**СОДЕРЖАНИЕ**

[Введение 6](#_Toc263651294)

[1 Анализ методов обработки текста 7](#_Toc263651295)

[1.1 Регулярные выражения 7](#_Toc263651296)

[1.2 Конечные автоматы 8](#_Toc263651297)

[1.3 Поиск по регулярным выражениям 11](#_Toc263651298)

[1.4 Генерация детерминированного конечного автомата на основании недетерминированного 11](#_Toc263651299)

[1.5 Расширенный конечный автомат (XFA) 12](#_Toc263651300)

[1.6 Постановка задачи 12](#_Toc263651301)

[2 Анализ особенностей программирование многоядерных систем 13](#_Toc263651302)

[2.1 Анализ отличий между CPU и GPU в параллельных расчётах 14](#_Toc263651303)

[2.2 GPGPU 18](#_Toc263651304)

[2.3 NVIDIA CUDA 19](#_Toc263651305)

[2.4 OpenCL 22](#_Toc263651306)

[3 Математические модели, положенные в основу проекта 24](#_Toc263651307)

[3.1 Теоретическая оценка роста производительности вычислительной системы при использовании нескольких вычислительных устройств 24](#_Toc263651308)

[4 Разработка программного средства 31](#_Toc263651309)

[4.1 Разработка архитектуры программного средства 31](#_Toc263651310)

[4.2 Построение конечного автомата для быстрого поиска слов 34](#_Toc263651311)

[4.3 Извлечение слов из текста 37](#_Toc263651312)

[4.4 Поиск ключевых слов в тексте, используя конечный автомат 42](#_Toc263651313)

[4.5 Выбор категории на основании найденных ключевых слов 42](#_Toc263651314)

[5 Реализация и тестирование программного средства 44](#_Toc263651315)

[5.1 Основные принципы реализации программного средства 44](#_Toc263651316)

[5.2 Обоснование технических приемов программирования 44](#_Toc263651317)

[5.3 Общая структура реализованного программного средства 45](#_Toc263651318)

[5.4 Тестирование, экспериментальные исследования и анализ полученных результатов 46](#_Toc263651319)

[6 Энергосбережение. Сокращение энергозатрат при внедрении проектируемой автоматизированной системы обработки информации 50](#_Toc263651320)

[7 Технико-экономическое обоснование разработки и использования программного средства 56](#_Toc263651321)

[7.1 Краткая характеристика программного средства 56](#_Toc263651322)

[7.2 Смета затрат и цена программного обеспечения 56](#_Toc263651323)

[7.3 Оценка экономической эффективности применения ПО у пользователя 68](#_Toc263651324)

[7.4 Выводы 73](#_Toc263651325)

[Заключение 74](#_Toc263651326)

[Список использованных источников 75](#_Toc263651327)

[Приложение Текст программы 78](#_Toc263651328)

ВВЕДЕНИЕ

В современном мире роль информации постоянно увеличивается. В некоторых отраслях производства данные играют уже роль, как первичного материала, так и конечного продукта. Распространение глобальной сети способствует стремительному увеличению не только потребителей, но и поставщиков новой информации. По данным на 2007 год, более чем 20% жителей мира являются пользователями глобальной сети интернет (в развитых странах достигает 70-80%) и это число сохраняет тенденцию ежегодного увеличения на более чем 160млн. [1]. В связи с этим вопрос автоматизации обработки информации становится все более важным. Не смотря на то, что распространение широкополосного доступа к глобальной сети интернет увеличивает популярность аудио и видео данных, текст, как один из способов передачи информация, по прежнему остается одним из самых выгодных по соотношению информативность/объем/стоимость. Автоматизированный анализ текста представляет в общем случае выделение в тексте определенных частей подходящих по правилу (например: поиск определенных слов или последовательностей слов) и деление текстов на различные классы. При этом очень важной проблемой остается производительность алгоритмов это реализующих. Не смотря на то, что закон Мура предсказывает удвоение количество транзисторов в процессорах каждые два года, для получение существенного прироста производительности, уже не достаточно просто использовать более новые процессоры. Более того рост тактовых частот у них уже почти остановился, уступая место тенденции использования многоядерных процессоров. При этом стоит заметить, что в современных компьютерах для вычислений может использоваться не только CPU, но и другие вычислительные блоки, например GPU, чьи потенциальные вычислительные возможности многократно превосходят CPU. Все это способствует развитию принципиально новым методам повышения производительности. Например, можно получить большие преимущества от использования концепции параллельного программирования, или массивно параллельного в случае GPU (и его SIMD архитектуры). Перспективность данного направления понимают и сами производители, о чем свидетельствует продвижение ими таких технологий как NVIDIA CUDA, AMD Stream, Intel Parallel, IBM Cell, а так же общего открытого стандарта вычислений в гетерогенной компьютерной системе OpenCL.

# АНАЛИЗ МЕТОДОВ ОБРАБОТКИ ТЕКСТА

Обзор различной литературы позволил сделать вывод, что тема обработки последовательных данных и текста (как частного случая последовательности символов) была востребована всегда. Схожие алгоритмы применялись и применяются для разнообразных целей, как научных так и прикладных.

В общем случае базовые операции при анализе текста делится на два вида: поиск на точное соответствие шаблону и нечеткий поиск подстроки

Каждый из этих видов имеет достаточно широкое применение и множество вариантов реализаций. Например, широкое распространение получило применение данных методик:

- в биологии при изучении ДНК, поиске совпадающих участков у различных видов [2] [3];

- в биоинформатике, области, появившейся благодаря симбиозу молекулярной биологии и компьютерных наук и в ее проекте генома человека(Human Genome Project) [4];

- для поиска вирусных сигнатур в файлах [5];

- обнаружение сетевых вторжений (Network Intrusion Detection Systems (NIDS)), фильтрация спама [6] [7] [8];

-поиск цитат и источников цитирования [9] [3];

-анализирование html, xml, разбиение текста на части [10] [11].

В основе многих алгоритмов и языков(Perl, Python, PCRE) с поиском на точное соответствие шаблону, используются различные вариации детерминированных (DFA) и недетерминированных (NFA) конечных автоматов, сгенерированных на основании регулярных выражений. [11]

## Регулярные выражения

Регулярное выражение – это формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (+\*?()|). Два регулярных выражения могут быть альтернативными или связанными, образую новое регулярное выражение. Например, если выражению e1 соответствует текст t1 и e2 соответствует t2, то e1|e2 соответствует t1 или t2, и выражению e1e2 соответствует t1t2.

Спецсимволы \*,+ и ? операторы повторения: e1\* соответствует последовательности с нуля или более (возможно разных) строк, каждая из которых соответствует выражению e1; e1 соответствует один или более; e1? – ноль или один.

Все эти операторы выполняются от самых слабых к самым сильным, сначала альтернативные, затем объединение и в конце операторы повторения. Но возможно использование скобок для группирования операторов и изменения последовательности их применения. Например ab\* и a(b\*) эквивалентны, и подходит строка “abbbb” , выражение (ab)\* уже имеет другой смысл, и ему соответствуют строки вида “ababab”.

В общем случае регулярные выражения могут содержать различные расширения, как и просто для более удобной и короткой записи (классы символов ([0-9],/w,..), количества повторений(a{3.5}) и т.д.), так добавляющие принципиально новую функциональность: обратные ссылки (backreferences). Стоит заметить, что использование обратных ссылок очень нежелательно, так в своей основе это NP-полная задача и наилучшая известная реализация требует экспоненциального времени поиска [11].

## Конечные автоматы

Конечные автоматы являются одним из способов представление последовательности символов.

Например, регулярное выражение: a(bb)+a , можно представить как:

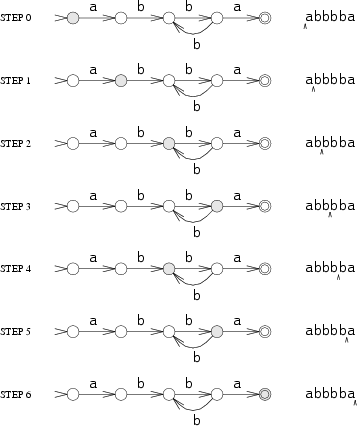


* + - 1. - Пример конечного автомата

В каждый момент времени конечный автомат находится в одном из своих состояний представленных на диаграмме окружностями. Автомат считывает посимвольно входную строку, и, следуя соответствующим дугам, меняет свое состояние (Рисунок 1.2).

Приведенный пример является детерминированным конечным автоматом(DFA), потому что из каждого состояний, при любом входном символе, существует единственное следующее состояние.

На Рисунок 1.3 изображен эквивалентный, но уже недетерминированный конечный автомат (из состояния s2, при входном символе b, имеется несколько выборов следующего состояния)



* + - 1. - Обработка строки “abbbba”



* + - 1. - Пример недетерминированного конечного автомата

### Генерация недетерминированного конечного автомата на основании регулярного выражения

Кен Томпсон был одним из первых, кто начал развивать и сформулировал хорошую теоретическую базу для использования регулярных выражений, конечных автоматов и алгоритмов поиска по регулярным выражениям [12]. Он так же впервые описал в 1968 году метод для конвертации регулярного выражения в недетерминированный конечный автомат.

Итоговый конечный автомат строился из частичных конечных автоматов каждого подвыражения. Частичные конечные автоматы имели одну или несколько ни с чем не соединенных дуг. Процесс создания итогового конечного автомата заканчивался соединением этих дуг с финальным состоянием, которое соответствовала успешному завершению поиска.

Single-character NFA

* + - 1. - NFA для сравнения на соответствие одного символа

Concatenation NFA

* + - 1. - NFA для объединения e1e2, выход e1 соединялся с входом e2

Alternation NFA

* + - 1. - NFA для альтернативы e1|e2, добавляется новое состояние с выбором любого автомата e1 или e2

Zero or 
one NFA

* + - 1. - NFA для e?, альтернатива между e автоматом и пустой дугой

Zero or 
more NFA

* + - 1. - NFA для e\*,

One or 
more NFA

* + - 1. - NFA для e+

Если посчитать количество новых состояний, то мы увидим что с использования этого метода на каждый символ или метасимвол из регулярного выражения создается одно новое состояние. Это значит, количество состояний у полученного конечно автомата будет не более длины регулярного выражения.

