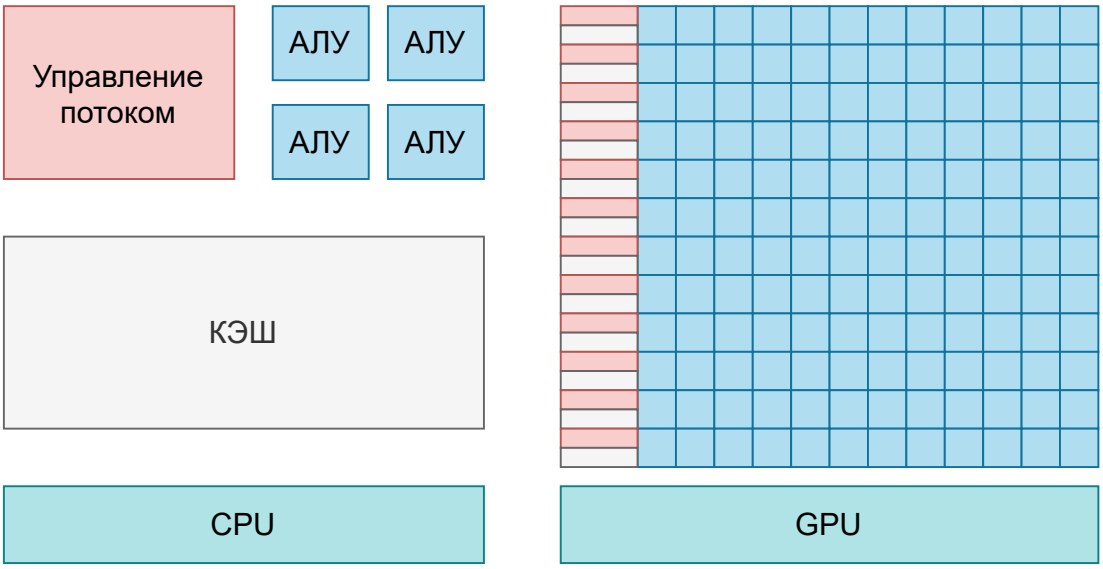

Рисунок 2.1 – Различие между архитектурами CPU и GPU.

2.2.1 Программное обеспечение для работы с GPU

Для работы с графическими процессорами существует несколько распространенных технологий.

1. CUDA. Данная технология предназначена для разработки приложений на массивно-параллельных вычислительных устройствах. Архитектура построена так, что GPU выступает в качестве сопроцессора для CPU

[18]. Вычисления организованы в виде сетки блоков потоков; внутри каждого блока потоки выполняются одновременно, но на остальных блоках они могут находиться на разных стадиях выполнения. Таким образом, каждая подзадача решается отдельно. Число одновременных выполняемых потоков зависит от числа ядер на графическом процессоре. В современных устройствах оно отличается от числа центрального процессора в большую сторону. Так NVIDIA TITAN X (PASCAL) содержит 3584 ядер [19], что позволит ускорить время выполнения задач.

2. OpenCL. Модель платформы состоит из хоста, подключенного к одному или нескольким устройствам OpenCL; каждое из них разделено на один или несколько вычислительных блоков, которые далее также разделены на один или несколько элементов обработки [20], внутри которых выполняются вычисления на устройстве. Предоставляется иерархическая модель параллельного программирования данных. Существует два способа указать иерархическое подразделение. В явной модели определяется общее количество рабочих элементов для параллельного выполнения, а также то, как рабочие элементы распределяются между рабочими группами. В неявной модели указывается только общее количество рабочих элементов для параллельного выполнения, а разделение на рабочие группы управляется реализацией OpenCL.

2.3 Движок Datalog на GPU

Программы Datalog могут быть оценены при помощи подходов «сверху-вниз» и «снизу-вверх». В первом начинает выполнение с цели и сводится к подцелям до тех пор, пока простое решение не будет найдено. Такой подход, использующийся в Prolog, не может быть выполнен на GPU ввиду того, что время обработки отдельных кортежей может отличаться в разы.

