САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Митенев Алексей Васильевич

Особенности взаимодействия CPU и GPU при решении задачи синтаксического анализа

Курсовая работа

Научный руководитель:
ст. преп. Григорьев С. В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Sistem Engineering

Aleksey Mitenev

Features of interaction of the CPU and GPU to solve the problem of parsing

Graduation Thesis

Admitted for defence.
Head of the chair:

Scientific supervisor:
senior lecturer Semyon Grigorev

Reviewer:

Saint-Petersburg
2015

Оглавление

Введение

Графический процессор (GPU) изначально проектировался только для обработки компьютерной графики. Несмотря на это, в настоящее время графический процессор применяют для общих вычислений, которые раньше выполнялись только на CPU. В данный момент, когда современные видеокарты имеют огромное количество ядер (Например, видеокарта NVidea GeForce GTX 980 имеет 2048 ядер), решение вычислительных задач с помощью GPU становится все более и более популярным. Одним из инструментов для написания программ для GPU является OpenCL.

Фреймворк OpenCL [5] (Open Computing Language) был разработан в 2008 году и предназначен для написания программ для GPU. Во фреймворк OpenCL входит язык программирования, который базируется на стандарте C99 с некоторыми ограничениями и расширениями: отсутствует поддержка рекурсии и указателей на функции, имеется другой набор встроенных функций. OpenCL является полностью открытым стандартом.

Драйвера OpenCL устройств автоматизируют старт и работу ядер GPU. Например, если нам нужно запустить миллион процессов, а у нас в распоряжении всего тысяча ядер, то драйвера будут автоматически запускать каждый процесс со следующей задачей после освобождения ресурсов. Понимание физического уровня разработчику требуется только для того, чтобы иметь представление о возможностях взаимодействия между процессами и доступа процессов в память.

Графические процессоры изначально проектировались под SIMD архитектуру, и в этом их отличие от процессоров общего назначения, которые изначально проектировались под SISD. CPU лучше справляется с последовательными задачами, GPU же дает выигрыш в производительности, если в задаче обрабатывается большой объем данных и присутствует параллелизм.

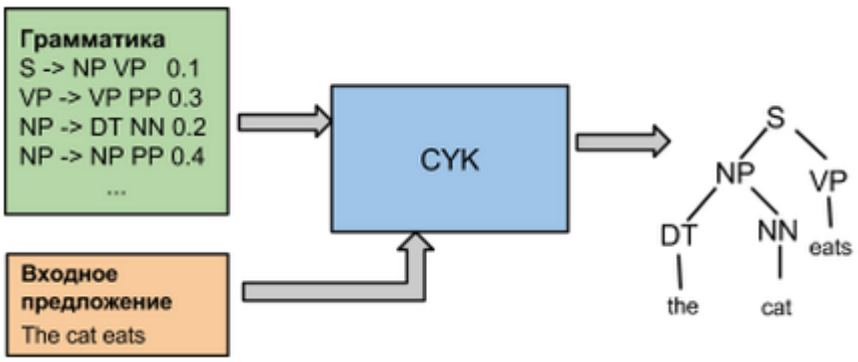

Рис. 1: CYK алгоритм: схема работы

Параллелизм можно найти далеко не в каждой задаче. Но есть задачи, в которых можно параллельно проводить очень большое количество вычислений. Самый простой пример – задача умножения матриц. Действительно, каждый элемент результи-

рующей матрицы можно вычислять независимо от других. Есть и менее очевидные задачи, в которых присутствует параллелизм. Одна из таких задач касается синтаксического анализа – задача синтаксического анализа с помощью CYK[3] алгоритма.

CYK принимает на вход последовательность слов языка и контекстно свободную грамматику в нормальной форме Хомского [2], и выдает дерево синтаксического анализа. Основным преимуществом этого анализатора является то, что он может разобрать любую контекстно свободную грамматику. Эта особенность может быть применена, например, для разбора контекстно свободных грамматик, приближенных к грамматикам естественных языков.

