САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Михайлов Дмитрий Петрович

# Прозрачное использование массово-параллельных архитектур для локальных оптимизаций приложений на .NET

Дипломная работа

Допущена к защите.

Зав. кафедрой:

д.ф.-м.н., проф. Терехов А.Н.

Научный руководитель:

аспирант кафедры системного программирования Григорьев С.В.

Рецензент:

ведущий архитектор Тимофеев Н.М.

Санкт-Петербург

2013

SAINT-PETERSBURG STATE UNIVERSITY

Mathematics&Mechanics Faculty

Software Engineering Chair

Dmitry Mikhaylov

# Transparent usage of massively parallel architectures for local optimizations of .NET applications

Graduation Thesis

Admitted for defence.

Head of the chair:

DSc, Professor A.N. Terekhov

Scientific supervisor:

postgraduate student of chair of software engineering S.V. Grigoriev

Reviewer:

Lead Architect N.M. Timofeev

Saint-Petersburg

2013

# Оглавление

Введение....................5
1 Постановка задачи....................7
2 Обзор существующих решений....................9
2.1 Использование нативного кода....................9
2.1.1 CUDA.NET....................10
2.1.2 Cloo....................10
2.1.3 OpenCLTemplate....................11
2.2 Библиотеки типов и операций....................11
2.2.1 MS Accelerator....................11
2.2.2 FCore....................12
2.3 Трансляция кода на лету....................12
2.3.1 Alea.cuBase....................13
2.4 Выводы....................13
3 Архитектура....................15
3.1 Код ядра и его трансляция....................15
3.2 Промежуточное представление....................16
3.3 Генерация кода....................17
3.4 Использование....................17
4 Особенности реализации....................19
4.1 Доработка Brahma.FSharp....................19
4.1.1 Инициализация массивов....................19
4.1.2 Синхронизация....................20
4.1.3 Локальная память....................21
4.2 Задача поиска подстрок в строке....................21
4.2.1 Представление результатов....................22
4.2.2 Дробление входной строки....................23
4.2.2.1 Последняя итерация....................23
4.2.2.2 Граница итераций....................24
4.2.2.3 Границы потоков....................25

4.2.3 Параллельные чтение и обработка........................................26
4.3 Вычисление оптимальных параметров........................................27
5 Алгоритмы........................................29
5.1 Полный перебор........................................29
5.2 Алгоритм Рабина-Карпа........................................30
5.2.1 Оптимизации........................................30
5.3 Полный перебор с хеш-таблицей........................................31
5.3.1 Оптимизации........................................31
5.4 Алгоритм Ахо-Корасик........................................32
5.4.1 Оптимизации........................................33
6 Апробация........................................34
6.1 Алгоритм Рабина-Карпа........................................34
6.1.1 Результаты........................................35
6.1.2 Выводы........................................35
6.2 Алгоритм Ахо-Корасик........................................36
6.2.1 Результаты........................................37
6.2.2 Выводы........................................37
6.3 Сравнение алгоритмов........................................37
6.3.1 Результаты........................................37
6.4 Тестирование на реальных данных........................................38
6.4.1 Результаты........................................39
6.4.1.1 Поиск 10 шаблонов........................................40
6.4.1.2 Поиск 30 шаблонов........................................40
6.4.1.3 Поиск 50 шаблонов........................................41
6.4.2 Выводы........................................41
Заключение........................................43
Результаты........................................43
Дальнейшее развитие........................................43
А Шаблоны и совпадения........................................45
Список литературы........................................46

# Введение

В задачах, требующих высокопроизводительных вычислений (High-performance computing, HPC), обычно используются такие эффективные языки программирования, как C и Fortran, хорошо зарекомендовавшие себя в этой роли. Многие современные приложения на .NET слишком велики и сложны, чтобы быть целиком написанными на одном из таких языков, но в то же время иногда содержат участки кода, производительность исполнения которых критична при использовании.

В тех случаях, когда необходимости в low-latency HPC нет, бывает достаточно ценой минимальных дополнительных усилий достичь некоторого допустимого уровня производительности, даже если используемые при этом средства предполагают некоторые накладные расходы.

Одним из возможных решений таких проблем является применение массово-параллельных архитектур. Наиболее распространённой их разновидностью, присутствующей в большинстве обычных современных персональных компьютеров, на которые рассчитано требующее оптимизации приложение, являются графические процессоры общего назначения (General-purpose graphics processing unit, GPGPU). В задачах, допускающих большую, чем может предоставить центральный процессор (CPU), степень параллелизма, даже наиболее дешёвые из них часто оказываются более производительными, чем CPU.

К задачам, которые часто требуют существенной оптимизации, относятся, например, сортировка массива и умножение матриц,

встречающиеся в приложениях самого различного назначения. При всей их простоте и известности они остаются весьма ресурсоёмкими на больших входных данных, а потому повышение эффективности их решения не теряет актуальности. Если говорить о менее тривиальных примерах таких задач, нельзя не упомянуть алгоритмы на графах (необходимые, к примеру, в задачах статического анализа кода, возникающих при реинжиниринге программного обеспечения и трансляции) и строковые алгоритмы (необходимые в задачах цифровой криминалистики и биоинформатики), некоторые из которых были реализованы и протестированы в данной работе.

Целью данной работы является существенно влияющая на производительность доработка средства, которое даёт возможность ценой небольших дополнительный усилий на порядки ускорить выполнение критически медленной задачи в рамках .NET-приложения путём исполнения логики, реализованной на одном из языков фреймворка .NET, на GPGPU.

# 1 Постановка задачи

Целью данной работы является доработка библиотеки Brahma.FSharp [1], предназначенной для прозрачной интеграции .NET-приложений и вычислений на GPGPU с помощью фреймворка OpenCL [10], для обеспечения возможности создания конкурентоспособной производительности при её использовании. Для достижения этой цели были выделены следующие основные подзадачи:

1. Изучить особенности GPGPU и их влияние на производительность вычислений.
2. Изучить существующие средства интеграции .NET-приложений и вычислений на GPGPU, сделать их обзор и проанализировать их преимущества и недостатки по сравнению с Brahma.FSharp.
3. Доработать библиотеку Brahma.FSharp. Реализовать возможность работы с массивами, локальной памятью OpenCL и средствами синхронизации. Исправить дефекты библиотеки и доработать её функциональность.
4. Использовать Brahma.FSharp для решения задачи поиска подстрок в строке с целью апробации библиотеки и демонстрации её возможностей, изучения влияния GPGPU и новых функций библиотеки на производительность различных алгоритмов. Отладить библиотеку и определить по результатам её тестирования функциональность, отсутствие которой критически важно для нетривиальных задач.

Задача поиска подстрок в строке предполагает разные архитектуру и алгоритмы решения в зависимости от имеющихся ограничений, поэтому был выбран её конкретный частный случай со следующими свойствами:

- Входная строка и искомые шаблоны становятся известны только непосредственно в момент запроса (online-задача);
- Входная строка может быть больше, чем объём оперативной памяти современных компьютеров (может достигать размеров жёстких дисков в терабайты или сотни гигабайт);
- Длина шаблонов не превосходит 32 байт;
- Количество шаблонов в одном запросе не превосходит 512;
- На компьютере, на котором запущено приложение (далее, хосте), доступно для решения задачи не более 256 мегабайт оперативной памяти.

# 2 Обзор существующих решений

На сегодняшний день в силу большой популярности .NET и растущего интереса разработчиков к GPGPU разнообразие средств для обеспечения их взаимодействия весьма велико, поэтому имеет смысл разделить такие средства на несколько основных категорий, описать их общие особенности и привести конкретные характерные примеры. Все рассмотренные в ходе работы средства интеграции действительно относились к одному из выделенных классов, наследуя большую часть его положительных и отрицательных сторон.

## 2.1 Использование нативного кода

Многие технологии для массово-параллельных вычислений содержат специальные языки программирования для реализации логики, которая будет выполняться на целевом аппаратном обеспечении. К ним можно отнести CUDA C, CUDA C++ и CUDA Fortran для CUDA, а также OpenCL C для OpenCL. Наиболее простым и естественным способом интеграции с такими технологиями является непосредственное написание и использование кода на таком языке. К преимуществам такого способа взаимодействия можно отнести отсутствие каких-либо накладных расходов на запуск задачи на GPGPU и возможность использовать всю полноту функциональности предоставленного разработчиком технологии языка без ограничений, обычно накладываемых средствами интеграции другого типа. Существенным его недостатком является необходимость для .NET-разработчика знать весьма специфический диалект не имеющего отношения к .NET и не поддерживаемого в привычных средствах разработки языка и в целом более высокая сложность

процесса разработки по сравнению с более высокоуровневыми языками. К тому же, в некоторых случаях такой код необходимо скомпилировать отдельным вызовом специального компилятора перед запуском программы, что усложняет процесс управления конфигурацией (Configuration management, CM [9]) при промышленной разработке программного обеспечения.