## Поиск по регулярным выражениям

Чтобы проверить соответствие регулярного выражения тексту, достаточно сгенерировать недетерминированный конечный автомат, и запустить его, использую текст как выходные данные. Однако за счет того что конечный автомат является недетерминированным, возможны ситуации когда из одного состояния возможны два и более перехода.

Один из подходов решения этой проблемы, использование обратных связей (backtracks): это простое рекурсивное решение, когда при не возможности перехода к следующему состоянию, конечный автомат возвращается к последней альтернативе, и проверяет другие альтернативы. При таком подходе возможно многократные чтения входной строки, и, если текст не будет соответствовать регулярному выражению, то конечный автомат должен будет проверить все возможные пути исполнения, а это значит, с увеличением альтернатив, время проверки будет расти экспоненциально.

Кен Томпсон в своей работе “Regular Expression Search Algorithm” [12] предлагает другой подход. Основная идея состоит в том, что конечный автомат находится не в одном состоянии, а сразу в нескольких, и соответственно переходит так же не в одно, а в несколько состояний. Это позволяет избежать обратных связей и повторного чтения входных данных.

## Генерация детерминированного конечного автомата на основании недетерминированного

Детерминированный конечный автомат (DFA) является более предпочтительным, потому что имеет только одно возможное состояние в каждый момент времени. Любой недетерминированный конечный автомат может быть преобразован в детерминированный, в котором каждое одно его состояние будет соответствовать списку состояний недетерминированного. В этом смысле выполнение NFA Томпсона [12] похоже на DFA: каждый список состояний соответствует какому-то DFA состоянию, а функция вычисляющая список следующих состояний, принимает список текущих и входной символ.

## Расширенный конечный автомат (XFA)

Во многих случаях в DFA модели может происходить резкий рост используемой памяти, это происходит, потому что в DFA нет разницы между явными состояниями и вычисляемыми. XFA модель [7] [8] предлагает использовать вместо вычисляемых состояний дополнительные переменные, такие как флаги и счетчики, которые будут использоваться вместе с текущим состоянием для идентификации текущего прогресса. Это позволят значительно уменьшить количество возможных состояний конечного автомата, но при этом появляется необходимость добавлять небольшие программы к некоторым состояниям (изменения флага, проверка флага, очистка, увеличение счетчика, проверка счетчика, очистка счетчика).

## Постановка задачи

Разработка программного средства параллельной обработки текста с использованием преимуществ многоядерных гетерогенных компьютерных с обеспечением возможности широкой масштабируемости.

Под обработкой текста в данном случае понимается:

- выделение отдельных слов в тексте;

- поиск слов из списка, используя детерминированные конечные автоматы;

- определение категорий, к которым принадлежит текст на основании шаблонов категорий.

# АНАЛИЗ ОСОБЕННОСТЕЙ ПРОГРАММИРОВАНИЕ МНОГОЯДЕРНЫХ СИСТЕМ

Вместе с совершенствованием компьютерных систем перед программистами встают различные задачи и проблемы:

1960-1980 года – основной язык ассемблер, перед программистами стоит задача написание кроссплатформенных и все более сложный программ, решение это проблемы это использование высокоуровневых языков C, Fortran;

1980-2000 года – необходимость реализовывать и поддерживать все более сложные программные системы, содержащие миллионы строк кода и разработанные десятками программистов, проблема обеспечения оптимальности кода для высокой производительности за счет закона Мура на время отходит на второй план, уступая место структурности, гибкости и легкости сопровождения как главным ориентирам. Решение: использование объектно-ориентированных языков С++, Java, C#; широкое использование библиотек компонентов; совершенствование методологий, шаблонов проектирования; увеличенная роль написания спецификации, тестирования, обзоров кода, и формальных инспекций.

До недавнего времени аппаратная и программная часть компьютерных систем были достаточно сильно разделены. Высокоуровневые языки позволили абстрагироваться от конкретной реализации, и программисту не нужно ничего знать о процессоре, закон Мура позволил ему не беспокоится о производительности, одни и те же программы, написанные на C еще в 70-х, можно запускать на современных процессорах и получать существенный прирост производительности.

В настоящий момент появляется новая проблема: производительность последовательных программ начинает отставать от закона Мура, но при этом программистам нужно продолжать увеличивать производительность, чтобы поддерживать внедрение новых функции и обработку всё увеличивающихся объемов данных (по оценкам IDC за 2009 год, общий объем информации в глобальной сети интернет составил около 800 миллиардов гигабайт, и к 2020 году эта цифра увеличится в 44 раза [13]), и при этом нужно избежать чрезмерного увеличения сложности.

Как одно из возможных решений этой проблемы это использование многоядерных устройств, и парадигмы массивно параллельных вычислений. Такая концепция позволяет достаточно легко масштабировать вычисления, распределять между различными устройствами, и в некоторых случаях даже между вычислительными блоками различного типа.

Методы программирования параллельных вычислений можно разделить на явные и неявные. К неявным относится автоматическая параллельность выполнения, обеспечивающаяся на аппаратном уровне (Superscalar Processors) и на уровне компилятора. К явным относятся же различные специализированные языки программирования (StreamIt, Star-P, BluSpec, Cilk) и специальные библиотеки и шаблоны проектирования. [14]

## Анализ отличий между CPU и GPU в параллельных расчётах

Рост тактовых частот универсальных процессоров упёрся в физические ограничения и высокое энергопотребление, и увеличение их производительности всё чаще происходит за счёт размещения нескольких ядер на одном чипе. Сейчас процессоры содержат лишь до 8 ядер (дальнейший рост не будет быстрым) и они предназначены для обычных приложений, используют архитектуру MIMD — множественный поток команд и данных. Каждое ядро работает отдельно от остальных, исполняя разные инструкции для разных данных.

Однако возрастающие требования графический приложений привело к тому что в универсальных процессоров появились некоторые особенности SIMD (одиночный поток команд, множество потоков данных) архитектуры, например специализированные векторные возможности (SSE2 и SSE3) для четырехкомпонентных (одинарная точность вычислений с плавающей точкой) и двухкомпонентных (двойная точность). Но для определённых задач применение GPU значительно выгоднее, так как они изначально сделаны для них.

В видеочипах NVIDIA основной блок — это мультипроцессор с восемью-тридцатью ядрами и сотнями ALU в целом [15], несколькими тысячами регистров и небольшим количеством разделяемой общей памяти. Кроме того, видеокарта содержит быструю глобальную память с разделяемым доступом к ней из всех мультипроцессоров, локальную память в каждом мультипроцессоре, а также специальную память для констант.

Самое существенная особенность, что все ядра мультипроцессора в GPU являются SIMD ядрами. И эти ядра исполняют одни и те же инструкции одновременно, такой подход позволяет увеличить количество исполнительных блоков за счёт их упрощения, но при этом требует достаточно специфического программирования, и если такой массивно параллельный стиль программирования является нормальным для графических алгоритмов и многих научных задач, то для применения к обычным прикладным программам, зачастую необходимо произвести значительную адаптацию изначальных алгоритмов.

Основные различия между архитектурами CPU и GPU. Ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью, а GPU проектируются для быстрого исполнения большого числа параллельно выполняемых потоков инструкций. Универсальные процессоры оптимизированы для достижения высокой производительности единственного потока команд, обрабатывающего и целые числа и числа с плавающей точкой. При этом доступ к памяти случайный.

Разработчики CPU стараются добиться выполнения как можно большего числа инструкций параллельно, для увеличения производительности. Для этого, начиная с процессоров Intel Pentium, появилось суперскалярное выполнение, обеспечивающее выполнение двух инструкций за такт, а Pentium Pro отличился внеочередным выполнением инструкций. Но у параллельного выполнения последовательного потока инструкций есть определённые базовые ограничения и просто увеличением количества исполнительных блоков кратного увеличения скорости не добиться.

У видеочипов работа простая и распараллеленная изначально. Видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели. Обработка полигонов и пикселей независима, их можно обрабатывать параллельно, отдельно друг от друга. Поэтому, из-за изначально параллельной организации работы в GPU используется большое количество исполнительных блоков, которые легко загрузить, в отличие от последовательного потока инструкций для CPU. Кроме того, современные GPU также могут исполнять больше одной инструкции за такт (dual issue). Так, архитектура Tesla в некоторых условиях запускает на исполнение операции MAD+MUL или MAD+SFU одновременно.

GPU отличается от CPU ещё и по принципам доступа к памяти. В GPU он связанный и легко предсказуемый — если из памяти читается тексель текстуры, то через некоторое время будут запрошены и соседние тексели. При записи то же — пиксель записывается в буфер кадра, и через несколько тактов будет записываться расположенный рядом с ним. Поэтому организация памяти отличается от той, что используется в CPU. И видеочипу, в отличие от универсальных процессоров, нет необходимости в кэш-памяти большого размера, а для текстур требуются лишь несколько (до 128-256 в нынешних GPU) килобайт.

Работа с памятью у GPU и CPU также имеет отличия. Не все центральные процессоры имеют встроенные контроллеры памяти, а у всех GPU обычно есть по несколько контроллеров, вплоть до восьми 64-битных каналов в чипе NVIDIA GT200 [15]. Кроме того, на видеокартах применяется более быстрая память, и в результате видеочипам доступна в разы большая пропускная способность памяти, что также весьма важно для параллельных расчётов, оперирующих с огромными потоками данных.

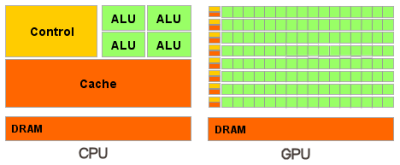
В универсальных процессорах на буферы команд идут большие количества транзисторов и площадь чипа, аппаратное предсказание ветвления и огромные объёмы начиповой кэш-памяти. Все эти аппаратные блоки нужны для ускорения исполнения немногочисленных потоков команд. Видеочипы тратят транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов. Вышеперечисленное не ускоряет выполнение отдельных потоков, оно позволяет чипу обрабатывать нескольких тысяч потоков, одновременно исполняющихся чипом и требующих высокой пропускной способности памяти.