1. Постановка задачи

Целью данной работы является оптимизация алгоритма синтаксического анализа CYK, интегрированного в проект YaccConstructor, для выполнения на GPU. Для выполнения этой цели необходимо выполнить следующие задачи:

- Проанализировать исходную версию алгоритма и выявить основные недочеты;
- Провести оптимизации для увеличения производительности, учитывая специфику разработки для GPU;
- Проанализировать реализованные оптимизации: выяснить, какие изменения произвели данные оптимизации(уменьшение времени выполнения, увеличение длины максимального входа).
- Протестировать алгоритм на грамматике Transact SQL

2. Обзор

2.1. Архитектура OpenCL

Несмотря на то, что OpenCL старается изолировать пользователя от внутренней архитектуры, при написании программ для OpenCL разработчику необходимо иметь некоторые представления об его внутреннем устройстве, особенно о модели памяти, по той причине, что разработчик может самостоятельно определять, в какой памяти хранить данные. Для начала, необходимо рассмотреть следующие ниже базовые понятия.

- Хост (host) — главная машина, которая выполняет управляющую роль — выделяет и освобождает память на устройстве и запускает задачи на устройстве. В нашем случае хостом будет являться CPU;
- Устройство (device) — выполняет задачи, которые получает с хоста. Хост может быть соединен с несколькими устройствами. В нашем случае устройством будет являться GPU;
- Ядро (kernel) — функция, запускаемая хостом на устройстве.

Модель выполнения OpenCL такова: имеется хост и некоторые устройства, которые с ним соединены. В момент, когда необходимо посчитать что-то на устройстве(в нашем случае, это GPU), хост передает данные устройству, ставит ядро в очередь на исполнение, после этого ядро выполняется на устройстве и результат возвращается на хост.

Перед тем, как поставить ядро в очередь на выполнение, нужно выбрать размер и размерность n-мерного индексного пространства, наиболее выгодную для выполнения исходного алгоритма. Индексное пространство отвечает за количество процессов, которые надо выполнить. Ядро, выполняющееся для конкретного индекса называется рабочим элементом(Work-Item). Когда идет речь о применении OpenCL к GPU, каждый рабочий элемент сопоставляется ядру GPU. Каждый рабочий элемент выполняет один и тот же код, но данные, с которыми он работает, могут различаться.

Каждый рабочий элемент имеет свою приватную память (private memory), к которой есть доступ только у него. Эта память наиболее быстрая по сравнению с другими, но у нее очень небольшой размер - обычно около 64 Кб. Все данные, к которым не должны иметь доступа другие процессы, нужно хранить в приватной памяти. Рабочие элементы организуются в рабочие группы (work-groups). Группы предоставляют более крупное разбиение в пространстве индексов. Каждому рабочему элементу, входящему в группу, сопоставляется уникальный, в рамках группы, локальный индекс.

У каждой рабочей группы есть локальная память (local memory). К локальной памяти имеют доступ все рабочие элементы, входящие в данную рабочую группу. Она

имеет промежуточный между приватной и глобальной памятью размер и довольно быструю скорость доступа. Используется для синхронизации рабочих элементов из одной рабочей группы. У всех рабочих групп имеется общая память – глобальная. Она имеет наибольший размер и наименьшую скорость доступа, используется для хранения данных, которые используются элементами различных рабочих групп.

Так же имеется константная память(Constant memory), к которой имеют доступ все рабочие группы, она используется для хранения констант.