### *2.1.1 CUDA.NET*[1]

Для использования этой библиотеки необходимо реализовать функцию, которая будет исполняться на GPGPU (kernel, ядро), на CUDA C в отдельном .cu-файле, а затем скомпилировать его в .cubin-файл бинарного модуля CUDA с помощью NVIDIA CUDA Compiler (NVCC [11]). После этого в коде .NET можно создать объектную обёртку для драйвера CUDA с помощью предоставленного библиотекой класса, а дальше с помощью вызовов его методов загрузить бинарный модуль, выбрать в нём ядро по имени и запустить его.

### *2.1.2 Cloo*[2]

С помощью этой библиотеки можно выбрать некоторые из имеющихся в системе OpenCL-совместимых устройств, скомпилировать под каждое из них код на OpenCL C прямо из переменной строкового типа .NET и проинициализировать ядро, выбранное по имени функции. После этого необходимо создать буферы для аргументов ядра с указанием типа, флагов прав доступа и соответствующих им на хосте переменных .NET и выбрать порядок, в котором они будут переданы в ядро, а также создать для одного из выбранных устройств очередь команд. С помощью этой очереди можно выполнить код ядра на указанном устройстве или

---

[1] http://www.cass-hpc.com/solutions/libraries/cuda-net

[2] http://sourceforge.net/projects/cloo/

считать с него на хост данные, передав на него указатель на переменную .NET в памяти хоста.

### *2.1.3 OpenCLTemplate*[1]

Эта библиотека является надстройкой над Cloo, которая в значительной степени упрощает работу с ней, сохраняя общий порядок действий. Она позволяет автоматически выбрать тип аргументов и сделать буферы доступными для чтения и записи, а также избавляет от необходимости явно создавать очередь команд и получать указатель на объект в управляемой памяти .NET при попытке считать с выбранного устройства данные.

## 2.2 Библиотеки типов и операций

Некоторые средства интеграции вычислений на GPGPU и .NET представляют собой коллекции типичных для параллельной обработки структур данных (таких как векторы и матрицы) и готовых операций над ними. С их помощью можно довольно быстро решать стандартные задачи, представляющие собой композицию нескольких операций из коллекции, но их не слишком удобно использовать в менее распространённых и более специфических задачах.

### *2.2.1 MS Accelerator* [2]

Являясь продуктом работы Microsoft Research, Accelerator позволяет не только выполнять операции на большинстве современных GPU с помощью API DirectX 9, но и предоставляет их эффективные параллельные реализации для многоядерных CPU, использующие SSE и другие SIMD-инструкции. Существенным же препятствием перед

[1] http://code.google.com/p/opencltemplate/

[2] http://research.microsoft.com/en-us/projects/accelerator/

использованием этой библиотеки является её лицензирование по умолчанию для некоммерческого использования и отсутствие отработанного механизма лицензирования на других условиях.

### *2.2.2 FCore*[1]

Эта библиотека также предоставляет возможность эффективного выполнения операций на CPU, но поддерживает только CUDA-совместимые GPU. К её положительным сторонам стоит отнести возможность выделения памяти сразу на GPU без создания соответствующих переменных на хосте и широкое использование перегрузки операторов в F#, которое позволяет использовать при реализации функций, выполняемых на GPGPU, синтаксис, весьма близкий к классической математической нотации. FCore является коммерческим продуктом и его промышленное использование требует лицензирования.

## 2.3 Трансляция кода на лету

Весьма удобным способом производить вычисления на GPGPU является трансляция кода на языке .NET-приложения во время его работы непосредственно перед исполнением. Это позволяет разработчкам использовать привычный для них язык и средства разработки со статическим анализом и прочими возможностями, а также упрощает взаимодействие с остальным кодом, поскольку избавляет от необходимости преобразовывать данные к виду, соответствующему каким-либо соглашениям о вызовах, или к специальным типам. К тому же, в ряде случаев оказывается возможным после незначительных изменения использовать для вычислений на GPGPU код, изначально

[1] http://www.statfactory.co.uk/fcore-numerical-library/

написанный для CPU. С другой стороны, трансляция требует дополнительных накладных расходов при запуске вычислений на GPGPU, что делает средства этого типа практически неприменимыми в low-latency HPC.

### *2.3.1 Alea.cuBase*[1]

Принцип работы этой библиотеки основан на трансляции F# Quotations [7] в CUDA C. Сохраняя основные преимущества этой категории средств интеграции, она частично решает проблему накладных расходов на трансляцию при помощи сохранения её промежуточных результатов в файл (что, впрочем, усложняет её использование и СМ), но из-за привязки к CUDA не может быть использована с GPU ATI и другими устройствами. Другим недостатком Alea.cuBase является то, что это коммерческий продукт с платным лицензированием.

## 2.4 Выводы

Средства первого типа требуют наибольших трудозатрат при использовании и потому не подходят для лёгкой интеграции в существующее приложение, которая требуется в выбранном классе задач.

Выбранная для решения с помощью вычислений на GPGPU задача не является типовой и предполагает реализацию нескольких нетривиальных алгоритмов, что препятствует её решению при помощи готовых наборов операций, содержащихся в средствах второго типа.

По этой причине было решено взять за основу для решения выбранной задачи одно из средств третьего типа, предполагающих трансляцию кода

[1] https://www.quantalea.net/products/introduction/

на лету. Основанием для выбора именно Brahma.FSharp послужило то, что фреймворк OpenCL позволяет использовать более широкий набор устройств, чем CUDA, а также тот факт, что эта библиотека бесплатна и имеет открытый исходный код, что позволило внести в неё необходимые изменения и свободно использовать в работе.

# 3 Архитектура

Для того, чтобы рассказать о технических деталях данной работы, необходимо описать контекст, в котором она была выполнена, а именно устройство библиотеки Brahma.FSharp, принципы её работы и использование.

## 3.1 Код ядра и его трансляция

Для реализации функции, которая будет выполнена на целевом устройстве, используются F# Quotations. Этот выбор обусловлен тем, что для кода, написанного в них, не требуется реализовывать парсер, а его трансляцию в промежуточное представление сравнительно легко осуществить с помощью стандартного модуля, содержащего коллекцию частичных параметризованных активных шаблонов [8] для большинства существующих языковых конструкций, и возможностей сопоставления шаблонов в F#. Логика такой трансляции в Brahma.FSharp реализована в проекте Brahma.FSharp.OpenCL.Translator.

Необходимо отметить, что использовать в коде ядра можно лишь некоторое подмножество языка F#. Часть ограничений, таких, как, например, отсутствие поддержки рекурсии, вызвана соответствующими ограничениями OpenCL C. Другим важным ограничением является то, что в коде ядра доступны только числовые типы данных и массивы, а также служебные классы, содержащие специфическую для OpenCL информацию, к которой относятся общее количество одновременно исполняемых экземпляров ядра (потоков) и размер рабочей группы

(набора потоков, имеющих возможность разделения локальной памяти и использования синхронизации).

Поскольку OpenCL C обладает возможностями и особенностями, не имеющими прямых аналогов в языке F#, для их поддержки пришлось добавить несколько служебных функций и операторов (например, для атомарных арифметический операций и присваиваний) в отдельном модуле в проекте Brahma.FSharp.OpenCL.Extensions с атрибутом [AutoOpen], что позволяет не заботиться об указании зависимостей на него при использовании библиотеки. Применение этого решения было минимизировано, поскольку излишнее количество служебных функций в значительной степени усложняет изучение возможностей библиотеки перед её использованием, а решение задач, такими функциями не покрытых, делает менее интуитивно понятным.

## 3.2 Промежуточное представление

В результате трансляции кода ядра на F# получается абстрактное синтаксическое дерево (Abstract Syntax Tree, AST), которое описано в проекте Brahma.FSharp.OpenCL.AST. На практике элементами первого уровня в нём обычно оказываются несколько #pragma-выражений, специфичных для OpenCL C, и объявление функции ядра. К другим возможным элементам первого уровня относятся объявления типов.

Объявление типов содержит список аргументов, тип возвращаемого значения и тело функции, которое является любым экземпляром класса Statement, служащего общим предком для классов всех поддерживаемых языковых конструкций.

## 3.3 Генерация кода

Для получения окончательного результата трансляции используется рекурсивных обход AST, логика которого описана в проекте Brahma.FSharp.OpenCL.Printer. В качестве целевого языка выбран OpenCL C, поскольку это даёт возможность использовать не только GPGPU, но и многие другие устройства, а OpenCL является открытым стандартом.