Есть отличия и в кэшировании: универсальные центральные процессоры используют кэш-память для увеличения производительности за счёт снижения задержек доступа к памяти, а GPU используют кэш или общую память для увеличения полосы пропускания. CPU уменьшает задержки доступа к памяти при помощи кэш-памяти большого размера, а также предсказания ветвлений кода. Эти аппаратные части занимают большую часть площади чипа и потребляют много энергии. Видеочипы скрывает задержки доступа к памяти при помощи одновременного исполнения нескольких тысяч потоков — в то время, когда одни потоки ожидают данные запрошенные из памяти, вычислительные ядра могут выполнять вычисления других потоков без ожидания и задержек.

Есть множество различий и в поддержке многопоточности. CPU исполняет 1-2 потока вычислений на одно процессорное ядро, а видеочипы могут поддерживать до 1024 потоков на каждый мультипроцессор, которых в чипе несколько. И если переключение с одного потока на другой для CPU обходится достаточно дорого (сотни тактов), то GPU переключает несколько потоков за один такт, чем и обеспечивает выполнения такого большого количества потоков без больших накладных расходов по переключению контекста.

Центральные процессоры используют SIMD (одна инструкция выполняется над многочисленными данными) блоки для векторных вычислений, а видеочипы применяют SIMT (одна инструкция и несколько потоков) для скалярной обработки потоков. SIMT не требует, чтобы разработчик преобразовывал данные в векторы, и допускает произвольные ветвления в потоках.

Можно сказать, что в отличие от современных универсальных CPU, видеочипы предназначены для параллельных вычислений с большим количеством арифметических операций. И значительно большее число транзисторов GPU работает по прямому назначению — обработке массивов данных, а не управляет исполнением (flow control) немногочисленных последовательных вычислительных потоков.

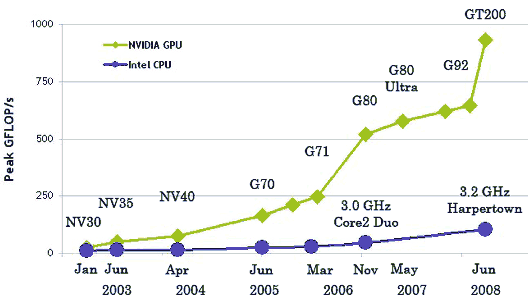


* + - 1. - Сравнительная схема, сколько места в CPU и GPU занимает разнообразная логика [15]

В итоге, основой для эффективного использования потенциала GPU в научных и иных неграфических расчётах является распараллеливание алгоритмов на множество независимых потоков, которые будут исполняться на сотнях (а в будущем возможно и еще на большем количестве) исполнительных блоков, имеющихся в видеочипах. Использование нескольких GPU даёт ещё больше вычислительных мощностей для решения различных задач.

Выполнение расчётов на GPU показывает отличные результаты в алгоритмах, использующих параллельную обработку данных. При этом лучшие результаты достигаются, если отношение числа арифметических инструкций к числу обращений к памяти достаточно велико. Это предъявляет меньшие требования к управлению исполнением (flow control), а высокая плотность математики и большой объём данных отменяет необходимость в больших кэшах, как на CPU.

В результате всех описанных выше отличий, теоретическая производительность видеочипов значительно превосходит производительность CPU. Компания NVIDIA приводит такой график роста производительности CPU и GPU за последние несколько лет (Рисунок 2.2)



* + - 1. - Сравнительный рост производительности CPU и GPU [15]

Естественно, нельзя сравнивать напрямую производительность вычислений. На CPU гораздо проще на практике достичь теоретических цифр, на GPU же чтобы получить максимальную производительность необходимо будет не только нагрузить все вычислительные ядра расчетами, но и так же эффективно организовать работу с памятью, что на практике требует достаточно глубокого знание архитектуры GPU.

## GPGPU

General Purpose Graphics Processing Units — технология использования графического процессора для вычислений общего назначения.

В настоящее время существует 4 реализации даного подхода — NVIDIA CUDA (NVIDIA GeForce 8 и старше), AMD Stream (AMD Radeon HD4000 и старше), Compute Shader (Microsoft DirectX 11), OpenCL (открытый стандарт).

## NVIDIA CUDA

Технология CUDA — это программно-аппаратная вычислительная архитектура NVIDIA, основанная на расширении языка C, которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. CUDA помогает реализовывать алгоритмы, выполнимые на графических процессорах видеоускорителей серии GeForce 8 и старше, а также Quadro и Tesla.

Программирования GPU при помощи CUDA все еще не является тривиальной задачей, и требует глубоких знаний аппаратной части, но все же значительно проще, чем с ранними GPGPU решениями (раньше приходилось писать программы в базисе компьютерной графике: использую текстуры, шейдеры и остальные специфичные для графики технологии, что в большинстве прикладных задач добавляло дополнительную сложность). Такие программы требуют разбиения приложения между несколькими мультипроцессорами, но без разделения данных, которые хранятся в общей видеопамяти. И так как CUDA программирование для каждого мультипроцессора требует хорошего понимания организации памяти.

В основе API лежит расширенный язык C, а для трансляции кода с этого языка в состав CUDA SDK входит компилятор командной строки nvcc, созданный на основе открытого компилятора Open64.

Основные характеристики CUDA:

унифицированное программно-аппаратное решение для параллельных вычислений на видеочипах NVIDIA;

большой набор поддерживаемых решений, от мобильных до мультипроцессорных;

стандартный язык программирования Си;

стандартные библиотеки численного анализа FFT (быстрое преобразование Фурье), BLAS (линейная алгебра), Thrust(библиотека С++ шаблонов);

оптимизированный обмен данными между CPU и GPU;

взаимодействие с графическими API OpenGL и DirectX;

поддержка 32- и 64-битных операционных систем: Windows XP, Windows Vista, Linux и MacOS X;

возможность разработки на низком уровне.

Так же официально поддерживаются все основные дистрибутивы Linux (Red Hat Enterprise Linux 3.x/4.x/5.x, SUSE Linux 10.x), и не официально, но без ощутимых проблем работает и на других: Fedora Core, Ubuntu, Gentoo и др.

Среда разработки CUDA (CUDA Toolkit) включает:

- компилятор nvcc;

- библиотеки FFT и BLAS;

- профилировщик;

- отладчик gdb для GPU;

- CUDA runtime драйвер в комплекте стандартных драйверов NVIDIA

- руководство по программированию;

- CUDA Developer SDK (исходный код, утилиты и документация).

Так же существует проект NVIDIA Parallel Nsight ("Nexus")- дополнение к Visual Studio значительно упрощающее отладку GPU приложений [16]

### Особенности CUDA

Графический конвейер можно представить набором стадий обработки. Блок геометрии генерирует треугольники, блок растеризации — пиксели, отображаемые в кадровый буфер. Раньше модель программирования GPGPU выглядела следующим образом:

Для перенесения вычисления на GPU, нужен был особый подход. Даже поэлементное сложение двух векторов потребует отрисовки фигуры во внеэкранный буфер. Фигура растеризуется, цвет каждого пикселя вычисляется по заданной небольшой программе - пиксельному шейдеру. Программа считывает входные данные из текстур для каждого пикселя, складывает их и записывает в выходной буфер. Как видно изначально графическая направленность накладывает множество дополнительных расходов ресурсов. Поэтому, применение GPGPU для вычислений общего назначения имеет ограничение в виде слишком большой сложности обучения разработчиков.(язык пиксельных и вершинных шейдеров только похож синтаксисом на C, но на само деле имеет множество особенностей) Ранние методы GPGPU являются в основном достаточно нетривиальными, позволяющим использовать мощность GPU, но без всякого удобства. Данные там представлены изображениями (текстурами), а алгоритм — процессом растеризации. Плюс весьма специфичную модель памяти и исполнения.

Программно-аппаратная архитектура CUDA для вычислений на GPU компании NVIDIA отличается от предыдущих моделей GPGPU тем, что позволяет писать программы для GPU на языке C со стандартным синтаксисом и необходимостью в минимуме расширений для доступа к вычислительным ресурсам видеочипов. CUDA не зависит от графических API, и обладает некоторыми особенностями, предназначенными специально для вычислений общего назначения.

Преимущества CUDA перед старым подходом (GPGPU вычисления):

-интерфейс программирования приложений CUDA основан на стандартном языке программирования C с расширениями, что упрощает процесс изучения и внедрения архитектуры CUDA;

-CUDA обеспечивает доступ к разделяемой между потоками памяти размером в 16 Кб на мультипроцессор, которая может быть использована для организации кэша с широкой полосой пропускания, по сравнению с текстурными выборками;

-более эффективная передача данных между системной и видеопамятью;

-отсутствие необходимости в графических API с избыточностью и накладными расходами;

-линейная адресация памяти, возможность записи по произвольным адресам, scatter или gather операции;

-аппаратная поддержка целочисленных и битовых операций.

Основные ограничения CUDA:

-отсутствие поддержки рекурсии для выполняемых функций;

-минимальная ширина блока в 32 потока;

-закрытая архитектура CUDA, принадлежащая NVIDIA.

Слабыми местами программирования при помощи предыдущих методов GPGPU является то, что эти методы не используют блоки исполнения вершинных шейдеров в предыдущих неунифицированных архитектурах, данные хранятся в текстурах, и выводятся во внеэкранный буфер, а многопроходные алгоритмы используют пиксельные шейдерные блоки. В ограничения GPGPU можно включить: недостаточно эффективное использование аппаратных возможностей, ограничения полосой пропускания памяти, обязательное использование графического API.

Основные преимущества CUDA по сравнению с предыдущими методами GPGPU вытекают из того, что эта архитектура спроектирована для эффективного использования неграфических вычислений на GPU и использует язык программирования C, не требуя переписывания алгоритмов в удобный для концепции графического конвейера вид. CUDA предлагает новый путь вычислений на GPU, не использующий графические API, предлагающий произвольный доступ к памяти (scatter/gather – режим косвенной адресации). Такая архитектура лишена недостатков GPGPU и использует все исполнительные блоки, а также расширяет возможности за счёт целочисленной математики и операций битового сдвига.