Во втором подходе применяются правила к данным фактам, тем самым выводятся новые; Этот процесс повторяется до тех пор, пока больше не будет ничего получено. Запрос рассматривается только в конце для выбора нужных факты, которые ему соответствуют. Этот подход подходит для GPU, поскольку такие реляционные операции показывают одинаковое время обработки для разных кортежей. Также правила могут оцениваться в любом порядке.

GPU-Datalog основан на трех основных операторах реляционной алгебры: select, join, projection [21]. Это представлено на рисунке 2.2.

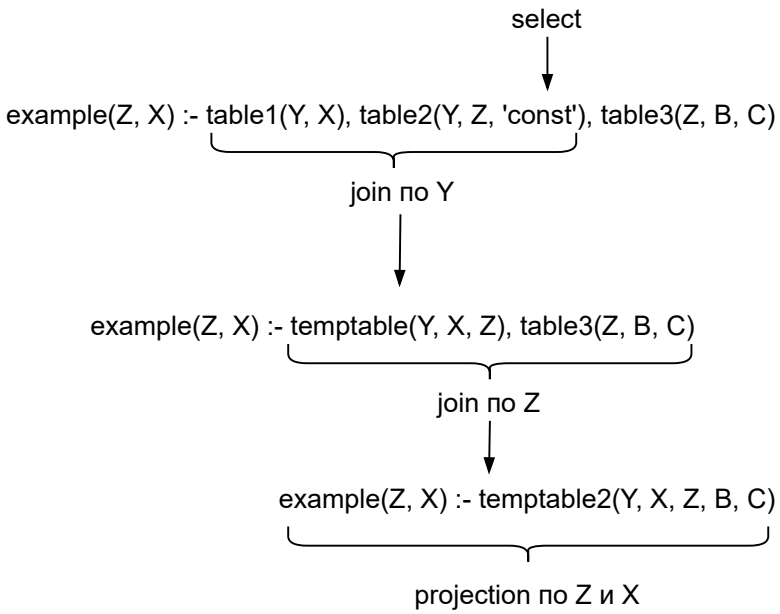

Рисунок 2.2 – Оценка Datalog на основе операций реляционной алгебры.

GPU-Datalog также включает вторичные операторы как отрицание (¬), сравнение (<, >, <, =, >=, <=), агрегатную функцию (GROUP BY) и связанные с ним операции (SUM, COUNT, AVG) и арифметические знаки (+, -, *, /))

GPU-Datalog был интегрирован в качестве модуля в YAP Prolog [22], чтобы связать модель базы данных с миром графических процессоров. Представляет собой комплексное решение, в котором используются как GPU, так

и CPU. Движок этой реализации организован при помощи различных ядер GPU. При раскрытии правила для каждой пары подцелей сначала выбираются ядра select и selfjoin для устранения нерелевантных кортежей. За ними следуют другие ядра - join и projection. При этом данные, посылаемые на GPU, организованны в виде массивов, хранящихся в глобальной памяти.

Чтобы использовать возможности графического процессора для обработки чисел и иметь минимальное возможное время обработки каждого кортежа, факты и правила идентифицируются с числами и используются как числа. Таким образом, строки хранятся в хэшированном словаре, где уникальной строке присваивается уникальный номер.

Выборка (select): определяется двумя значениями (номер столбца и значение константы), которые посылаются массив, хранящий более чем одну выборку. Например, столбцы с номерами 0, 2, 5 будут определять константы a,b,c соответственно (формула (2.1)).

$$fact1('a', X, 'b', Y, Z, 'c'). \rightarrow [0, 'a', 2, 'b', 5, 'c'] \tag{2.1}$$

Соединение (join): определяется двумя значениями - номер столбца по отношению к первому соединению и номером столбца по отношению ко второму соединению (формула (2.2)):

$$fact1(A, X, Y, Z), fact2(Z, X, B, C, Y). \rightarrow [1, 1, 2, 4, 3, 0] \tag{2.2}$$

Другие операции организованы также с использованием массивов чисел, которые хранятся в разделяемой памяти GPU ввиду их малого размера.