В результате такого обхода конструируется экземпляр класса Layout, входящего в коллекцию библиотек и инструментов F# PowerPack[1]. Для печати отформатированного кода из него используется готовая функция из той же коллекции.

## 3.4 Использование

Для того, чтобы использовать библиотеку для вычислений на массово-параллельном устройстве, необходимо создать очередь команд для выбранного устройства и скомпилировать под него F# Quotation с кодом ядра. Результат компиляции вернёт функцию с той же сигнатурой, что и функция ядра, в которую необходимо передать аргументы, а также команду, которую после этого необходимо поместить в очередь. При первом запуске команды для массивов, являющихся её аргументами, выделяются в видеопамяти буферы необходимого размера, соответствие которых объектам .NET в памяти хоста поддерживается до их явного закрытия. Для получения результатов используется пустой аргумент и специальная команда копирования содержимого буфера с видеокарты на хост, а для переиспользования аргументов - специальные

[1] http://fsharppowerpack.codeplex.com/

команды копирования массивов в соответствующий буфер, которые запускаются в той же очереди.

Таким образом, при многократном запуске одного и того же кода на GPGPU с разными аргументами трансляцию, компиляцию и некоторые другие подготовительные операции осуществлять каждый раз не нужно, что делает накладные расходы на них пренебрежимо малыми при обработке достаточного количества запусков.

# 4 Особенности реализации

Поскольку техническая часть данной работы состояла из двух подзадач, первая из которых заключалась в доработке библиотеки Brahma.FSharp, а вторая - в решении с её помощью задачи поиска подстрок в строке, то и особенности реализации имеет смысл явно разделить на две соответствующие части.

## 4.1 Доработка Brahma.FSharp

Реализация новой функциональности библиотеки представляет собой преимущественно архитектурную задачу, поскольку важно было как можно меньше отклоняться от выбранного изначально при её разработке подхода и максимизировать удобство её использования.

В ходе этой части работы была реализована поддержка следующих возможностей OpenCL C:

- Инициализация массивов;
- Синхронизация;
- Работа с локальной памятью.

### *4.1.1 Инициализация массивов*

Несмотря на то, что F# - мультипарадигменный язык программирования с поддержкой функциональной, объектно-ориентированной и императивной парадигм, а C, лежащий в основе OpenCL C, - процедурный, большинству языковых конструкций выбранного подмножества языка F# можно найти в C прямые соответствия, а

трансляцию между ними осуществить простой конвертацией конструкций F# в соответствующие узлы AST.

Одним из случаев, когда это не так, оказалась инициализация массивов, реализация поддержки которой была одной из задач данной работы. С точки зрения языка F# и F# Quotations инициализация числового массива нулями - вызов метода Arrays.zeroCreate, в то время как в С для этого имеется специальный синтаксис, заключающийся в перечислении значений через точку с запятой в фигурных скобках.

Выбранным решением стало создание специальных узлов AST для инициализации массивов, а именно общего для них абстрактного суперкласса ArrayInitializer, хранящего длину массива, и его наследника ZeroArray. При генерации из него кода просто возвращается строка “{0}”.

### *4.1.2 Синхронизация*

OpenCL C предлагает собственный механизм синхронизации потоков внутри рабочей группы, который заключается в вызове специальной функции, внутри которой потоки блокируются до тех пор, пока все не окажутся там, после чего все одновременно разблокируются.

В F# соответствующей конструкции нет, поэтому было решено создать в модуле для служебных расширений функцию без аргументов barrier, которая, будучи вызванной, не оказывает побочных эффектов и возвращает unit, а при трансляции преобразуется в специальный узел AST Barrier, из которого всегда генерируется код "barrier(CLK_LOCAL_MEM_FENCE)".

### *4.1.3 Локальная память*

Локальная память OpenCL является специфичным для этого стандарта понятием и тоже не имеет аналогов в .NET. Для того, чтобы сделать возможной поддержку выделения для переменных локальной памяти, было решено реализовать в модуле для служебных расширений функцию local с одним аргументом, которая, будучи вызванной, просто возвращает свой аргумент, что позволяет сохранить соответствие типов, если обернуть значение, которым инициализируется переменная при объявлении, вызовом этой функции.

При трансляции конструкции let, в которой объявляется переменная с начальным значением, равным результату вызова функции local от какого-либо выражения, соответствующему экземпляру класса VarDecl устанавливается флаг isLocal, а для дальнейшей трансляции используется выражение-аргумент.

При генерации кода из VarDecl в случае, когда флаг установлен, перед объявлением переменной генерируется модификатор “__local”, а значение, которым переменная инициализировалась, игнорируется, поскольку OpenCL C не поддерживает инициализации переменных в локальной памяти при их объявлении.

## 4.2 Задача поиска подстрок в строке

Несмотря на то, что эта задача известна давно и способы её решения хорошо изучены, при реализации решения с помощью OpenCL возникли дополнительные трудности, чаще всего связанные с особенностями модели вычислений и GPGPU.

#### *4.2.1 Представление результатов*

С учётом ограниченного набора доступных типов представлять результаты необходимо в виде массива чисел, поскольку других структур данных нет. В нём было бы логично сохранить для каждого индекса номер шаблона, с которым найдено совпадение с началом в соответствующей позиции, но тогда возникает неопределённость в случае, когда таких шаблонов несколько.

Очевидно, что в этом случае для любых двух из них они имеют различную длину и при этом более короткий является префиксом более длинного. Ясно также, что есть совпадение со всеми шаблонами, которые являются префиксами самого длинного шаблона, с которым есть совпадение. Таким образом, для однозначного задания набора шаблонов, с которыми есть совпадение в заданной позиции, достаточно номера самого длинного из них.

Поскольку большинство алгоритмов перебирают шаблоны в некотором порядке хотя бы на этапе инициализации, удобно оказалось отсортировать шаблоны в порядке возрастания длины, чтобы при переборе просто запоминать номер последнего, с которым было совпадения. Ясно, что из двух шаблонов равной длины совпадение может быть не более чем с одним и один из них не может являться префиксом другого, поэтому порядок шаблонов одинаковой длины при сортировке не имеет значения.

Рассмотрим следующий пример. Пусть входная строка - “acbabca”, а искомые шаблоны - “ab” (0), “ca” (1), “bc” (2) и “abc” (3). Тогда результатом поиска шаблонов в строке будет массив из чисел (-1, -1, -1,

3, 2, 1, -1), поскольку в 4-ой позиции, считая от 1, есть совпадения с шаблонами 0 и 3, в 5-ой - с шаблоном 2, а в 6-ой - с шаблоном 1. В других позициях совпадений нет, что обозначается числом -1.

## *4.2.2 Дробление входной строки*

Несколько проблем были вызваны тем, что прочитать входную строку в память нельзя как из-за ограничений на её использование, так и просто физически, поэтому читать её приходится по частям, а каждую часть, в свою очередь, параллельно обрабатывать большим числом потоков.

### 4.2.2.1 Последняя итерация

В силу ограничений OpenCL количество потоков должно быть кратно размеру рабочей группы и равно произведению её размера на количество рабочих групп. При этом, большинство реализаций ядра оптимизированы для конкретных значений этих параметров и для постоянной длины участка, обрабатываемого одним потоком, чтобы избежать лишних вычислений границ и ветвлений, поэтому каждый запуск GPGPU обрабатывает отрезок входной строки одинаковой длины. При этом общая длина входной строки может быть произвольной, в результате чего на последней итерации может остаться часть, длина которой отлична от той, на которую рассчитано ядро.

Для решения этой проблемы предложено дополнить последнюю часть входной строки до нужной длины произвольными (на практике, оставшимися от чтения на предпоследней итерации) данными, а лишние результаты (далее будем называть их некорректными) отсечь на хосте с помощью следующего алгоритма.

Для начала, построим бор из шаблонов, добавляя их туда в порядке возрастания длины (т.е. в их естественном порядке, поскольку они уже отсортированы). При этом для каждого шаблона будем хранить номер самого длинного шаблона из набора, являющегося префиксом данного, или -1, если такого шаблона нет. Изначально для всех шаблонов сохраним -1. Теперь заметим, что, если при добавлении шаблона в бор какой-либо его шаблон-префикс в бор уже добавлен, то мы пройдём лист, соответствующий этому шаблону. Поскольку мы добавляем шаблоны в порядке возрастания длины, то на момент добавления любого из них все его префиксы уже добавлены, поэтому достаточно просто обновлять сохранённый индекс при прохождении каждого листа, так как листы будут, очевидно, пройдены в порядке возрастания длины соответствующих шаблонов.