Кроме того, CUDA открывает некоторые аппаратные возможности, недоступные из графических API, такие как разделяемая память. Это память небольшого объёма (16 килобайт на мультипроцессор), к которой имеют доступ блоки потоков. Она позволяет кэшировать наиболее часто используемые данные и может обеспечить более высокую скорость, по сравнению с использованием текстурных выборок для этой задачи. Что, в свою очередь, снижает чувствительность к пропускной способности параллельных алгоритмов во многих приложениях.

Также, графические API в обязательном порядке хранят данные в текстурах, что требует предварительной упаковки больших массивов в текстуры, что усложняет алгоритм и заставляет использовать специальную адресацию. CUDA же позволяет читать данные по любому адресу. Для разработчиков, желающих получить доступ к низкому уровню (например, при написании другого языка программирования), в CUDA есть возможность низкоуровневого программирования на ассемблере.

Уже существует множество действующих решений использующих это архитуктуру. [17] Значительная часть этого списка составляют программы для экономических и научных расчетов. Среди них есть клиенты проектов распределенных вычислений Folding@home и SETI@home. Существуют ПС для кодирования и декодирования аудио и видео данных. Прирост производительности в вышеуказанных приложениях при включении CUDA составляет от 1,5 до 100 раз. [17]

## OpenCL

OpenCL – открытый стандарт программирования гетерогенных компьютерных систем [18]. При чем нужно обратить внимание, это не стандарт для разработки приложений только для GPU, OpenCL изначально задумывался как единый стандарт для написания приложений, которые должны исполняться в системе, где установлены различные по архитектуре процессоры, ускорители и платы расширения(x86, x86-64, Itanium, SpursEngine (Cell), NVidia GPU, AMD GPU, VIA (S3 Graphics) GPU). В разработке и финансировании участвовали такие крупные компании как, Apple, AMD, IBM, Activision Blizzard, Intel, NVidia, ARM и др. [18]. Необходимо отметить, что основные идеи CUDA и OpenCL достаточно схожи. OpenCL – по своей сути расширения языка С, со сходным синтаксисом, использующие одинаковую программную модель в качестве основной: Data Parallel (SIMD), так же OpenCL поддерживает Task Parallel programming model – модель, когда одновременно могут выполняться различные kernel. О схожести двух технологий говорит даже то что NVidia выпустила специальный документ для облегчения написания кода на CUDA, который можно с минимальными затратами перенести на OpenCL [19] .Основной проблемой реализации OpenCL от NVidia является не высокая производительность по сравнению с CUDA, но нужно обратить внимание что с каждой новой версией драйверов производительность OpenCL под управлением CUDA повышается. После не особо удачного конкурирования платформы AMD Stream с CUDA, AMD в 2009 году внедрила в Stream поддержку OpenCL. Концептуально программирование с использованием OpenCL не сильно отличается от использования CUDA, основная задача состоит в правильном разбиении всей программы на блоки, а блоки на потоки. И точно так как и в CUDA большее внимание нужно уделять работе с памятью. Такая схожесть во многом объясняется тем, что обе эти архитектуры разрабатывались с учетом архитектуры и особенностей современных GPU.

# МАТЕМАТИЧЕСКИЕ МОДЕЛИ, ПОЛОЖЕННЫЕ В ОСНОВУ ПРОЕКТА

## Теоретическая оценка роста производительности вычислительной системы при использовании нескольких вычислительных устройств

### Теорема Грэкхама и Брента

Если некоторое вычислительное устройство выполняет одну операцию за единицу времени, то M операций оно выполнит за M единиц. Cистеме из N устройств на ту же работу понадобится M/N единиц времени. Это идеальный случай, когда задача состоит из списка информационно независимых между собой операций. Для вычисления повышения производительности в задачах, с некоторой степенью зависимости по данным, которая порождает последовательное поведение, используем теоремы Грэкхама [20] и Брента [21].

Для описания роста производительности используем величины:

(3.1)

где P – степень параллельности задачи;

– время необходимое для выполнения задачи, когда вычислительная система имеет одно вычислительное устройство (последовательное выполнение);

– время необходимое для выполнения задачи, когда вычислительная система может иметь столько вычислительных устройств, сколько требуется для задачи (полностью параллельное выполнение);

Для расчета оценки роста производительность системы основанной на p вычислительных устройств:

, (3.2)

где – рост производительности системы основанной на p вычислительных устройств;

– время необходимое для выполнения задачи, когда вычислительная система имеет одно вычислительное устройство (последовательное выполнение);

– время необходимое для выполнения задачи, когда вычислительная система имеет p вычислительных устройств (частично параллельное выполнение);

При отсутствии особых приемов оптимизаций с кэшпамятью (которые иногда позволяют достичь супер линейного повышения производительности [22]), нижняя граница оценки времени определяется:

, (3.3)

где количество вычислительных устройств;

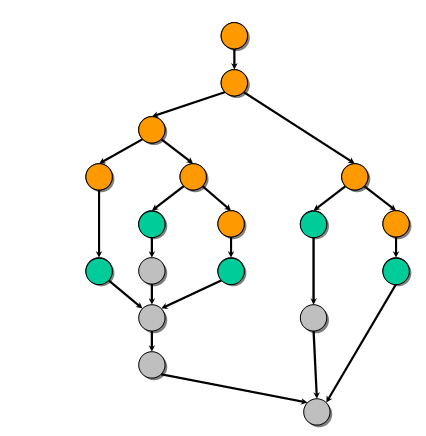
Это означает что, если имеется 2 вычислительных устройства, невозможно получить ускорение в 2 раза, так как всегда будет дополнительные расходы из-за возможных зависимостей по данным и дополнительных расходов связанных с параллельным выполнением. С прикладной точки зрения, верхняя граница оценки времени выполнения, является более информативной, так как позволяет оценить время в наихудшем сценарии, в то время как нижняя граница оценки дает информацию о возможном теоретическом максимуме, который труднодостижим в реальных задачах.

В теореме Грэкхама и Брента используется модель представления приложения в виде направленного графа (рисунок 3.1), где вершины представляют собой выполняемые операции или группы последовательных операций, а дуги - зависимости по данным (поток данных). Граф имеет одну начальную вершину и одну конечную. Операция может быть выполнена, только если все операции, от которых она зависит по данным, завершены. Операция выполняется на одном вычислительном устройстве. Такая модель с акцентом на поток данных, позволяет оценить потенциал программы для распараллеливания вычислений.

В теореме Грэкхама и Брента используется в основе модель системы выполнения с «жадным» планированием.

Планирование выполнения – это стратегия распределения операций по вычислительным устройствам.

«Жадное» планирование (Greedy Scheduling) – это планирование, в котором ни одно вычислительное устройство не бездействует при наличии операций готовых к выполнению.



* + - 1. - Модель приложения в виде графа.

Операции готовы к выполнению, если все предшествующие им операции выполнены.

Как следствие теорем Грэкхама и Брента, доказано что, если для приложения доступно бесконечное количество вычислительных устройств, то время выполнения зависит от длины критического пути на графе (путь максимальной длины в ориентированном ациклическом графе). В таком случае можно рассматривать, как время, затрачиваемое на критический путь на таком графе. Это частный случай, в котором количество вычислительных устройств бесконечно (*p=∞),* в общем случае, повышение производительности зависит от критического пути и от количества доступных для планировщика выполнения вычислительных устройств. Основывая на этих двух параметрах, они предложили верхнюю оценку времени необходимого приложению при использовании *p* вычислительных устройств:

(3.4)

где – время необходимое для выполнения задачи, когда вычислительная система имеет p вычислительных устройств;

– суммарное время выполнения всех операций (вычислительная система имеет одно вычислительное устройство (последовательное выполнение));

– суммарное время необходимое для выполнения всех операций критического пути;

Краткое доказательство:

Верхнюю границу оценки времени, которое система работой с полной утилизацией доступных вычислительных ресурсов (задействует все вычислительные устройства), можно оценить как:

, (3.5)

Времени, которое система работой с неполной утилизацией доступных вычислительных ресурсов (задействует не все вычислительные устройства), не может превышать длительность критического пути:

, (3.6)

Общее время:

, (3.7)

Подставляя (3.5) и (3.6) в выражение (3.7) получим:

, (3.8)

Что согласуется с (3.4).

### Закон Амдала

В общем случае закон Амдала используется для нахождения максимума ожидаемого роста производительности всей системы, при увеличении производительности только части [23]. В отношении параллельных вычислений закон Амдала это модель ожидаемого повышения производительности параллельной реализации алгоритма по сравнению с его последовательной версией, с допущением, что размер задачи остается прежним.

Тогда ускорение, которое может быть получено на вычислительной системе из *p* вычислительных устройств, по сравнению с системой с одним вычислительным устройством, не будет превышать величины:

где – ускорение на системе из процессоров;

– доля объема вычислений, которая может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных процессоров );

– количество процессоров в системе;

Закон Амдала показывает, что прирост эффективности вычислений зависит от алгоритма задачи и ограничен сверху для любой задачи с .

Данный закон позволяет сделать грубую оценку эффективности распараллеливания алгоритма и о возможном оптимальном необходимом количестве вычислительных устройств.

* + - * 1. – Значение максимального ускорение выполнения

|  |  |  |  |
| --- | --- | --- | --- |
| ,% |  | | |
| 8 | 16 | 32 |
| 0 | 1 | 1 | 1 |
| 10 | 1,09 | 1,10 | 1,11 |
| 25 | 1,02 | 1,31 | 1,32 |
| 50 | 1,78 | 1,88 | 1,94 |
| 75 | 1,49 | 3,37 | 3,66 |
| 100 | 8 | 16 | 32 |

Например, если доля параллельных вычислений в алгоритме 75% то верхний предел ускорение в 1.49 раз при использовании 8 процессоров, эффективность составляет 18%; в 3.37 раза при 16 процессорах - эффективность 21%; и в 3.66 раза при 32 процессорах – эффективность 11%.

### Особенности оценки увеличения производительности

В общем случае, исходя из выражения (3.4), повышения производительности системы можно добиться несколькими способами: уменьшением общего времени выполнения () или уменьшением времени выполнения критического пути (). Причем эти две величины взаимосвязаны, и уменьшение одной из них приводит к увеличению другой. Такой эффект приводит к тому, что оптимизированная версия алгоритма, может демонстрировать противоположные результаты на системах с разным количеством вычислительных устройств. Этот парадокс объясняется теоремой Грэкхамаи Брента.