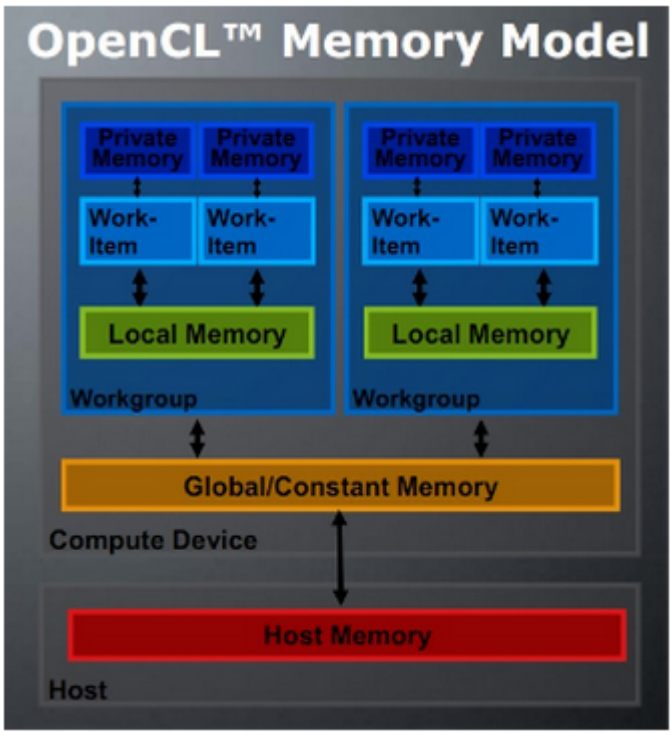

Рис. 2: OpenCL: модель памяти

2.2. CYK как метод синтаксического анализа

Алгоритм был впервые описан в статье Kasami, T. (1965) “An efficient recognition and syntax-analysis algorithm for context-free languages” [3]. CYK является восходящим табличным анализатором, способным разобрать любую контекстно-свободную грамматику, но имеет сложность |G| * O (n3), где n - количество токенов в предложении, которое подается на вход, а |G| - константа для конкретной грамматики, зависящая от количества правил в ней. Очень часто эту константу недооценивают, хотя она может оказаться довольно большой.

К счастью, одним из достоинств CYK анализатора является то, что разные части таблицы заполняются независимо друг от друга, что дает возможность получить выигрыш в производительности, используя параллелизм. Это одна из тех задач, которые идеально подходят для выполнения на графическом процессоре.Еще одной важ-

ной особенностью CYK анализатора является то, что он принимает на вход только контекстно-свободные грамматики в нормальной форме Хомского. Грамматика находится в нормальной форме Хомского, если каждое правило этой грамматики имеет вид: A -> B C либо A -> a либо S -> ε где S - стартовый нетерминал; A, B, C - нетерминалы; a - терминал. При этом любую контекстно-свободную грамматику можно привести к нормальной форме Хомского.

2.3. YaccConstructor и Brahma.FSharp

Чтобы абстрагироваться от работы с грамматикой и преобразований с ней (в том числе, в нормальную форму Хомского), будут использованы возможности проекта YaccConstructor [7].

YaccConstructor – Это инструмент для синтаксического анализа и обработки грамматик. Также это фреймворк для исследований и разработки в области генераторов синтаксических анализаторов, компиляторов и других исследований, связанных с грамматиками на платформе .NET, который разрабатывается на кафедре Системного Программирования на математико-механическом факультете СПБГУ. Архитектура проекта YaccConstructor разделяется на 3 основные части:

- Frontends - набор DSL для задания грамматики;
- Conversions - набор преобразований над грамматиками, заданными во фронтендах;
- Backends - набор синтаксических анализаторов, получающих на вход некоторое внутреннее представление грамматики, полученное из фронтенда.

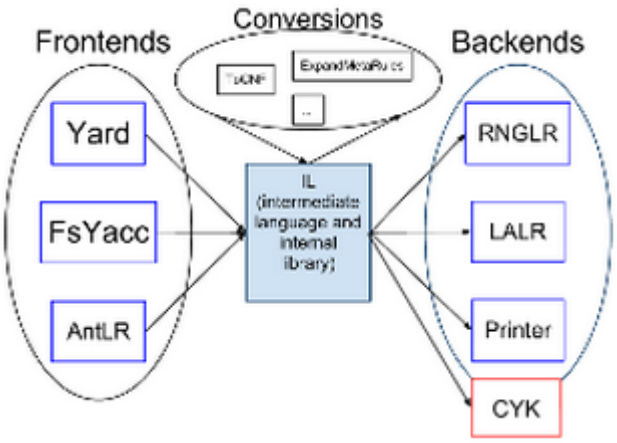

Рис. 3: YaccConstructor: архитектура

Реализация CYK будет выступать в качестве одного из Backend'ов. Главным преобразованием, которое будет использовано, станет TOCNF. Таким образом, будет получена уже подготовленная грамматика в НФХ.