Теперь заметим, что проверять корректность совпадений необходимо только в последней 31-ой позиции, поскольку до них любое совпадение гарантированно заканчивается не дальше последнего символа, даже если длина соответствующего шаблона максимальна и равна 32. Для того, чтобы проверить позиции, в которой есть совпадение, необходимо либо принять её, если шаблон заканчивается до границы входной строки, либо, иначе, повторить проверку для максимального префикса шаблона. Если на некоторой итерации префикса нет, нет и корректного совпадения в данной позиции.

#### 4.2.2.2 Граница итераций

Возможна ситуация, когда начало совпадения находится в участке входной строки, прочитанном на одной итерации, а его конец - в участке,

прочитанном на следующей. Ясно, что независимая обработка двух участков в таком случае не даст возможности найти это совпадение.

Было выбрано следующее решение этой проблемы. Перед чтением всех участков входной строки, кроме первого, последние 32 символа копируются из конца буфера в начало, а новые данные начинают заполнять буфер с 32-ой позиции, считая от 0. Таким образом, на соседних итерациях обрабатываются части входной строки, пересекающиеся по 32-ум символам, вследствие чего любое совпадение лежит целиком хотя бы в одной из частей.

Для того, чтобы не учитывать попавшие в пересечение совпадения дважды, верхняя граница всех участков входной строки, кроме последнего, искусственным образом уменьшается на 32 при помощи отсечения совпадений алгоритмом, предложенным для отсечения совпадений, выходящих за границу входной строки.

#### 4.2.2.3 Границы потоков

Аналогичная предыдущей проблема возникает при обработке одного участка входной строки между участками, которые обрабатываются разными потоками.

Для решения этой проблемы в большинстве алгоритмов достаточно позволить потокам читать любые символы, которые им нужны, но строго разделить участки результирующего массива, которые потоки обновляют. На стыке участков двух потоков оба эти потока могут прочитать одни и те же символы, но, поскольку они ищут совпадения, начинающиеся в разных позициях, они не найдут одно и то же дважды.

В некоторых алгоритмах (таких, как вариации алгоритма Ахо-Корасик) нет однозначного номера позиции, совпадения с началом в которой алгоритм ищет в данный момент, поэтому для них это решение не подходит. В таких случаях было решено прочитать первых 64 символа, чтобы гарантированно найти все совпадения, начинающиеся в первых 32-ух позициях, и синхронизировать потоки, а после обновлять результат только в том случае, если уже имеющееся там значение меньше того, которое поток собирается туда записать.

### *4.2.3 Параллельные чтение и обработка*

Для большинства алгоритмов, исполняемых на CPU, время обработки входной строки значительно превосходит время её чтения даже со старых и медленных жёстких дисков, поэтому им можно пренебречь. Для наиболее производительных решений на GPGPU оказалось, что время чтения уже сравнимо с временем обработки, поэтому появился смысл в том, чтобы сэкономить большую его часть, читая следующий участок входной строки параллельно с обработкой текущего.

Осложняет реализацию таких паралелльных действий ограничение на расход памяти на хосте, поскольку из-за него нельзя просто завести новый буфер и прочитать в него следующий участок сразу, как предыдущий был отправлен на обработку, а необходимо переиспользовать старый.

В результате было решено запускать обработку асинхронно, реализовав её в F# Asynchronous Workflow, запустив его в виде задачи (экземпляра класса Task) с помощью метода Async.StartAsTask и вернув эту задачу.

Цикл чтения и обработки файла устроен следующим образом. Сначала в буфер читается очередной участок входной строки, а после окончания чтения основной поток дожидается окончания обработки предыдущего участка, получая результат выполнения задачи с помощью метода Wait(), если запуск ядра не первый и предыдущий участок существует. Затем поток асинхронно запускает очередную задачу.

## 4.3 Вычисление оптимальных параметров

Оптимальные значения таких параметров, как размер рабочей группы и размер участка, обрабатываемого одним потоком, можно подобрать эмпирическим путём, но при их выбранных значениях необходимо вычислить максимально возможные количество рабочих групп и, прямо пропорциональный ему, размер буфера, поскольку с их ростом растёт степень параллелизма и, соответственно, итоговая производительность вычислений.

Введём несколько необходимых величин:

- Пусть $H = 256 * 2^{10}$ - размер доступной памяти на хосте.
- Пусть G - размер доступной памяти на видеокарте, который можно узнать как параметр CL_DEVICE_MAX_MEM_ALLOC_SIZE OpenCL.
- Пусть A - размер аргументов ядра, отличных от входной строки и массива для результатов. Он был вычислен для каждого алгоритма при помощи точной оценки на размер аргументов-массивов (которая следует из того, что все они имеют числовой тип) и оценки сверху константой всех аргументов простых типов, после чего было выбрано максимальное значение. Ясно, что эти аргументы будут храниться как на хосте, так и на видеокарте, и в силу своих числовых типов будут требовать одинакового количества памяти.

- Пусть D - размер дополнительных временных данных, нужных для инициализации алгоритмов и обработки данных на хосте, который можно вычислить тем же способом, что и A.
- Пусть S - размер участка входных данных, который обрабатывается одной рабочей группой, равный произведению количества потоков в группе на длину участка одного потока.
- Пусть R - размер массива результатов для одной группы, тогда ясно, что $R = 2 * S$, поскольку для хранения номера одного из 512 шаблонов или -1 требуется 2 байта.
- Пусть M - размер памяти, доступной для хранения входной строки и результатов.
- Пусть X - максимальное возможное количество рабочих групп.
- Пусть L - максимальная возможная длина буфера для чтения входной строки.

Поскольку входная строка и результаты должны помещаться в видеопамяти, $M \le G - A$. Поскольку они же должны помещаться в памяти хоста, $M \le H - A - D$. Таким образом, $M = \min \{G - A, H - A - D\}$. В таком случае, $X = [M / (R + S)] = [M / (3S)] = [\min \{G - A, H - A - D\} / (3S)]$. Ясно, что $L = X * S$.

К примеру, при $G = 512 * 2^{10}$ на видеокарте, использованной при тестировании, с учётом полученных значений A и D, и при выбранном значении $S = 512 * 1024 = 524288$ (рабочая группа состояла из 512-ти потоков, каждый из которых обрабатывал участок в 1024 байта), были использованы $X = 154$ группы и $L = 80740352$ байт (77 мегабайт).

# 5 Алгоритмы

Некоторые из реализованных алгоритмов не могли оказаться эффективным решением выбранной задачи, но, в силу относительной лёгкости их реализации, хорошо подошли для того, чтобы на их примере научиться пользоваться библиотекой, а также получить первую информацию об особенностях и производительности GPGPU в задаче поиска подстрок в строке. К тому же, в ходе тестирования было обнаружено, что в некоторых случаях алгоритмы с разной асимптотической сложностью показывают близкие результаты, поскольку производительность операций над данными в памяти разных типов на GPGPU существенно отличается, из-за чего алгоритм, делающий больше (и асимптотически, и по абсолютному значению) операций, но над более быстрой памятью, может оказаться более или столь же производительным при всех реальных размерах входных данных, чем асимптотически более эффективный.

## 5.1 Полный перебор

Для каждой позиции во входной строке перебираются шаблоны и для каждого из них последовательно сравниваются символы шаблона с символами входной строки с таким же смещением относительно выбранной позиции, как у символа шаблона от его начала, начиная с первого символа. Если символы одинаковые, переходим к следующему, если его нет - фиксируем совпадение, а если символы разные - переходим к следующему шаблону.

## 5.2 Алгоритм Рабина-Карпа

Основная идея этого алгоритма - перед тем, как начинать сравнивать шаблон с подстрокой входной строки посимвольно, сравнить их хеши [2]. В качестве хеш-функции была выбрана сумма кодов всех символов по модулю 256, поскольку она обладает свойством аддитивности по строке (если считать сумму строк определённой как их конкатенацию). Из этого свойства очевидным образом следует тот важный факт, что хеш суффикса строки равен разности хеша строки и хеша соответствующего префикса в кольце вычетов по модулю 256. Для сложения и вычитания в этом кольце достаточно использовать обычные арифметические операторы и тип byte.

Для каждого потока будем хранить массив из 32 байт, в котором будем поддерживать хеши подстрок входной строки соответствующей длины с началом в текущей позиции. Для того, чтобы вычислить их начальные значения, в первый элемент достаточно сохранить код первого символа, а в каждый следующий - сумму предыдущего элемента и код символа с соответствующим смещением.

Для пересчёта таких хешей для следующей позиции достаточно для первых 31-ой подстроки сохранить в этом массиве разность следующего элемента массива и кода пройденного символа, а потом для последней подстроки - сумму предпоследнего элемента массива и кода последнего символа последней подстроки.