Время выполнения, как верхняя оценка времени выполнения:

. (3.10)

Пример:

До оптимизации: ,

Рассчитаем по (3.10):

Рассчитаем по (3.10):

После оптимизации: .

Рассчитаем по (3.10):

Рассчитаем по (3.10):

Получаем , но .

В результате, оптимизация по уменьшению общего времени выполнения за счет увеличение продолжительности критического пути, приводит к повышению итоговой производительности при использовании 32 процессоров, но ухудшает результат при использовании 512 процессоров.

Вывод: эффективная оптимизация алгоритма возможно только при наличии информации о системе (или предполагаемой системе), на которой этот алгоритм будет выполняться.

# РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

## Разработка архитектуры программного средства

По техническому заданию данное программное средство должно представлять собой библиотеку, обеспечивающие две основные функции: управление набором правил, и эффективная проверка на соответствие им определённого набора текстовых файлов. Основным критерием качества итогового программного средства является обеспечиваемая им производительность (скорость обработки текстовых файлов). Для достижения необходимой производительности наиболее ресурсоёмкие алгоритмы необходимо проектировать с учётом особенностей массивно параллельных вычислений, но так как проектирование, реализация, отладка и поддержка такого кода очень трудоёмка, а так же для обеспечения лучшей структуры, часть модуля проектируется для выполнения в обычном последовательном режиме. Как следствие такого архитектурного решения, разрабатываемое ПС, будет требовать наличия в компьютерной системе различных вычислительных устройств: GPU и CPU, то есть компьютерная система должна быть гетерогенной. На основании проведённых математических исследований (П. 3.1) было приято решение, использовать архитектуру, при которой только те части программы разрабатываются для параллельных вычислений, которые являются наиболее требовательными по ресурсам.

* + - 1. – Архитектура приложения



* + - 1. - Схема программы

Одним из самых важных преимуществ такой архитектуры это широкие возможности автоматической масштабируемости основных алгоритмов программного средства.



* + - 1. - Распределения вычислений в зависимости от количества вычислительных ядер в устройстве

Таким образом, основной задачей становится разделение алгоритмов на набор не зависимо выполняющихся блоков.

Основные алгоритмы, используемые в разрабатываемом программном средстве:

- построение конечного автомата для быстрого поиска слов;

- выделение слов в тексте;

- поиск ключевых слов в тексте, использую конечный автомат;

- выбор категории на основании найденных ключевых слов.

## Построение конечного автомата для быстрого поиска слов

В основе данного алгоритма лежит работа Томпсона Кена [12]. Входным сигналом для конечного автомата является текущий символ, сигналом на выходе, служит идентификатор слова. Так последовательно по каждому из слов в словаре, обрабатывается конечным автоматом, при этом, если перехода к следующему состоянию не существует, то создаётся новое состояние и сохраняется. Таким образом, для поиска слов из словаря в тексте, используя полученный детерминированный конечный автомат, требуется только один проход, или время O(N).

Рассмотрим последовательность шагов, чтобы получить конечный автомат для словаря из 5 слов (Рисунок 4.4):

Словарь:

- BSUIR;

- BELARUS;

- SCIENCE;

- SCIENTIST;

- SOFTWARE.

Последовательность шагов:

- для слова «BSUIR» необходимо создать состояния 2, 3, 4, 5, 6 и соответствующие переходы между ними, состоянию 6 присвоить выходной сигнал идентификатор слова «BSUIR»;

- для слова «BELARUS» необходимо создать только состояния 7, 8, 9, 10, 11, 12, состояние 2 уже существу (с переходом из 1 в 2 при входном сигнале «B»), состоянию 12 присвоить выходной сигнал идентификатор слова «BELARUS»;

- для слова «SCIENCE» необходимо создать состояния 13, 14, 15, 16, 17, 18, 19 и соответствующие переходы между ними, состоянию 19 присвоить выходной сигнал идентификатор слова «SCIENCE»;

- для слова «SCIENTIST» необходимо создать состояния 20, 21, 22, 23 и соответствующие переходы между ними, состоянию 23 присвоить выходной сигнал идентификатор слова «SCIENTIST»;

- для слова «SOFTWARE» необходимо создать состояния 24, 25, 26, 27, 28, 29, 30 и соответствующие переходы между ними, состоянию 30 присвоить выходной сигнал идентификатор слова «SOFTWARE».

В итоге было создано 30 состояний, 5 из которых имеют выходные сигналы и 29 переходов между состояниями. Время построения конечного автомата 37 итераций.



* + - 1. - Пример конечного автомата полученного на основании словаря из 5 слов

Особенностью такого алгоритма является то, что чем большее количество слов присутствует в словаре, тем меньше новых состояний нужно создавать (например, при добавлении слова «SCIENTIST» необходимо создать только 4 новых состояния и 4 новых перехода), и в общем случае количество состояний и переходов не превышает сумму длин всех ключевых слов. Время построения конечного автомата пропорционально сумме длин всех слов: O(∑(Li)).



* + - 1. - Генерация таблицы переходов для конечного автомата. Схема алгоритма.

## Извлечение слов из текста

Сложность в распараллеливании алгоритма извлечению слов и предложений из текста состоит в том, что необходимо обеспечивать конкурентный доступ на запись. Использования варианта с атомарной операцией добавления в список, сведёт большую часть коду к необходимости организовывать сериализованный доступ к памяти, что в свою очередь приведёт к резкому уменьшению показателя параллельности кода. Чтобы решить эту проблему, было решено применить шаблон параллельного программирования «сканирование». Задача, которую решает данный шаблон, это нахождение для каждого элемента: . При последовательном вычислении необходимо N-1 операций сложения, и O(N) время. При параллельной реализации количество операций порядка O(N\*Log(N)), но время выполнения при наличии бесконечного числа вычислительных ядер порядка T∞ = O(Log(N)).



* + - 1. – Параллельная реализация шаблона «сканирование»

Несмотря на то, что в последовательном варианте выполнения такого алгоритма меньше вычислений, чем в простой реализация параллельного алгоритма, данный алгоритм обладает большим коэффициентом параллельности, что позволяет ему эффективно выполнятся на устройствах с большим количество вычислительных ядер. Для уменьшения количества выполняемых операция, можно использовать оптимизированную по количеству операций, версию реализации данного шаблона [15]. Основная идея которой сводится к вычислению сумм в две фазы, использую шаблон параллельного программирования «сбалансированное двоичное дерево» (balanced trees). Идея этого шаблона заключается в построении по входным данным сбалансированного двоичного дерева и движение от листьев к корню для вычисления частичной суммы и затем от корня к листьям для вычисления префиксной суммы. Бинарное дерево с N листьями будет иметь Log(N) уровней. Если мы выполняем одну операцию сложения в узле, то нам понадобится только O(N) операций для одного прохода по дереву.

В первой фазе (восходящей) идет проход от листьев к корню, считаются частичные суммы, что требует N-1 операций сложения и Log(N) времени, это фаза так же называется «редукция», потому что в итоге ее выполнения в корне дерева будет сумма всех листьев дерева (Рисунок 4.7.)



* + - 1. - Фаза 1. Эффективная параллельная реализация шаблона «сканирование».

Псевдокод для первой фазы:

for d := 0 to log2(n) - 1 do

for k from 0 to n – 1 by 2d + 1 in parallel do

x[k + 2d + 1- 1] := x[k + 2d - 1] + x [k + 2d + 1 - 1]

Во второй фазе (нисходящей), идет обратный проход от корня к листьям, и использую частичные суммы, полученные в результате первой фазы, вычисляются префиксные суммы для каждого элемента (Рисунок 4.8).



* + - 1. - Фаза 2. Эффективная параллельная реализация шаблона «сканирование»

Псевдокод для второй фазы:

x[n - 1] := 0

for d := log2(n) – 1 down to 0 do

for k from 0 to n – 1 by 2d + 1 in parallel do

t := x[k + 2d - 1]

x[k + 2d - 1] := x [k + 2d + 1 - 1]

x[k + 2d + 1 - 1] := t + x [k + 2d + 1 - 1]

Сам алгоритм выделения слов в тексте основывается на выделение терминальных символов, таких как пробельные символы (пробел, знак табуляции, переноса строки), знаки пунктуации (точка, запятая, точка с запятой и другие), и вычисления порядковых номеров терминальных символов, который и будут считаться началом слов. Операция выделение символов не зависима по данным и способна выполняется параллельно. Вычисление порядковых номеров происходит с помощью применения шаблона «сканирование», которое позволяет добиться значительной степени параллельности (Рисунок 4.9 и Рисунок 4.10).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Шаг 1.  Загрузка текста  Время: 1 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |  | B | S | U | I | R |  |  |  | s | c | i | e | n | c | e | |
|  |  |
| Шаг 2. Нормализация текста  Время: 1 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |  | B | S | U | I | R |  |  |  | S | C | I | E | N | C | E | |
|  |  |
| Шаг 3.  Выделение терминальных символов  Время: 1 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | | 1 |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |  | |
|  |  |
| Шаг 4.  Алгоритм параллельного сканирования  Время: Log(N) | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | |
|  |  |
| Шаг 5. Выделение порядковых номеров слов  Время: 1 | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
|  |  |
| Шаг 6. Сохранение позиций слов  Время: 1 | |  |  |  | | --- | --- | --- | | 0 | 1 | 2 | | 0 | 0 | 8 | |

* + - 1. - Последовательность шагов алгоритма параллельного извлечения слов из текста



* + - 1. - Извлечение слов из текста. Схема алгоритма.

## Поиск ключевых слов в тексте, используя конечный автомат

При наличии таблицы переходов конечного автомата, и начальных позиций всех слов, задача параллельного поиска слов сводится к параллельной обработке каждого слова этим конечным автоматом, и сохранением сигнала на выходе полученного в результате. Затем схожим алгоритмом, с выделением слов в тексте, удаляются слова, которые не являются ключевыми (отсутствие выходного сигнала), в итоге получаем список ключевых слов данного текста.

На основании примеров изображенных на Рисунок 4.4 и Рисунок 4.9 для текста «BSUIR science»(между словами согласно примеру 3 символа пробела), получаем для первого потока начало обработки в позиции 0 (соответственно символ «B») , для второго позиция 8 (символ «S»), начиная с этих позиций обработку можно вести параллельно.