Проект YaccConstructor реализован на языке F#. Чтобы облегчить интеграцию, для реализации работы с GPU будет использована библиотека Brahma.FSharp [1]. Brahma.FSharp – это библиотека, позволяющая писать программы для GPU на языке F# путем транслирования в OpenCL.

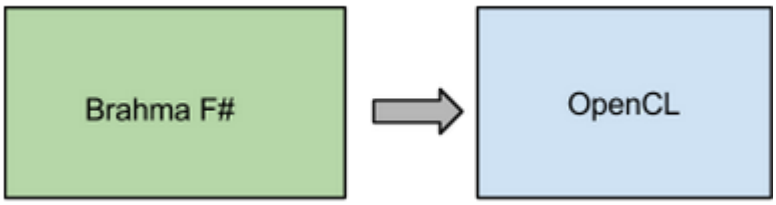

Рис. 4: Brahma.Fsharp

2.4. Обзор алгоритма CYK

При реализации была использована статья Youngmin Yi, Chao-Yue Lai, Slav Petrov, Kurt Keutzer. Efficient Parallel CKY Parsing on GPUs [6]. В ней описан модифицированный алгоритм CYK, который можно использовать для взвешенных грамматик для получения дерева разбора максимального веса, а так же архитектура OpenCL и основные оптимизации для CYK парсера на GPU. Подробно описывается применения CYK анализатора для разбора грамматики английского языка.

Рассмотрим общую схему алгоритма. CYK является табличным восходящим анализатором. Это значит, что получая на вход последовательность терминалов, CYK пытается группировать их в нетерминалы и в конце получить стартовый нетерминал. Основная структура данных - массив table [r][c][nTermSize], где вход:

- r - количество слов во входном предложении;
- c = r = количество слов во входном предложении;
- nTermSize - количество нетерминалов в грамматике. В ячейке таблицы хранится минимальный вес вывода нетерминала и некоторая дополнительная информация.

Разберем псевдокод алгоритма:

```
table - CYK table
nWords - number of words from input
grammar - input grammar
//initialize first row
//S -> a, where  - terminal, put rule weight in cell
for column = 0 to nWords - 1
```

```
    foreach nTerminal   grammar
        foreach rule r per nTerminal
            //r is "nTerminal   a", a is terminal
            table[0][column][nTerminal]
for currRow = 1 to wordsLength - 1
    for column = 0 to nWords - wordsLength
  foreach nTerminal   grammar
  max = 0;
      foreach rule r per nTerminal
          //r is "nTerminal   A B"
          for k = 0 to currRow - 1
              //calculate weight
              lweight = table[k][column][A];
              rweight = table[currRow-k-1][col+k+1][B];
              weight = rule weight + lweight + rweight;
              //maximum change
              if weight > max
                  max = weight;
      table[currRow][column][nTerminal] = max;
```

1. Осуществляется заполнение первой строки таблицы, при этом обходятся все унарные правила вида Neterminal -> terminal, где терминал соответствует столбцу, который мы заполняем, то есть, номеру слова в предложении.

2. Далее, последовательно заполняются последующие строки. В первой строке все нетерминалы, которые можно получить из 1 символа, во второй - из 2, в третьей - из трех и т.д. При заполнении каждой последующей строки используются ранее заполненные ячейки из предыдущих строк, при этом просматриваются только определенные (дочерние) ячейки, как показано на рисунке 5. Если в грамматике есть правило S -> X Z, в исходной ячейке проверяется выводимость нетерминала S, а в соответствующих дочерних ячейках нетерминалы X и Y имеют вес, отличный от 0, то и S выводим(рис 5 (2)), а вес вывода равен сумме весов в дочерних ячейках и веса правила.