### *5.2.1 Оптимизации*

Этот алгоритм допускает несколько оптимизаций с помощью реализованных в рамках этой работы новых возможностей библиотеки:

1. Выделение для хешей подстрок массива в приватной памяти потока.
2. Копирование хешей и длин шаблонов в локальную память рабочей группы. Может быть использована вместе с предыдущей.
3. Копирование шаблонов в локальную память. От этой оптимизации пришлось отказаться, поскольку на некоторых видеокартах для неё локальной памяти не достаточно, а её тестирование показало, что она не улучшает производительности.

## 5.3 Полный перебор с хеш-таблицей

Основная идея этого алгоритма в том, чтобы перебирать подстроки входной строки и искать в хеш-таблице шаблоны той же длины с тем же хешем с целью посимвольного сравнения. Он полностью заимствует механизм работы с хешами подстрок из предыдущего алгоритма.

Поскольку в качестве аргументов ядра могут быть использованы только числовые типы и массивы, хеш-таблица была представлена как массив из 256 начал списков, в каждом из которых хранятся шаблоны с соответствующим значением хеш-функции [4]. Алгоритм перебирает подстроки, начинающиеся в каждой позиции, в порядке возрастания их длины, поэтому совпадение с самым длинным шаблоном будет найдено последним и сохранится в результатах.

### *5.3.1 Оптимизации*

Для этого алгоритма были разработаны и протестированы следующие оптимизации:

1. Выделение для хешей подстрок массива в приватной памяти потока. Заимствована из предыдущего алгоритма.

2. Копирование длин шаблонов в локальную память рабочей группы. Также заимствована из предыдущего алгоритма.
3. Копирование хеш-таблицы в локальную память. Аналогична копированию хешей из предыдущего алгоритма и также оказалась полезной.
4. Разворачивание циклов, пересчитывающих хеши подстрок и перебирающих подстроки при сравнении, с шагом 4. Производительность улучшена лишь незначительно.

## 5.4 Алгоритм Ахо-Корасик

В начале алгоритма используется построение бора из шаблонов [3], который в нашем случае уже построен для отсечения совпадений на хосте, поэтому он был переиспользован.

Классический алгоритм Ахо-Корасик оказался малоприменимым в условиях GPGPU, поэтому в него пришлось внести ряд изменений. Основной проблемой оказалось ленивое вычисление суффиксных ссылок и переходов в суффиксном автомате, поскольку оно требует невозможных в OpenCL рекурсивных вызовов, а также добавило бы риск одновременного изменения разделяемой памяти, в которой находится автомат, из большого числа потоков.

Был разработан алгоритм, позволяющий проинициализировать все данные, вычисляемые в классической реализации лениво, перед запуском алгоритма. Для этого суффиксный автомат обходится в глубину по рёбрам бора, при этом при попадании в каждую вершину из неё строятся рёбра автомата для каждого из 256 символов (те, которым

соответствует ребро бора, просто копируются, а остальные вычисляются рекурсивно известным способом).

### *5.4.1 Оптимизации*

Для полученной в результате версии алгоритма Ахо-Корасик были применены следующие оптимизации:

1. Копирование длин шаблонов в локальную память. Заимствована из двух предыдущих алгоритмов.
2. Построение суффиксного автомата сразу в одномерном массиве. Причина существования этой оптимизации в том, что для каждой вершины автомата необходимо хранить 256 рёбер из неё, что естественно представить в виде двумерного массива, но который, в силу отсутствия поддержки двумерных массивов как аргументов ядра, приходилось реорганизовывать в виде одномерного. Впрочем, большого эффекта на производительность инициализации алгоритма это не оказало.
3. Вычисление функции выхода вместо суффиксных ссылок. Для того, чтобы найти все совпадения после очередного перехода в автомате, нужно не только проверить, не является ли его текущая вершина листом, но и спуститься от неё до корня по суффиксным ссылкам, поскольку среди таких вершин может оказаться лист, соответствующий другому, более короткому, совпадению с шаблоном. Оптимизация же заключается в том, чтобы для каждой вершины заранее вычислить ребро в ближайший лист, встречающийся в пути от неё до корня по суффиксным ссылкам, и переходить сразу по таким рёбрам, пропуская вершины, заведомо не являющиеся листом какого-либо из шаблонов.

# 6 Апробация

Для финального тестирования различных алгоритмов решения задачи поиска подстрок в строке использовалась следующая конфигурация оборудования:

- Четырёхъядерный процессор Intel Core i7-3770K с тактовой частотой 3500 МГц;
- Видеокарта ASUS GeForce GTX670-DC2-2GD5 с 2 ГБ GDDR5;
- Оперативная память Corsair Vengeance CMZ16GX3M4X1866C9 4x4ГБ DDR3 с частотой 1866 МГц;
- Жёсткий диск SATA Seagate Barracuda ST2000DM001 со скоростью вращения 7200 оборотов в минуту.

## 6.1 Алгоритм Рабина-Карпа

Тестирование алгоритма Рабина-Карпа проводилось на случайно сгенерированном входном файле длиной 156 мегабайт и случайно сгенерированных шаблонах с минимальной длиной 2 байта. В файле присутствовали совпадения с шаблонами в 32475 позициях. Использовались следующие его реализации:

1. Алгоритм Рабина-Карпа для CPU.
2. Алгоритм Рабина-Карпа для GPGPU.
3. Алгоритм Рабина-Карпа для GPGPU с использованием массивов в приватной памяти потока и локальной памяти OpenCL.

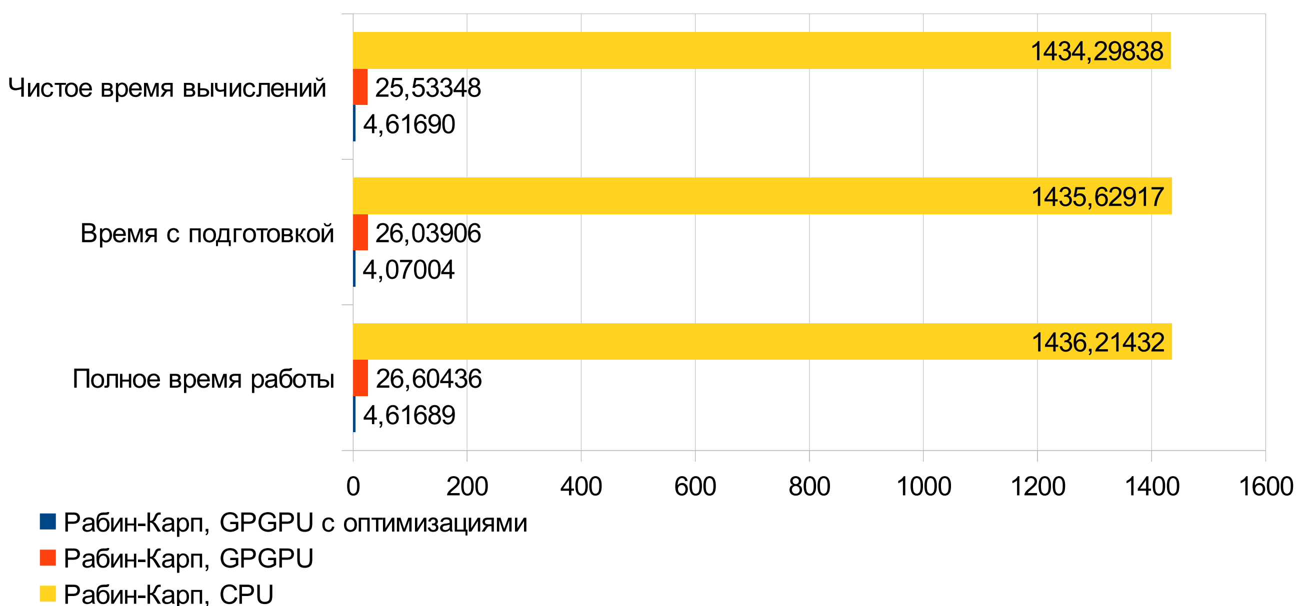
Для исполнения на GPGPU было взято две реализации алгоритма, поскольку первая из них не использовала возможностей библиотеки Brahma.FSharp, реализованных в рамках данной работы, а вторая была

существенно оптимизирована с их помощью, что позволило оценить влияние этих возможностей на производительность.

### 6.1.1 Результаты

Время с подготовкой помимо времени вычислений включает в себя время, затраченное на трансляцию кода и инициализацию алгоритма (например, на вычисление хешей шаблонов), а полное время работы - ещё и время, затраченное на чтение входного файла и обработку результатов на хосте (отсечение лишних и подсчёт корректных совпадений).

Все результаты измерений в этой и последующих диаграммах представлены в секундах.