2.3.1 Распределение памяти на GPU

Передача данных между памятью графического процессора и памятью хоста (CPU) является дорогостоящей. Предложена схема управления памятью, которая пытается свести к минимуму количество таких передач [21]. Её цель состоит в том, чтобы сохранять факты и результаты правил в памяти графического процессора как можно дольше, чтобы, если они используются более одного раза, их можно было часто повторно использовать из памяти графического процессора. Для этого отслеживается доступная и используемая память графического процессора, а также ведется список с информацией о каждом факте и результате правила, которые хранятся в памяти графического процессора. Когда данные (факты или результаты правил) запрашиваются для загрузки в память графического процессора, они сначала просматриваются в этом списке. Если он найден, его запись в списке перемещается в начало списка; в противном случае для данных выделяется память, и для них создается запись списка в начале списка. В любом случае возвращается его адрес в памяти . Если для выделения памяти для данных требуется освободить другие факты и результаты правил, сначала освобождаются те, которые находятся в конце списка, до тех пор, пока не будет получено достаточно памяти — результаты правил записываются в память процессора перед их освобождением. Таким образом, результаты последних использованных фактов и правил сохраняются в памяти графического процессора.

2.3.2 Операторы реляционной алгебры на GPU

Выборка (select): для запуска на GPU необходимо возникают следующие моменты:

1. результат выборки предварительно неизвестен;

2. каждый поток графического процессора должен знать, в какую ячейку глобальной памяти он запишет свой результат, не связываясь с другими потоками графического процессора.

Для решения этих проблем используются три разных ядра: первое помечает все строки, удовлетворяющие предикату выбора; второе выполняет суммирование префиксов для меток, чтобы определить размер буфера результатов и местоположение. Последнее — записывает полученные результаты.

Соединение (join): используются три типа соединения - одиночное (необходимо для соединения двух столбцов $table_1(X, Y)$ и $table_2(Y, Z)$), множественное (соединение двух и более столбцов - $table_1(X, Y)$ и $table_2(X, Y)$), самообъединение (соединение двух одинаковых столбцов той же таблицы $table_1(X, X)$). Улучшение версии Indexed Nested Loop Join включает CSS-дерево (Cache Sensitive Search Tree - дерево поиска с учетом кэша). Такая структура очень подходит для графических процессоров, поскольку её можно построить параллельно.

Проекция (projection): включает в себя взятие всех элементов каждого требуемого столбца и сохранение их в новой ячейке памяти. Более высокая пропускная способность памяти графического процессора по сравнению с пропускной способностью центрального процессора и тот факт, что результаты остаются в ней графического процессора для дальнейшей обработки, делают проекцию подходящей операцией для обработки на графическом процессоре.

2.3.3 Экспериментальная оценка

На рисунках 2.3 и 2.4 представлены результаты эксперимента по сравнению скорости выполнения реализации GPU-Datalog и YAP-Prolog [7].