Выполнение алгоритма заканчивается после заполнения таблицы. Чтобы узнать, выводимо ли входное предложение в нашей грамматике, нужно посмотреть в ячейку table[lastRow][FirstColunm][S], где S - стартовый нетерминал грамматики. Если она содержит вес, отличный от 0, значит, предложение выводимо, иначе - нет.

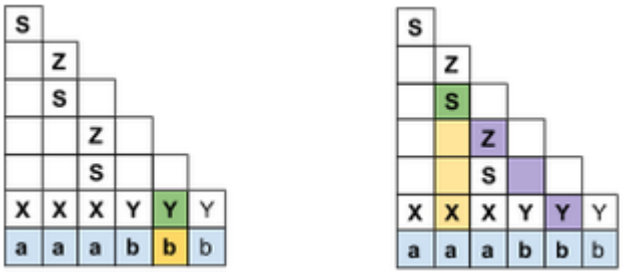

Рис. 5: Пример работы CYK анализатора

3. Реализация

Необходимо было оптимизировать синтаксический анализатор CYK, реализованный на языке F#, на базе проекта YaccConstructor, используя библиотеку Brahma.FSharp.

3.1. Анализ исходного алгортима

Был проанализирован исходный алгоритм, интегрированный в проект YaccConstructor. Была выявлена проблема в производительности: при запуске на тесте, где входная цепочка состояла из 5 лексем и использовалась грамматика SQL, алгоритм работал за 20 секунд. При увеличении количества лексем время выполнения сильно возрастало. На тесте из 10 лексем алгоритм работал 60 секунд. Кроме того, на тестах большего размера возникала проблема: драйвер видеокарты неожиданно заканчивал свою работу, и дальнейшие вычисления не проводились. Простым исправлением ошибок проблемы решить не удалось, было принято решение переписать значительную часть алгоритма.

3.2. Особенности интеграции с YaccConstructor

Как было упомянуто выше, CYK анализатор получает на вход некоторое представление грамматики, которая была написана на одном из фронтендов. Грамматика получается виде массива продукций, каждая из которых представляется в виде uint64. Представление правила вида A -> B C <weight> <label> происходит следующим образом:

- Голова продукции - 16 бит;
- Каждый из нетерминалов продукции - 16 бит;
- Вес продукции (для взвешенных грамматик) - 8 бит;
- Метка (используется для диалектов) - 8 бит.

Следует упомянуть, что всем нетерминалам в грамматике присваивается номер от 0 до n-1, где n – количество нетерминалов. Доступ к соответствующим элементам

продукции осуществляется с помощью побитовых операций. Так же CYK поддерживает разбор взвешенных грамматик [4]. Взвешенные грамматики предназначены, чтобы убрать неоднозначность у контекстно-свободных грамматик. Каждому правилу в грамматике назначается вес. Соответственно, если у входной строки в данной грамматике несколько деревьев разбора, выбирается дерево с наименьшим весом.

3.3. Параллелизм

Как уже было написано ранее, последовательный алгоритм не эффективен. Можно отметить, что при заполнении каждой последующей строки таблицы используются предыдущие. Таким образом, строки таблицы должны заполняться последовательно. В отдельно взятой строке, вес в каждой ячейке таблицы, соответствующей нетерминалу, может вычисляться независимо от других ячеек из той же строки. В отдельно взятой ячейке, соответствующей нетерминалу, наибольший результат для каждого из правил грамматики может вычисляться назависимо(учитывая синхронизацию).

Наиболее производительным оказывается тот случай, когда используется параллелизм по максимуму: в рамках одной строки вычисление максимального веса проходит параллельно для каждой ячейки, соответствующей нетерминалу, параллельно и для каждого правила из грамматики. Рассмотрим вышеописанное в псевдокоде алгоритма:

```
//fill one row
for currRow = 1 to wordsLength - 1                    //sequential
    for column = 0 to nWords - wordsLength            //parallel
        foreach nTerminal   grammar                  //parallel
            foreach rule r per nTerminal              //parallel
```

3.4. Размер рабочей группы

Кроме того, важным фактором для выполнения программ на OpenCL является возможность выставления размера рабочей группы.