### 6.1.2 Выводы

Исполнение алгоритма на GPGPU без оптимизаций позволило достичь ускорения с учётом подготовки примерно в 55,13 раза по сравнению с однопоточным исполнением на CPU, тогда как ранее эта величина не превосходила 24 в разных алгоритмах поиска подстрок в строке [5, 6].

Использование локальной и приватной памяти позволило ускорить решение задачи ещё в 6,4 раза, увеличив ускорение по сравнению с CPU до 352,73 раза. Ускорение чистых вычислений и полного цикла с чтением и обработкой на хосте составило 355,05 и 311,08 раза соответственно.

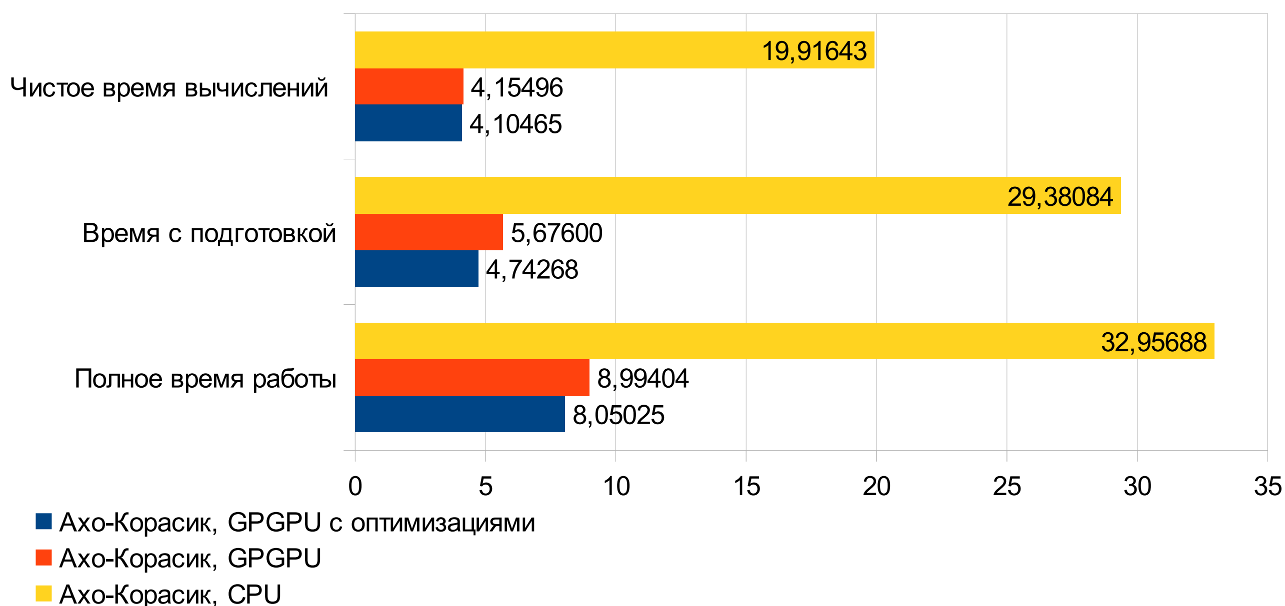
В дальнейшем эти значения будут использоваться для аппроксимации времени работы алгоритма на CPU на больших входных файлах.

## 6.2 Алгоритм Ахо-Корасик

Тестирование алгоритма Рабина-Карпа проводилось на случайно сгенерированном входном файле длиной 1 гигабайт и случайно сгенерированных шаблонах с минимальной длиной 2 байта. В файле присутствовали совпадения в 214993 позициях. Использовались следующие его реализации:

1. Алгоритм Ахо-Корасик для CPU.
2. Алгоритм Ахо-Корасик для GPGPU.
3. Алгоритм Ахо-Корасик для GPGPU с построением суффиксного автомата в одномерном массиве и вычислением функции выхода.

### *6.2.1 Результаты*




### *6.2.2 Выводы*

Ускорение с помощью наиболее оптимизированной версии алгоритма по сравнению с однопоточным исполнением на CPU составило 4,85 раза для чистого времени вычислений, 6,19 раза для времени с подготовкой и 4,09 раза для времени полного цикла. Вне зависимости от способа измерений, этот результат превосходит полученный для алгоритма Ахо-Корасик ранее, равный 3,1 раза [6].

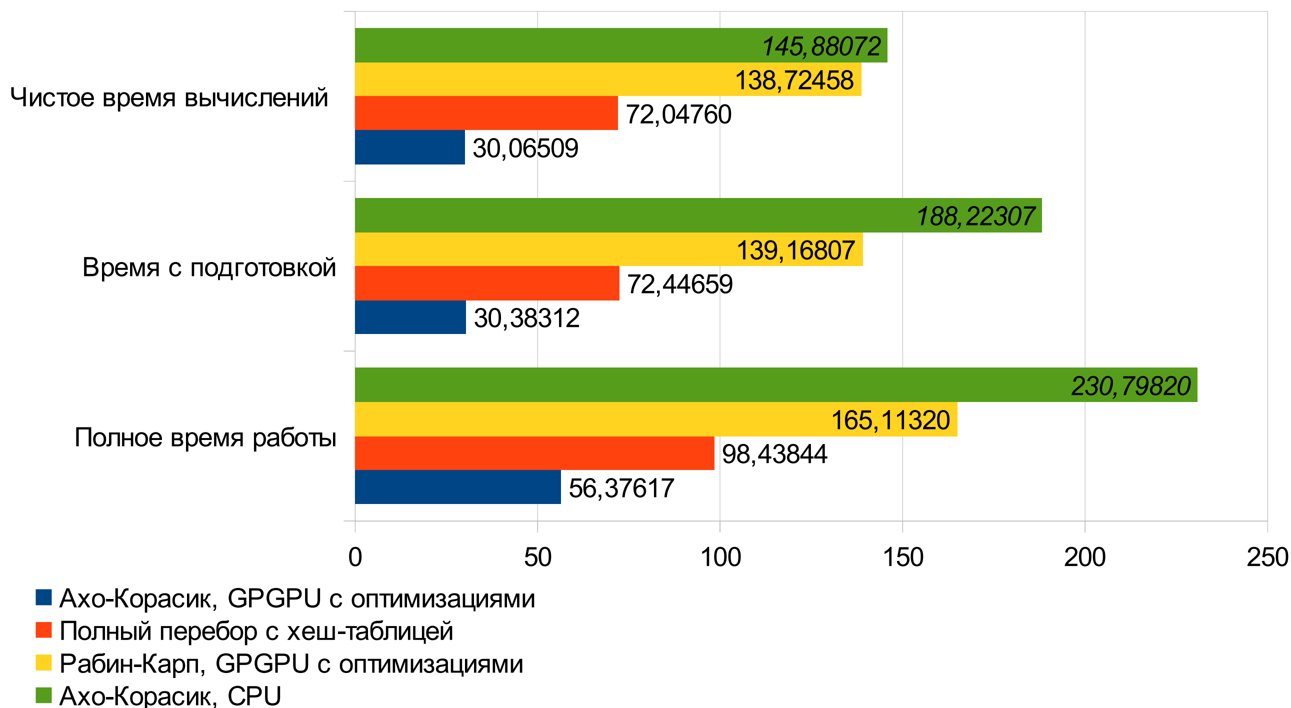
## 6.3 Сравнение алгоритмов

Для сравнения производительности разных алгоритмов использовался случайно сгенерированный файл размером 8 гигабайт и случайно сгенерированные шаблоны с минимальной длиной в 2 байта. В файле присутствовали совпадения в 1720392 позициях.

### *6.3.1 Результаты*

Для оценки времени работы алгоритмов, исполняющихся на CPU, были использованы значения, полученные в предыдущих измерениях,

поскольку ясно, что соотношение времени работы сохраняется в силу его линейности по размеру входной строки в силу того, что вхождения случайных шаблонов распределены по случайно сгенерированному файлу равномерно. Вычисленные таким образом приближённые значения помечены выделены в диаграмме курсивом.




## 6.4 Тестирование на реальных данных

С целью выяснения реальной эффективности полученного решения был проведён эксперимент в условиях, близких к тем, в которых эта задача возникает в цифровой криминалистике. Для этого был взят бывший в использовании, а после отформатированный жёсткий диск номинальным объёмом 250 гигабайт и организовано его низкоуровневое чтение с помощью WinAPI и Platform Invocation Services (PInvoke [12]). Каждый шаблон являлся сигнатурой файла - последовательностью байтов, с которой файл определённого формата обязательно должен начинаться. Были выбраны 50 сигнатур файлов распространённых форматов [13], преимущественно предназначенных для хранения изображений, видео,

звуков, текстовых документов и архивов. Также были взяты подмножества этого набора сигнатур размером 10 и 30. Полный список набора из 30-ти сигнатур приведён в приложении А. На этих входных данных были запущены самые быстрые из имеющихся реализаций алгоритма Ахо-Корасик, полного перебора с хеш-таблицей и алгоритма Рабина-Карпа.