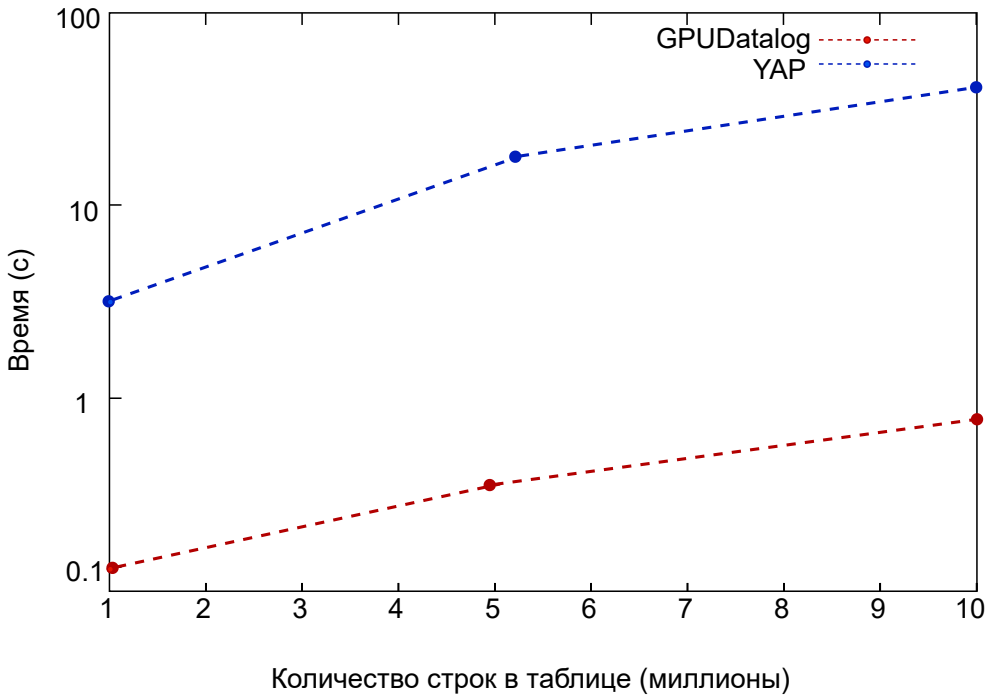

Рисунок 2.3 – Выполнение операции join.

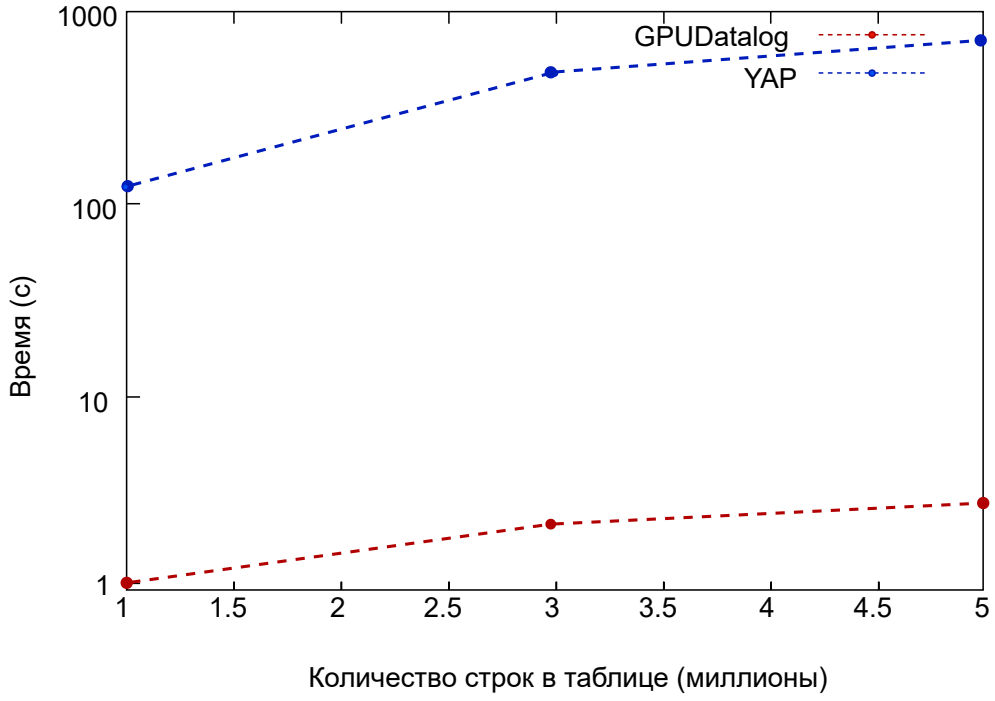

Рисунок 2.4 – Выполнение рекурсивного запроса.

Приведенные результаты представляют перспективным направление раз-

вития GPU-Datalog на основе Prolog и расширенных инструкций WAM.

2.4 Вывод

Подход к распараллеливанию логического вывода формировался в разных представлениях. Стали развиваться идеи реализации на графических процессорах ввиду их быстродействия по сравнению с CPU [23].

Заключение

В результате выполнения работы была проанализирована предметная область и обзор текущих исследований в деятельности развития параллельной реализации логического вывода. Данный обзор подчеркивает разнообразие методов, изучаемых сообществом логического программирования для использования параллелизма в различных вариантах данной парадигмы.

Развитие архитектуры графических процессоров создает новые подходы в исследовании данного направления. Использование большего числа ядер по сравнению с центральным процессором позволит ускорить выполнения поиска решений для логического вывода.