Перед началом выполнения программы на GPU задается размер индексного пространства, наиболее удобный для данной задачи. После этого задается размер рабочей группы. От выставленного размера рабочей группы зависит производительность. Наиболее подходящее значение выбирается опытным путем для каждого GPU отдельно.

Кроме того, стоит поговорить о том, как рабочий элемент понимает, какое именно правило грамматики нужно выбирать, ведь ядро содержит одинаковый код. Это происходит благодаря тому, что каждое ядро имеет свой глобальный индекс, globalId, по

нему восстанавливается индекс правила, которое будем выполнять. Таким образом, каждый рабочий элемент будет выполнять свое правило.

3.5. Восстановление дерева

Нужно помнить о том, что наша главная задача не в том, чтобы понять, разбирается ли входная строка в грамматике, а в том, чтобы воспроизвести дерево вывода с наибольшим весом. Если хранить в каждой ячейке таблицы только вес, это сделать не получится. Нужна некоторая дополнительная информация для восстановления вывода. В данной реализации алгоритма в ячейке таблицы хранится номер правила, по которому получен минимальный вес, а так же некоторый параметр, по которому можно понять, какие из дочерних ячеек были задействованы. Это дает возможность после завершения заполнения таблицы пройтись рекурсивно сверху-вниз по таблице и получить дерево с наибольшим весом.

3.6. Память

В начале выполнения программы таблица CYK, у которой инициализированы веса в первой строке, передается из памяти хоста в глобальную память GPU. К сожалению, таблицу возможно хранить только в глобальной памяти, из за ее внушительного размера (n * n * количество нетерминалов * размер ячейки), и из за того, что ее используют все рабочие элементы. Поэтому нужно минимизировать количество информации, которая содержится в каждой ячейке. В реализации версии алгоритма в каждой ячейке решено хранить параметры, указанные в списке ниже.

- Текущий вес – 8 бит;
- Номер правила, которое было применено для вывода нетерминала – 32 бита;
- Флаг синхронизации - необходим для запрета одновременной записи в одну ячейку несколькими потоками – 8 бит;
- Параметр, отвечающий за то, в каких ячейках выбирались нетерминалы при проверке правила – 8 бит.

При этом все локальные переменные каждого потока хранятся в его private памяти. Кроме того, видно, что ячейки выше главной диагонали матрицы не используются. Таким образом, можно хранить только ту часть матрицы, которая используется.

3.7. Синхронизация

Синхронизация нужна в тот момент, когда оказывается, что в ячейку таблицы пытаются одновременно писать несколько потоков. Поток, относящийся к определенно-

му правилу, проходит по дочерним ячейкам таблицы, вычисляя локальный максимум, после чего пытается записать результат в ячейку таблицы. Проблема заключается в том, что в это же время другой поток вычисляет свой локальный максимум и пытается записать его в ту же самую ячейку таблицы, при этом возникает конфликт. Чтобы это предотвратить, используются атомарные операции. В каждой ячейке таблицы добавлен флаг, который используется как семафор: перед началом записи в ячейку таблицы он ставится в 1 с помощью атомарного присваивания, а после окончания записи он ставится в 0.

3.8. Фильтрация грамматики

Итак, алгоритм CYK состоит из 2 основных шагов: заполнение первой строки таблицы и заполнение последующих строк. Основным этапом является второй. При этом, имеется возможность использовать только правила вида A -> B C, а остальные не использовать, и передавать GPU только нужные правила.

3.9. Замеры производительности

Была проведена апробация CYK на грамматике Transact Sql. Оборудование: Процессор: AMD A6-5200 APU, 4 ядра Видеокарта: AMD Radeon HD 8400, 256 ядер.

Начальный алгоритм был реализован наивно, после этого были произведены следующие оптимизации.