Использовалась следующая конфигурация оборудования:

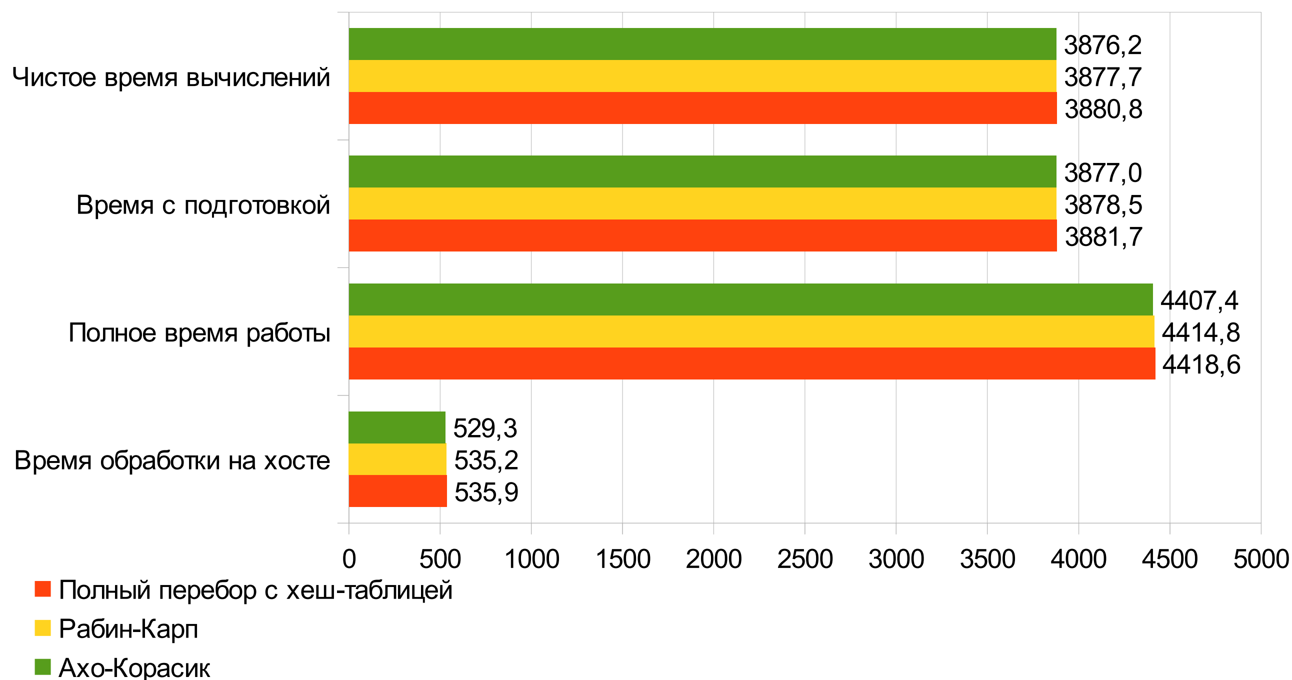
- Процессор Intel Celeron 450 с тактовой частотой 2200 МГц;
- Видеокарта с графическим процессором NVIDIA GeForce GTX 560 Ti и 2 ГБ GDDR5;
- Оперативная память Kingston KVR800D2N6/1G 2x1ГБ DDR2 с частотой 800 МГц;
- Жёсткий диск SATA Seagate Barracuda ST3250410AS со скоростью вращения 7200 оборотов в минуту.

### *6.4.1 Результаты*

Поскольку использовался большой объём входных данных, решено было отдельно измерить время, затраченное на обработку результатов на хосте. В диаграммах указано среднее время в трёх запусках.

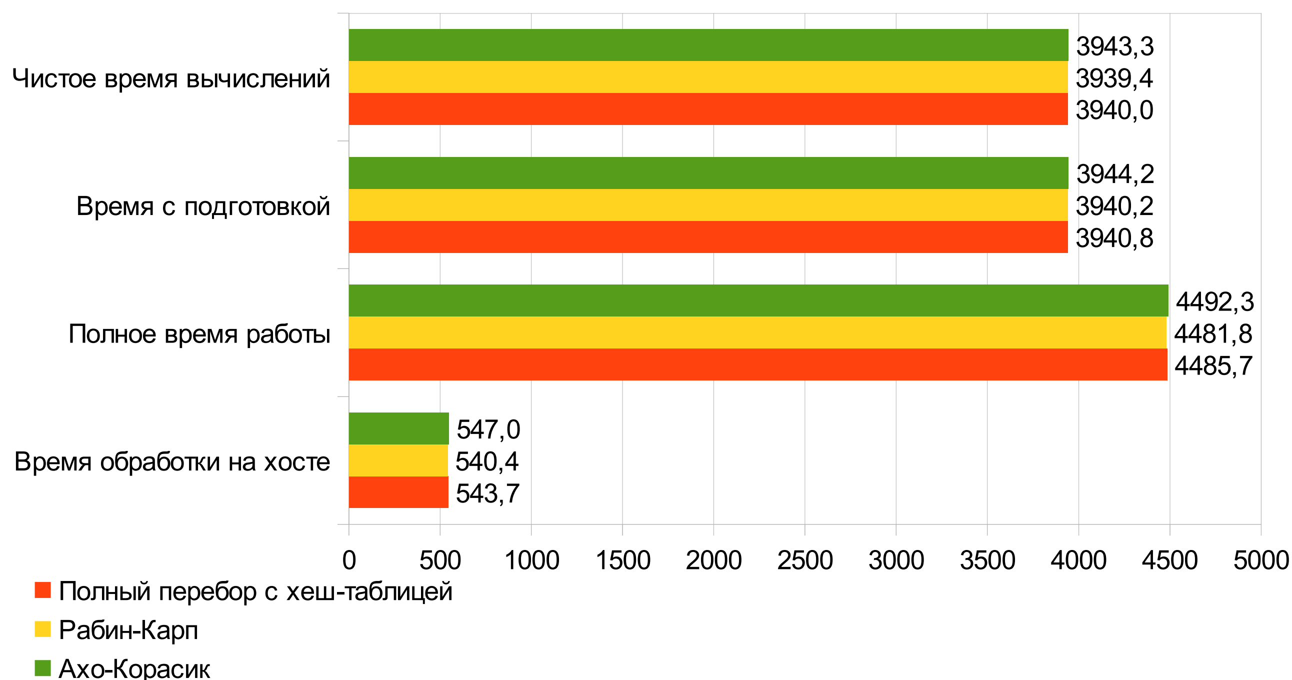
#### 6.4.1.1 Поиск 10 шаблонов

Всего на диске было найдено 3068458 совпадений.