* + - * 1. - Пример поиска ключевых слов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Поток №1 | | | Поток №2 | | |
| Вх. сигнал | Состояние | Вых. сигнал | Вх. сигнал | Состояние | Вых. сигнал |
| B | 1 | 0 | S | 1 | 0 |
| S | 2 | 0 | C | 13 | 0 |
| U | 3 | 0 | I | 14 | 0 |
| I | 4 | 0 | E | 15 | 0 |
| R | 5 | 0 | N | 16 | 0 |
|  | 6 | 1 | C | 17 | 0 |
|  | 0 | 1 | E | 18 | 0 |
|  | 0 | 1 |  | 19 | 4 |

## Выбор категории на основании найденных ключевых слов

Категория представляет собой неупорядоченный набор идентификаторов слов. Идентификатор слова представляет собой набор синонимичных ключевых слов. При этом в данном контексте это могут быть не обязательно синонимы, возможен случай, когда несколько различных слов, в контексте определенных категорий имеют синонимичный смысл, тогда они в словаре хранятся с одинаковыми идентификаторами. Таким образом, после фазы извлечения ключевых слов, когда получен только массив идентификаторов слов, такие слова синонимы являются эквивалентными и не различимыми (представляются одинаковыми идентификаторами см. Таблица 4.2).

* + - * 1. - Пример идентификаторов слов

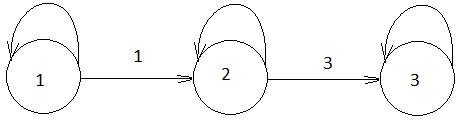
|  |  |  |
| --- | --- | --- |
| Ключевое слово | Выходной сигнал конечного автомата | Идентификатор |
| BELARUS | 1 | 1 |
| BSUIR | 2 | 2 |
| SCIENCE | 3 | 3 |
| SCIENTIST | 4 | 3 |
| SOFTWARE | 5 | 4 |

Текст считается принадлежащим категории, если в нем есть предложение полностью включающее все слова из данной категории. Например категория «Belarus scientist’s»: представляет собой шаблон:

* + - * 1. - Пример преобразования категории

|  |  |
| --- | --- |
| Название | Belarus scientist’s |
| Правило | (SCIENCE OR SCIENTIST) AND BELARUS |
| Шаблон | ( 3 | 4 ) 1 |
| Шаблон  с использованием идентификаторов | 3 1 |
| Нормализованный шаблон  с использованием идентификаторов | 1 3 |

Нормализованный шаблон категории, это шаблон категории с упорядоченными идентификаторами ключевых слов. Для проверки соответствия категории, массив идентификаторов ключевых слов для каждого предложения нормализуется (сортируется по значениям идентификаторов), затем для него запускается конечный автомат ля поиска соответствия категории.



* + - 1. - Конечный автомат для категории с шаблоном «1 3»

# РЕАЛИЗАЦИЯ И ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## Основные принципы реализации программного средства

Для обеспечения высокого качества программного средства, было принято решения использовать Test Drive Development, когда написание тестов предшествует написанию кода. Так как в процессе реализации программы, использовалась относительно новая технология CUDA, часть тестов составляли «смоук» тесты, для проверки работы и определения всех устройств в гетерогенной компьютерной системе. После определения базового функционала, были написаны тесты для проверки такой функциональности как выделение слов в тексте, поиск слов по словарю. Для более полного покрытия было написано по два варианта основного функционала: для выполнения в гетерогенной компьютерной системе(NVIDIA GPU и AMD CPU), и гомогенной (AMD CPU). Впоследствии данные тесты смогут помочь оценить достигнутого изменения производительности при использовании GPU для общих вычислений.

## Обоснование технических приемов программирования

В качестве основной технологии параллельного программирования в гетерогенной системе была выбрана технология NVIDIA CUDA [17], так как на данный момент она является наиболее выгодной с точки зрения соотношения удобство разработки и обеспечиваемая скорость разрабатываемого программного средства. В качестве языка разработки С++ и его расширение CUDA C. Специально для разработчиков, использующих CUDA технологии, NVIDIA представила для закрытого бета тестирования расширение для Microsoft Visual Studio под название Parallel Nsight (в прошлом имевшем название Nexus) [16]. С помощью его разработчик имеет возможность для удаленной отладки кода, выполняющегося на вычислительных устройствах NVIDIA GPU. Системные требования этого расширения ограничили выбор возможных операционных систем: Windows Vista и Windows 7. Для контроля версия разрабатываемого программного средства в качестве серверной части был выбран сервис Code Google, в качестве клиентской: Tortoise SVN. Это позволило поддерживать в актуальном состоянии исходный код при разработке программного средства, вести запись изменений.

## Общая структура реализованного программного средства

Программное средство параллельной обработки данных в гетерогенной компьютерной системе представляет собой набор библиотек, часть из которых является специфическими для данной задачи:

- labelsCuda;

- wordFinder;

- cudpp.

Часть являются стандартными библиотеками обеспечивающими работу CUDA технологии:

-cudart;

-cutil32;

-cuda;

Интерфейс основной библиотеки представляет собой экспортируемые функции и классы:

Создание класса для поиска слов:

WordFinder \* CreateWordFinder();

class WordFinder

{

//Добавление слова в словарь

void AddWord(std::string word, int id);

//Добавление списка слов в словарь

void AddWords(std::vector<std::string> words);

//Генерация таблицы переходов дял конечного автомата

TransitionsTable\* Generate();

WordFinder(void);

~WordFinder(void);

};

Функции поиска слов:

Подготовка к поиску (выделение нужных буферов в памяти, создание и инициализация необходимых структур и объектов)

void deviceFindAllWordsPrepare(TransitionsTable \* transTable, size\_t len);

Поиск слов:

void deviceFindAllWords(char\* text, int len, Word\* words, int\* count, int \* allWords, int\* allCount);

## Тестирование, экспериментальные исследования и анализ полученных результатов

Для проверки правильности поиска ключевых слов были разработаны тестовые последовательности:

* + - * 1. – Основные тестовые случаи

| Иденти-фикатор тесто-вого случая | Описание тестового случая | Ожидаемые результаты |
| --- | --- | --- |
| W1 | Текст состоит из одного ключевого слова  . | Количество найденных ключевых слов 1 |
| W2 | Текст состоит из одного ключевого слова отличающегося первым символом | Количество найденных ключевых слов 0 |
| W3 | Текст состоит из одного ключевого слова отличающегося последним символом | Количество найденных ключевых слов 0 |
| W4 | Текст состоит из одного ключевого слова отличающегося наличием дополнительного символа в конце слова | Количество найденных ключевых слов 1 |
| W5 | Текст состоит из одного ключевого слова отличающегося отсутствием одного символа | Количество найденных ключевых слов 0 |
| W6 | Текст состоит из двух ключевых слов без разделителя | Количество найденных ключевых слов 1 |
| W7 | Текст состоит из двух ключевых слов с разделителем | Количество найденных ключевых слов 2 |
| W8 | Текст состоит из двух ключевых слов с разделителем на позиции меньше длины первого слова | Количество найденных ключевых слов 0 |
| W9 | Текст состоит из двух ключевых слов с множеством разделителей | Количество найденных ключевых слов 2 |
| W10 | Текст состоит из трех ключевых слов с разделителем | Количество найденных ключевых слов 3 |
| W11 | Текст состоит из трех ключевых слов с множество разделителей | Количество найденных ключевых слов 3 |

Для анализа производительности разработанного программного средства (использует GPU NVIDIA GeForce GT220), была реализована эквивалентная функциональность, выполняющаяся последовательно в гомогенной компьютерной среде при использовании только центрального процессора(CPU Athlon2 X2 215) и проведены замеры времени необходимого для обработки 1000 текстовых файлов размером от 4КБ до 4МБ.

* + - * 1. - Сравнительное время обработки текстовых файлов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер файла, КБ | Время работы CPU, сек | Время работы GPU, сек | Пропускная способность CPU, МБ/c | Пропускная способность GPU, МБ/c | Увеличение скорости, % |
| 4 | 0,1 | 1,2 | 40,00 | 3,33 | -91,67 |
| 8 | 0,2 | 1,4 | 40,00 | 5,71 | -85,71 |
| 16 | 0,4 | 1,3 | 40,00 | 12,31 | -69,23 |
| 32 | 0,7 | 1,6 | 45,71 | 20,00 | -56,25 |
| 64 | 1,5 | 1,8 | 42,67 | 35,56 | -16,67 |
| 128 | 2,9 | 2,3 | 44,14 | 55,65 | 26,09 |
| 256 | 5,8 | 3,2 | 44,14 | 80,00 | 81,25 |
| 512 | 11,7 | 5,3 | 43,76 | 96,60 | 120,75 |
| 1024 | 23,7 | 9,3 | 43,21 | 110,11 | 154,84 |
| 2048 | 48,4 | 17 | 42,31 | 120,47 | 184,71 |
| 4096 | 98,6 | 32,5 | 41,54 | 126,03 | 203,38 |

* + - 1. - Зависимость времени обработки от размера файла
      2. - Зависимость пропускной способности от размера обрабатываемых файлов
      3. - Зависимость достигаемого ускорения от размера обрабатываемых файлов

Как видно из полученных результатов, при малых размерах обрабатываемых файлов параллельная обработка, из-за накладных расходов и не максимальной утилизации всех вычислительных устройств, занимает больше последовательной. При увеличении объемов данных, параллельная обработка данных, за счет преимуществ использования нескольких вычислительных ядер и более полной утилизации позволяет достичь прироста производительности более 200% или получить пропускную способность порядка 125 МБ/сек, что составляет больше чем максимальная пропускная способность (1Гб/c) большинства локальных сетей.

Стоит заметить, что по данным статистики даже при тестах последовательного выполнения, центральный процессор был нагружен на 50%(задействовано полностью одно ядро из двух), при тестах параллельного выполнения графический адаптер был утилизировано только на 20%, что говорит о большом потенциале, и возможностях по оптимизации и дополнительному ускорению данного алгоритма.