- Алгоритм был переработан, был реализован параллелизм, базирующийся на выполнении правила. Это дало основной выигрыш в скорости. Если в начальном алгоритме на грамматике SQL на тестовом входе из 10 токенов алгоритм работал 60 секунд, то после переработки тест из 90 токенов работает за 13 секунд;

- После путем перебора был выбран размер рабочей группы. Выигрыш в производительности виден на больших тестах: на тесте из 90 токенов он составляет порядка 35%;

- Грамматика была отфильтрована: Во время заполнения первой строки таблицы используются унарные правила вида A -> a, а при выполнении второго шага используются бинарные правила вида A -> B C. Это дало 5-7% выигрыш в производительности;

- Матрица CYK была приведена к треугольной и развернута в одномерный массив, это дало небольшой прирост в производительности (2-3%) из за уменьшения объема передаваемых данных, и, кроме того, позволило увеличить количество лексем во входной строки.

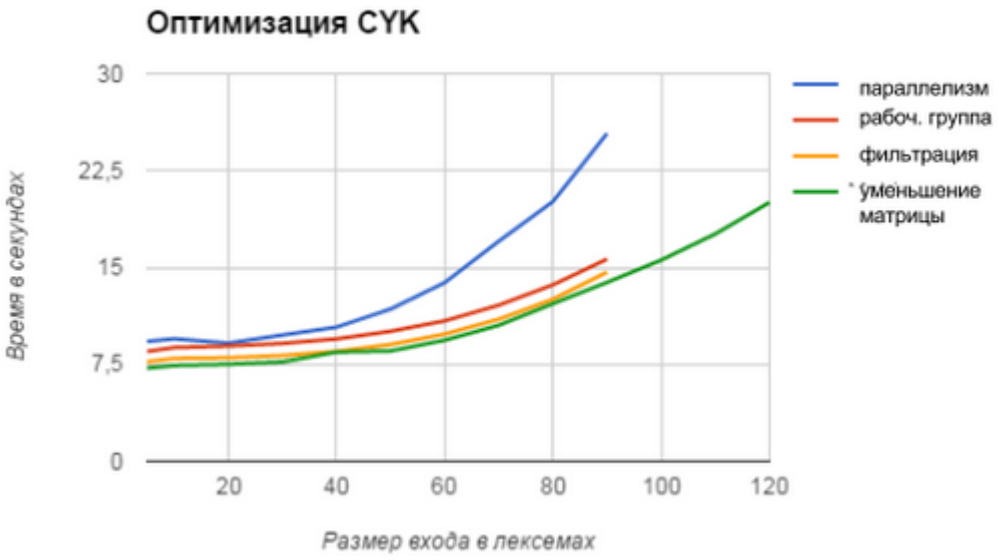

Рис. 6: Влияние оптимизаций

Оптимизации были применены последовательно. Результаты отражены на графике:

На графике не был показан начальный алгоритм по причине его низкой скорости работы и очень маленкого размера максимального входа.

3.10. Сравнение версий алгоритма

Были проведены замеры скорости работы в зависимости от длины входной строки для 3 версий алгоритма - последовательной, использующей параллелизм стандартных средств F#, и GPU версия алгоритма. Результаты представлены на графике ниже.

Несмотря на то, что видеокарта, использованная в этом примере была довольно слабой по сравнению с современными аналогами, на больших тестах CYK на GPU показывает более высокую производительность, чем CPU-параллельный аналог и, тем более, последовательную версию.