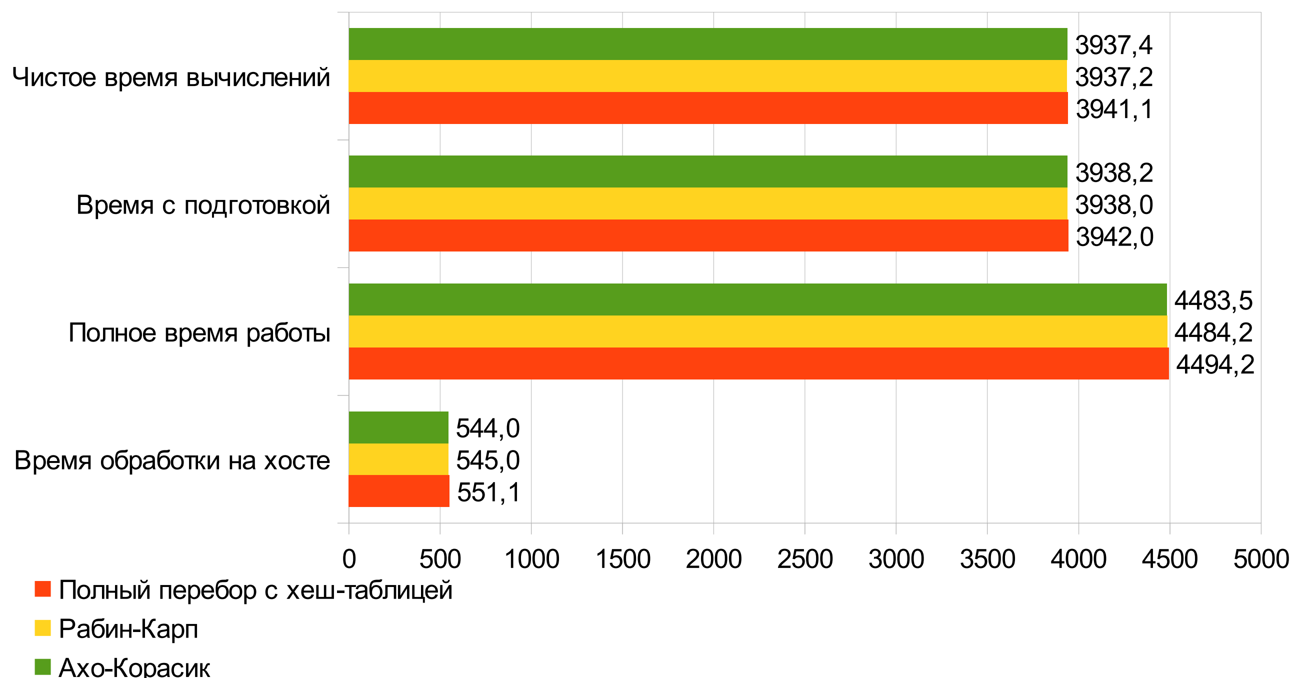

#### 6.4.1.2 Поиск 30 шаблонов

Всего на диске было найдено 4076639 совпадений. Пример подробных результатов работы, содержащих точное количество совпадений с каждым из шаблонов, приведён в приложении А.



#### 6.4.1.3 Поиск 50 шаблонов

Всего на диске было найдено 167106721 совпадений.




### *6.4.2 Выводы*

Легко видеть, что накладные расходы на трансляцию кода при обработке большого объёма данных пренебрежимо малы, поскольку время работы с подготовкой, куда помимо трансляции входит ещё и специфичная для алгоритма инициализация (такая, как построение суффиксного автомата в случае алгоритма Ахо-Корасик), больше чистого времени вычислений лишь на величину порядка секунды. В то же время, подобные накладные расходы действительно препятствуют использованию библиотеки при необходимости low-latency HPC.

Стоит обратить внимание на то, что при сравнительно небольшом (до нескольких десятков) числе шаблонов выбор алгоритма перестаёт иметь сколь-нибудь существенное значение, поскольку разница в среднем времени работы алгоритмов оказывается не больше, чем разница во времени работы одного алгоритма в разных запусках. Вероятно, при

достаточно быстрой обработке данных на GPGPU большее значение начинает иметь работа по копированию данных и результатов из памяти хоста в видеопамять и обратно.

# Заключение

## Результаты

В рамках данной работы были получены следующие результаты:

1. Изучены особенности GPGPU. Исследовано влияние на производительность вычислений использования видеопамяти разных типов, количества потоков и размера участка данных, обрабатываемого одним потоком.
2. Изучены существующие средства интеграции .NET-приложений и вычислений на GPGPU. Выявлены общие особенности средств использования нативного кода (CUDA.NET, Cloo, OpenCLTemplate), библиотек структур данных и операций (MS Accelerator, FCore), трансляции кода на лету (Alea.cuBase). Сделан их обзор и проанализированы их преимущества и недостатки по сравнению с Brahma.FSharp.
3. Доработана библиотека Brahma.FSharp. Реализована возможность работы с массивами, локальной памятью OpenCL и средствами синхронизации. Внесены исправления и доработана функциональность.
4. Библиотека Brahma.FSharp использована для решения задачи поиска подстрок в строке. Продемонстрировано влияние её новых возможностей на производительность различных алгоритмов (полного перебора, алгоритма Рабина-Карпа, полного перебора с хеш-таблицей, алгоритма Ахо-Корасик). По результатам апробации библиотека отлажена и дополнена её функциональность.

## Дальнейшее развитие

В будущем возможно развитие работы в следующих направлениях: дальнейшее расширение функциональности библиотеки Brahma.FSharp и создание на её основе библиотеки стандартных функций для использования в .NET приложениях.

В первое направление входит, например, добавление поддержки дополнительных типов данных, реализация трансляции дополнительных математических функций и логических операторов. Также возможна разработка генерации кода для инициализации переменных в локальной памяти OpenCL с учётом размера рабочей группы и для инициализации массивов с помощью метода Array.init.

В рамках создания библиотеки стандартных функций можно реализовать операции над массивами и списками (такие, как сортировку), а также над матрицами (например, их умножение). Кроме того, в неё можно включить реализованные в ходе данной работы алгоритмы над строками.

Полученный в ходе решения задачи поиска подстрок в строке результат может быть внедрён в продукт Belkasoft Evidence Center, который является инструментом для цифровой криминалистики.

# A Шаблоны и совпадения

| Шаблон | Тип файла | Совпадения |
|---|---|---|
| 42 4D | BMP | 2989273 |
| 49 44 33 | MP3 | 24426 |
| 46 57 53 | SWF | 14455 |
| 43 57 53 | SWF | 53388 |
| 49 20 49 | TIFF | 56862 |
| 00 00 01 BA | MPG, VOB | 76737 |
| 50 4B 07 08 | ZIP | 18530 |
| 38 42 50 53 | PSD | 196 |
| 50 4B 05 06 | ZIP | 6655 |
| 49 49 2A 00 | TIFF | 25779 |
| 25 50 44 46 | PDF | 2085 |
| 50 4B 03 04 | ZIP | 453978 |
| 37 7A BC AF 27 1C | 7z | 116 |
| 47 49 46 38 37 61 | GIF | 3819 |
| 47 49 46 38 39 61 | GIF | 111941 |
| 4A 41 52 43 53 00 | JAR | 17 |
| 52 61 72 21 1A 07 00 | RAR | 1118 |
| 50 4B 03 04 14 00 06 00 | DOCX | 4845 |
| 66 4C 61 43 00 00 00 22 | FLAC | 0 |
| 89 50 4E 47 0D 0A 1A 0A | PNG | 214853 |
| E3 10 00 01 00 00 00 00 | INFO | 53 |
| D0 CF 11 E0 A1 B1 1A E1 | DOC | 1497 |
| 50 4B 03 04 14 00 08 00 08 00 | JAR | 15095 |
| 00 00 00 18 66 74 79 70 33 67 70 35 | MP4 | 0 |
| 00 00 00 14 66 74 79 70 69 73 6F 6D | MP4 | 9 |
| 4F 67 67 53 00 02 00 00 00 00 00 00 00 00 | OGG | 870 |
| 50 4B 03 04 14 00 01 00 63 00 00 00 00 00 | ZIP | 0 |
| 30 26 B2 75 8E 66 CF 11 A6 D9 00 AA 00 62 CE 6C | WMA, WMV | 41 |
| 1A 45 DF A3 93 42 82 88 6D 61 74 72 6F 73 6B 61 | MKV | 1 |
| 00 00 00 1C 66 74 79 70 4D 53 4E 56 01 29 00 46 4D 53 4E 56 6D 70 34 32 | MP4 | 0 |

# Список литературы


1. *Григорьев С.В.* Brahma.FSharp // https://sites.google.com/site/semathsrprojects/home/brahma-fsharp
2. *Richard M. Karp, Michael O. Rabin* Efficient randomized pattern-matching algorithms // IBM Journal of Research and Development 31 (2), March 1987, pp. 249-260.
3. *Alfred V. Aho, Margaret J. Corasick* Efficient string matching: An aid to bibliographic search // Communications of the ACM 18 (6), June 1975, pp. 333-340.
4. *Thomas H. Cormen* Introduction to Algorithms (3rd ed.) // Massachusetts Institute of Technology, 2009, pp. 253-280.
5. *Charalampos S. Kouzinopoulos, Konstantinos G. Margaritis* String Matching on a multicore GPU using CUDA // Parallel and Distributed Processing Laboratory, Department of Applied Informatics, University of Macedonia.
6. *Xinyan Zha, Sartaj Sahni* GPU-to-GPU and Host-to-Host Multipattern String Matching On GPU // Computer and Information Science and Engineering, University of Florida, April 5, 2011, pp. 25-27.
7. *Don Syme, Adam Granicz, Antonio Cisternino* Expert F# 2.0 (Expert's Voice in F#) // Apress, June 7, 2010, Chapter 9.
8. *Don Syme, Gregory Neverov, James Margetson* Extensible pattern matching via a lightweight language extension // Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, ACM, 2007, pp. 29-40.

9. *J. K. Buckle* Software Configuration Management // Macmillan Publishers Limited, 1982.Processing Laboratory, Department of Applied Informatics, University of Macedonia.
10. The OpenCL Specification // http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
11. NVIDIA CUDA Compiler Driver NVCC // http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc
12. Platform Invoke Tutorial // http://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx
13. *Gary C. Kessler.* File Signatures Table // http://www.garykessler.net/library/file_sigs.html, 15 May 2013.