# ЭНЕРГОСБЕРЕЖЕНИЕ. СОКРАЩЕНИЕ ЭНЕРГОЗАТРАТ ПРИ ВНЕДРЕНИИ ПРОЕКТИРУЕМОЙ АВТОМАТИЗИРОВАННОЙ СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ

В результате разработки данного дипломного проекта, получено программное средство, позволяющие эффективно в реальном времени обрабатывать параллельно поток текстовых документов. Использования преимуществ многопоточных вычислительных систем позволяет получить значительный прирост производительности на системе со специализированным вычислительным процессором по сравнению с процессором общего назначение.

Принятые в расчетах допущения:

- учитываются энергозатраты только при эксплуатации. Так как необходимые при разработке проектируемой системы ресурсы и оборудование не сильно отличаются количественно и качественно от ресурсов, используемых для разработки других систем, далее рассчитываются без учета энергозатрат при разработке;

- энергозатраты при эксплуатации составляют в основном только затраты электрической энергии на работу компьютерных систем. Так как проектируемая система не требует прямого вмешательства оператора в работу системы, далее они рассчитываются без учета энергозатрат для обеспечения эргономических требований систем «человек-машина»;

- аппаратно компьютерная система проектируемой автоматизированной системы обработки информации отличается от других систем только наличием специализированного процессорного модуля GPU NVIDIA GT220;

- для обеспечения необходимой производительности автоматизированная система обработки информации должна обеспечивать обработку 100 текстовых файлов в секунду размером от 80Кб до 160Кб;

- скорость обработки данных на CPU Athlon 2 X2 215 8Мб/с, на GPU NVIDIA GeForce GT 220 – 20Мб/с.

Рассчитаем энергозатраты при эксплуатации систем обработки информации основанных на процессорах общего назначения.

Для оценки эффективности энергосбережения проведем анализ и расчет энергозатрат при эксплуатации автоматизированных систем обработки информации основанных на процессоре общего назначения AMD Athlon 2 X2 215.

Для процессора воспользуемся спецификацией предоставляемой производителем [24].

За среднюю потребляемую мощность остального оборудования для одной компьютерной системы возьмем 100 Вт [25].

* + - * 1. - Краткая спецификация процессора AMD Athlon 2 X2 215

|  |  |
| --- | --- |
| Процессор | AMD Athlon™ II X2 |
| Модель | 215 |
| Частота | 2700 МГц |
| Напряжение | 0.85 - 1.425 В |
| Макс. температура | 74 C |
| Макс. мощность | 65 Вт |
| Технологический процесс | 45 нм |
| Количество вычислительных ядер | 2 |

Исходя из того что вся информация должна быть обработана, рассчитаем суммарную необходимую пропускную способность:

, (6.1)

где Throughput – необходимая суммарная пропускная способность (Мб/с);

N – количество обрабатываемых текстовых файлов;

t – время затраченное на обработку (с);

Size – средний размер файла (Кб).

Средний размер файла вычисляется по формуле

где – минимальный размер текстового файла (Кб);

– максимальный размер текстового файла (Кб).

Получаем:

По (6.1) получаем, что необходимо обеспечить скорость обработки данных:

С учетом того, что скорость обработки данных с использованием одного процессора общего назначение = 8 Мб/с, количество необходимых компьютерных систем:

где – необходимая скорость обработки (Мб/с);

- скорость обработки, обеспечиваемая одной компьютерной системой на основе процессора общего назначения (Мб/с).

.

Таким образом, требуется как минимум 2 компьютерные системы на базе процессора общего назначение AMD Athlon™ II X2.

Для расчета потребляемой мощности используем формулу:

, (6.4)

где – потребляемая мощность оборудования для одной компьютерной системы;

– потребляемая мощность процессора общего назначения.

По данным [25] = 100 Вт, по данным производителя = 65 Вт, тогда общая потребляемая мощность системы:

.

Таким образом, потребляемая мощность системы, основанной на процессорах общего назначение и удовлетворяющая заданным условиям, 330 Вт.

Рассчитаем энергозатраты при эксплуатации систем обработки информации основанных на процессорах специального назначения.

Для оценки эффективности энергосбережения проведем анализ и расчет энергозатрат при эксплуатации проектируемой автоматизированной системы обработки информации, основанной на процессоре специального назначения NVIDIA GeForce GT 220

Для процессора воспользуемся спецификацией предоставляемой производителем [26].

За среднюю потребляемую мощность остального оборудования для одной компьютерной системы возьмем 100 Вт [25].

* + - * 1. – Краткая спецификация NVIDIA GeForce GT 220

|  |  |
| --- | --- |
| Процессор | NVIDIA GeForce |
| Модель | GT220 |
| Частота | 625 МГц |
| Напряжение | 3 В |
| Макс. температура | 105 C |
| Макс. мощность | 58 Вт |
| Технологический процесс | 40 нм |
| Количество вычислительных ядер | 48 |

Количество необходимых компьютерных систем:

где – требуемая скорость обработки (Мб/с);

- скорость обработки, обеспечиваемую одной компьютерной системой на основе процессора специального назначения (Мб/с).

С учетом того, что скорость обработки данных с использованием одного процессора специального назначение = 20 Мб/с, а = 12 Мб/с (по формуле (6.5)):

Таким образом, требуется только одна компьютерная системы на базе процессора специального назначение NVIDIA GeForce GeForce GT220 для обеспечения необходимой производительности.

Для расчета потребляемой мощности используем формулу:

, (6.6)

где – потребляемая мощность оборудования для одной компьютерной системы;

– потребляемая мощность процессора общего назначения;

- потребляемая мощность процессора специального назначения.

По данным [25] = 100 Вт, по данным производителя = 65 Вт [24], = 58 Вт [26] тогда общая потребляемая мощность системы:

Таким образом, потребляемая мощность системы основанной на процессорах специального назначение 223 Вт.

Рассчитаем эффективность сокращения энергозатрат при использовании проектируемой автоматизированной системы.

Для оценки эффективности сокращения энергозатрат, рассчитаем относительную экономию энергоресурсов на основе потребляемой системой мощности:

где – относительная эффективность сокращения энергозатрат;

- потребляемая мощность до внедрения проектируемой системы;

- потребляемая мощность после внедрения проектируемой системы.

Тогда,

Спроектированная автоматизированная система, основанная на процессорах NVIDIA GeForce GT220, позволяет при сохранении заданной производительности обеспечивать при эксплуатация сокращение энергозатрат на 32%.

# ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И ИСПОЛЬЗОВАНИЯ ПРОГРАММНОГО СРЕДСТВА

## Краткая характеристика программного средства

В результате данного дипломного проекта был спроектирован программный модуль обработки больших массивов текстовой информации, который может использоваться в других системах для сокращения затрат времени на обработку различных текстовых документов и информационных источников.

Система позволяет выделять в тексте слова, производить поиск цитат, относить документ к определенному типу на основании заданных правил.

Разработка данного проекта и разработка проектов программных средств в целом связана со значительными затратами трудовых, материальных, финансовых и других ресурсов. В связи с этим создание и реализация проекта программного обеспечения (ПО) нуждается в соответствующем технико-экономическом обосновании (ТЭО).

Целью технико-экономического обоснования является расчет экономической эффективности разработки программного средства. Разработка программных средств происходит со значительными затратами трудовых, материальных, финансовых ресурсов. В связи с этим при создании и реализации каждого проекта программного обеспечения необходимо соответствующее технико-экономическое обоснование.

Методика расчета основывается на том, что для разработчика экономический эффект выступает в виде чистой прибыли, остающейся в распоряжении предприятия от реализации программного модуля, а для пользователя – в виде экономии трудовых, материальных и финансовых ресурсов. Расчеты выполнены на основе методического пособия [27]

Экономические параметры проекта вычислены по данным на 1 мая 2010 года.

## Смета затрат и цена программного обеспечения

Экономический эффект у пользователя выражается в экономии трудовых, материальных и финансовых ресурсов.

Стоимостная оценка ПО и определение экономического эффекта у разработчика предполагает составление сметы затрат. Все исходные данные, используемы в последствии для расчета сметы затрат заказного ПО, представлены в таблице 7.1.

* + - * 1. – Исходные данные

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование показателей | Буквенное  обозначение | Единицы  измерения | Количество |
| 1 | 2 | 3 | 4 |
| Месячная тарифная ставка 1-го разряда | Тм1 | руб. | 81 000 |
| Среднемесячная норма рабочего времени | Фр | - | 170 |
| Продолжительность рабочего дня | Тч | ч | 8 |
| Коэффициент премирования | К | - | 1,4 |
| Норматив дополнительной заработной платы исполнителей |  | % | 10 |
| Норматив отчислений в фонд социальной защиты населения |  | % | 34 |
| Средний расход материалов |  | руб./100  строк кода | 0,38 |
| Цена одного машино-часа |  | руб. | 700 |
| Норматив расхода машинного времени на отладку 100 строк исходного кода |  | машино-часов | 12 |
| Норматив командировочных расходов |  | % | 30 |
| Норматив прочих затрат |  | % | 20 |
| Норматив накладных расходов |  | % | 100 |
| Ставка налога на добавленную стоимость |  | % | 20 |
| Норматив расходов на освоение ПО |  | % | 10 |
| Норматив расходов на сопровождение ПО |  | % | 10 |
| Отчисления в Белгосстрах | Нгс | % | 0.3 |
| Уровень рентабельности |  | % | 15 |
| Налог на прибыль |  | % | 24 |

### Определение объема и трудоемкости ПО

Базой для расчета плановой сметы затрат на разработку ПО является объем ПО. Общий объем (Vо) программного продукта определяется исходя из количества и объема функций, реализуемых программой:

 (7.1)

где Vi – объем отдельной функции ПО;

n – общее число функцией.

Все функции, реализованные в проекте модуля обработки текстовой информации, представлены в таблице 7.2.

* + - * 1. – Функции программного обеспечения

|  |  |  |
| --- | --- | --- |
| № | Наименование (содержание) функций | Объем функции Vi |
| 1 | 2 | 3 |
| 101 | Организация ввода информации | 150 |
| 102 | Контроль, предварительная обработка и ввод информации | 450 |
| 305 | Обработка файлов | 720 |
| 307 | Совместная обработка группы файлов | 6180 |
| 309 | Формирование файла | 1020 |
| 506 | Обработка ошибочных и сбойных ситуаций | 410 |
| 507 | Обеспечение интерфейса между компонентами | 970 |

*Vо = 150 + 450 + 720 + 6180 + 1020 + 410 + 970 = 9900*

По общему объему ПО Vо и нормативам затрат труда в расчете на единицу объема определяются нормативная и общая трудоемкость разработки ПО.