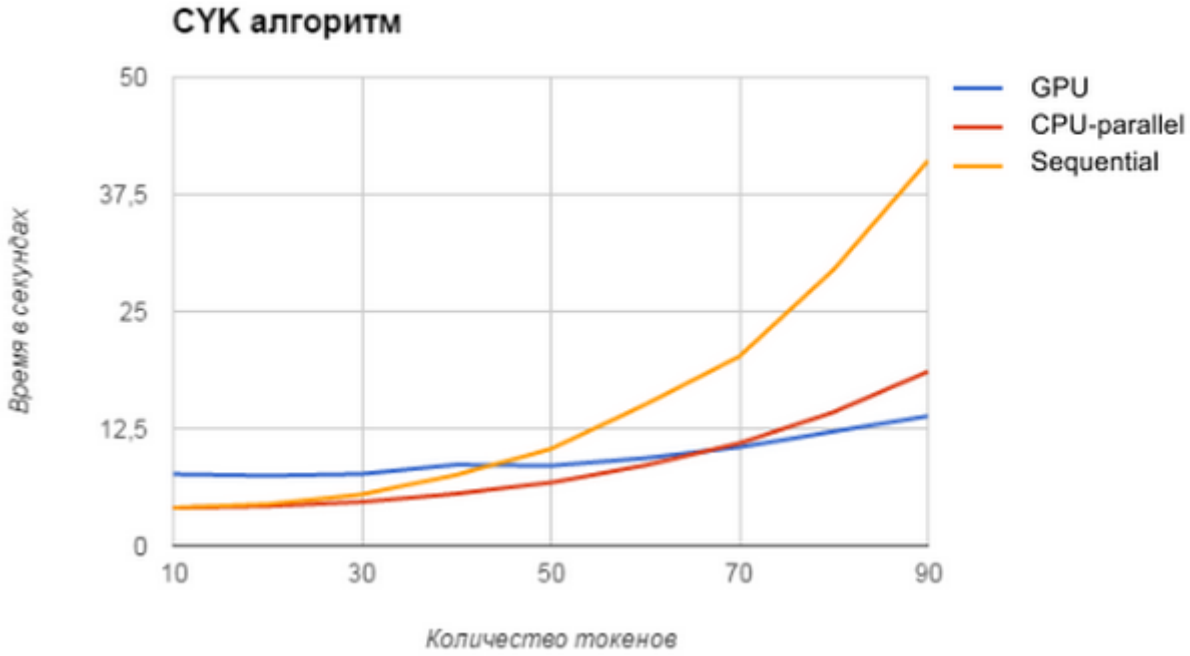

Рис. 7: Сравнение последовательной, параллельной и GPU версий CYK

Заключение

В ходе курсовой работы бели выполнены следующие поставленные задачи:

- Проанализирована исходная версию алгоритма и выявлены основные недочеты;
- Проведены оптимизации для увеличения производительности, учитывая специфику разработки для GPU;
- Реализованные оптимизации проанализированы: выявлено, какие изменения в работе программы произвели данные оптимизации(уменьшение времени выполнения, увеличение длины максимального входа).
- Алгоритм протестирован на грамматике Transact SQL

Направления дальнейшего развития

Главным направлением дальнейшего развития является апробация алгоритма на грамматиках, приближенных к грамматикам естественного языка. Как известно, грамматики естественных языков не являются контекстно свободными, тем не менее, существуют взвешенные контекстно свободные грамматики с сотнями тысяч правил и тысячами нетерминалов, которые порождают языки, практически эквивалентные естественным. Из за большого количества правил в грамматиках можно ожидать большой прирост производительности на GPU.

Список литературы

[1] Brahma.Fsharp. — 2015. — URL: https://sites.google.com/site/semathsrprojects/home/brahma-fsharp/.

[2] Chomsky Noam. On Certain Formal Properties of Grammars. — Indormation and control, 1959. — URL: http://somr.info/lib/Chomsky_1959.pdf.

[3] Kasami T. An efficient recognition and syntax-analysis algorithm for context-free languages. — 1965.

[4] Noah A. Smith Mark Johnson. Weighted and Probabilistic Context-Free Grammars Are Equally Expressive. — Association for Computational Linguistics, 2007. — URL: http://www.aclweb.org/anthology/J07-4003.

[5] OpenCL // Khronos group. — 2015. — URL: https://www.khronos.org/opencl/.

[6] Youngmin Yi Chao-Yue Lai Slav Petrov Kurt Keutzer. Efficient Parallel CKY Parsing on GPUs. — Proceedings of the 12th International Conference on Parsing Technologies, 2011. — URL: www.aclweb.org/anthology/W11-2921.

[7] Кириленко Я.А Григорьев С.В. Д.А. Авдюхин. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем. — Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации., 2013. — URL: http://ntv.spbstu.ru/telecom/article/T3.174.2013_11/.