Направление стиля развития логического вывода вытекает из распространенной реализации — Prolog. Основанные на нем описанные методы позволяют выполнить параллелизацию современных вариантов языков логического программирования. На примере Datalog было показано развитие метода OR-параллелизма с использованием операторов реляционной алгебры на примере графического процессора.

Список литературы

1. Kowalski R. Logic for Problem Solving. Elseiver North Holland //Inc. – 1979.

2. Dovier A. Formisano A. Gupta G. Hermenegildo M. Pontelli E. Rocha R. Parallel Logic Programming: A Sequel //arXiv preprint arXiv:2111.11218. - 2021.

3. Shapiro E. Y. Alternation and the computational complexity of logic programs //The Journal of Logic Programming. - 1984. – Т. 1. – №. 1. - С. 19-33.

4. Адаменко А. Н., Кучуков А. М. Логическое программирование и Visual Prolog //СПб.: БХВ-Петербург. - 2003. - Т. 992.

5. Тутынин В. А. Применение многоагентного подхода в проектировании архитектур машины логического вывода //Информационные технологии. - 2019. – С. 113-113.

6. Bramer M. Logic programming with Prolog. – Secaucus : Springer, 2005. - Т. 9. – С. 10.5555.

7. Abiteboul S., Vianu V. Datalog extensions for database queries and updates //Journal of Computer and System Sciences. – 1991. - Т. 43. – №. 1. – С. 62-124.

8. Chin B. et al. Yedalog: Exploring knowledge at scale //1st Summit on Advances in Programming Languages (SNAPL 2015). - Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

9. Мельцов В. Ю., Симонов А. И. Разработка мультиагентной системы для моделирования параллельного логического вывода //Общество, наука, инновации (НПК-2013). - 2013. - С. 950-952.

10. Томчук М. Н., Страбыкин Д. А., Агалаков Е. В. Метод параллельного логического вывода следствий для исчисления высказываний //Программные продукты и системы. – 2012. – №. 2. – С. 142-144.

11. Conery J. S. Parallel execution of logic programs. - Springer Science & Business Media, 2012. – Т. 25.

12. Gabriel J. R. et al. A tutorial on the warren abstract machine for computational logic. – 1985.

13. Conway T. Towards parallel Mercury. - University of Melbourne, Department of Computer Science and Software Engineering, 2002.

14. Lopes R., Costa V. S., Silva F. A design and implementation of the Extended Andorra Model1 //Theory and Practice of Logic Programming. – 2012. – Т. 12. - №. 3. - С. 319-360.

15. Bone P. Automatic parallelisation for Mercury : дис. – 2012.

16. Costa V. S., Dutra I., Rocha R. Threads and or-parallelism unified //Theory and Practice of Logic Programming. – 2010. – Т. 10. – №. 4-6. – С. 417-432.

17. Santos J., Rocha R. On the implementation of an Or-parallel prolog system for clusters of multicores //Theory and Practice of Logic Programming. – 2016. – Т. 16. – №. 5-6. – С. 899-915.

18. Козлов С. О., Медведев А. А. Использование технологии cuda при разработке приложений для параллельных вычислительных устройств //Вестник Курганского государственного университета. - 2015. – №. 4 (38). – С. 106-112.

19. Bianco S. et al. Benchmark analysis of representative deep neural network architectures //IEEE Access. – 2018. – Т. 6. – С. 64270-64277.

20. Munshi A. The opencl specification //2009 IEEE Hot Chips 21 Symposium (HCS). – IEEE, 2009. – С. 1-314.

21. Alberto Martinez Angeles C. A. et al. A Datalog Engine for GPUs. -– 2014.

22. Costa V. S., Rocha R., Damas L. The YAP prolog system //Theory and Practice of Logic Programming. - 2012. – Т. 12. – №. 1-2. – С. 5-34.

23. Козина А. В. и др. Сравнение эффективностей многопоточных реализаций алгоритма шифрования RSA на GPU и CPU //Электронный журнал: наука, техника и образование. – 2018. – №. СВ1. – С. 124-132.