Нормативная трудоемкость (Тн) разработки ПО определяется на основании общего объема и с учетом категории сложности ПО.

Разработанный проект относится к 1-ой категории сложности (так как обеспечивает существенное распараллеливание вычислений).

Общая трудоемкость рассчитывается по формуле:

То= Тн · Кс ·Кт ·Кн, (7.2)

где Тн – нормативная трудоемкость;

Кс – коэффициент, учитывающий сложность ПО;

Кт – поправочный коэффициент, учитывающий степень использования при разработке стандартных модулей;

Кн – коэффициент, учитывающий степень новизны ПО.

В таблице 7.3 описаны все коэффициенты и нормативы, используемые в расчетах.

*То = 327 ⋅ 1,07 ⋅ 1 ⋅ 0,9 = 315 (чел/дн).*

На основе общей трудоемкости определяется плановое число разработчиков (Чр) и плановые сроки, необходимые для реализации проекта в целом (Тр).

* + - * 1. – Исходные данные

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование показателей | Обозна-чение | Единицы измерения | Значение |
| 1 | 2 | 3 | 4 |
| Категория сложности |  | - | 1 |
| Нормативная трудоемкость | Тн | чел/дн | 327 |
| Коэффициент сложности ПО | Кс | - | 1,07 |
| Коэффициент, учитывающий степень использования при разработке стандартных модулей | Кт | - | 1 |
| Коэффициент, учитывающий степень новизны ПО | Кн | - | 0,9 |
| Эффективный фонд времени | Фэф | дн. | 249 |
| Срок разработки проекта | Тр | лет | 0,5 |

Численность исполнителей проекта (Чр) рассчитывается по формуле:

 (7.3)

где То – общая трудоемкость разработки проекта (чел/дн.);

Тр – срок разработки проекта (лет);

Фэф – эффективный фонд времени работы одного работника в течение года (дн.).

*Чр = 315 / (0,5 ⋅ 249) = 3 чел.*

Если рассматривать разработку программного модуля учета и конвертации библиографической информации как совокупность определенных стадий, то трудоемкость следует определять отдельно на каждой из следующих стадий:

1) техническое задание (ТЗ);

2) эскизный проект (ЭП);

3) технический проект (ТП);

4) рабочий проект (РП);

5) внедрение (ВН).

Общая трудоемкость рассчитывается с учетом распределения ее по стадиям:

 (7.4)

где То – общая трудоемкость разработки ПО (чел/дн.);

Тi – трудоемкость разработки ПО на i-й стадии (чел/дн.);

n – количество стадий разработки.

Трудоемкость стадий определяется по формуле:

Тстi = Тн ⋅ dcтi ⋅ Kс ⋅ Kн ⋅ Кт, (7.5)

где Туi – трудоемкость разработки ПО на i-й стадии;

Тн – нормативная трудоемкость;

dстi – удельный вес трудоемкости i-й стадии разработки ПО в общей трудоемкости разработки ПО;

Кс – поправочный коэффициент, учитывающий сложность ПО;

Кт – поправочный коэффициент, учитывающий степень использования при разработке стандартных модулей;

Кн – коэффициент, учитывающий степень новизны ПО.

Результаты расчета уточненной трудоемкости ПО и численности исполнителей по стадиям представлены в Таблица 7.4.

* + - * 1. – Расчет уточненной трудоемкости ПО и численности исполнителей по стадиям

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Показатели | Обозна-чение | Стадии | | | | | Итого |
| ТЗ | ЭП | ТП | РП | ВН |
| Уточненная трудоемкость (чел/дн) | Туi | 32 | 25 | 28 | 183 | 47 | 315 |
| Удельный вес трудоемкости | dстi | 0,10 | 0,08 | 0,09 | 0,58 | 0,15 | 1 |
| Коэффициент учитываю­щий использование типо­вых программ | Кт | - | - | - | 1 | - | - |
| Коэффициент новизны | Кн | 0,9 | 0,9 | 0,9 | 0,9 | 0,9 | - |
| Срок разработки (лет) | ТР | - | - | - | - | - | 0,5 |
| Численность исполнителей (чел) | Чр | - | - | - | - | - | 3 |

Общая численность исполнителей проекта составит:

*ЧР =315 / (0,5 ⋅ 249) ≈ 3 (чел).*

### Расчет сметы затрат и цены заказного ПО

Основной статьей расходов на создание ПО является заработная плата разработчиков.

Общая трудоемкость, плановая численность работников и плановые сроки разработки ПО являются базой для расчета основной заработной платы разработчиков проекта. По данным о специфике и сложности выполняемых функций составляется штатное расписание группы специалистов-исполнителей, участвующих в разработке ПС, с определением образования, специальности, квалификации, должности и тарифного коэффициента (см. таблицу 7.5).

Месячная тарифная ставка каждого исполнителя (Тм) определяется по формуле:

 (7.6)

где ТМ1 – действующая месячная тарифная ставка 1-го разряда;

ТК – тарифный коэффициент, соответствующий установленному тарифному разряду.

* + - * 1. – Расчет месячных и почасовых тарифных ставок

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Должность | Количество сотрудников | Тарифный разряд | Тарифный коэффициент | Месячная тарифная ставка (тыс. руб.) | Часовая тарифная ставка (руб.) |
| Ведущий  инженер-программист | 1 | 12 | 2,84 | 230 | 1 352,94 |
| Инженер-программист  I-ой категории | 2 | 11 | 2,65 | 214,7 | 1 262,94 |

Часовая тарифная ставка рассчитывается путем деления месячной тарифной ставки на установленную при 40-часовой недельной норме рабочего времени расчетную среднемесячную норму рабочего времени в часах (Фр):

 (7.7)

где ТМ – месячная тарифная ставка (тыс. руб.).

Для инженера-программиста I-ой категории часовая тарифная ставка составляет:

*Тч = 214 700 / 170 = 1 262,94 (руб.).*

Для ведущего инженера-программиста часовая тарифная ставка составляет:

*Тч = 230 000 / 170 = 1 352,94 (руб.).*

Основная заработная плата исполнителей на конкретное ПО рассчитывается по формуле:

 (7.8)

где n – количество исполнителей;

ТЧi – часовая тарифная ставка i-го исполнителя;

Тч – количество часов работы в день;

Фп – плановый фонд рабочего времени i-го исполнителя (дней);

К – коэффициент премирования.

Для инженеров-программистов плановый фонд рабочего времени равен *Фп* = 125 дней; для ведущего инженера-программиста плановый фонд рабочего времени равен *Фп* = 315 – 125\*2=65 дня.

*Зо = 2 ⋅ (1 262,94 ⋅ 8 ⋅ 125 ⋅ 1,4) + (1 352,94 ⋅ 8 ⋅ 65 ⋅ 1,4)=*

*4 521 172*

Дополнительная заработная плата (Здi) включает выплаты, предусмотренные законодательством о труде, и определяется по формуле:

 (7.9)

где НД – норматив дополнительной заработной платы.

*Зд = 4 521 172 ⋅ 10 / 100 = 452 117 (руб.).*

Сумма отчислений в фонд социальной защиты населения определяется по формуле:

 (7.10)

где НСЗ – норматив отчислений в фонд социальной защиты населения.

*Зсз = (4 521 172+ 452 117) ⋅ 35 / 100 = 1 740 651 (руб.).*

Отчисления в Белгосстарх (Згс) рассчитываются по формуле:

, (7.11)

где Нгс – норматив отчислений в Белгосстрах от несчастных случаев на производстве (%).

*Згс = (4 521 172+ 452 117) ⋅ 0,3 / 100 = 14 920 (руб.).*

Расходы по статье “Материалы” определяются на основании сметы за­трат с учетом действующих нормативов. По данной статье отражаются рас­ходы на магнитные носители, бумагу, красящие ленты и другие материалы, необходимые для разработки ПО. Сумма затрат на материалы определяется следующим образом:

 (7.12)

где Нм – норма расхода материалов в расчете на 100 строк кода ПО (руб.);

Vоi – общий объем ПО (строк исходного кода).

*М = 0,38 ⋅ 9900/ 100 = 38 (руб.).*

Расходы по статье “Машинное время” включают оплату машинного времени, необходимого для разработки и отладки ПО, которое определяется по нормативам (в машино-часах) на 100 строк исходного кода (Нмв) машинного времени в зависимости от характера решаемых задач и типа ПК.

 (7.13)

где Vоi – общий объем программного средства (строк исходного кода);

Цм – цена одного машино-часа (руб.);

Нмв – норматив расхода машинного времени на отладку 100 строк исходного кода (машино-часов).

*Рм = 700 ⋅ 9900/100 ⋅ 12 = 831 600 (руб.).*

Расходы по статье «Научные командировки» (Рнкi) определяются по нормативу, разрабатываемому в целом по организации, в процентах к основной заработной плате:

 (7.14)

где Нрнк – норматив расходов на командировки в целом по организации (%).

*Рнк = 4 521 172 ⋅ 30 / 100 = 1 356 351 (руб.).*

Прочие затраты включают затраты на приобретение и подготовку специальной научно-технической информации и специальной литературы определяются, по формуле:

 (7.15)

где Нпз – норматив прочих затрат в целом по организации (%).

*Пз = 4 521 172 ⋅ 20 / 100 = 904 234 (руб.).*

Затраты по статье «Накладные расходы», связанные с необходимостью содержания аппарата управления, вспомогательных хозяйств и опытных производств, рассчитываются по формуле:

 (7.16)

где Нрн – норматив накладных расходов в целом по организации (%).

*Рн = 4 521 172 ⋅ 100 / 100 = 4 521 172 (руб.).*

Общая сумма расходов по смете на ПО рассчитывается по формуле:

 (7.17)

*Ср=* *4 521 172 + 452 117 + 1 740 651 + 14 920 + 38 + 831 600 +*

*1 356 351 + 904 234 + 4 521 172 = 14 342 255 (руб.).*

Затраты на освоение ПО (Роi) рассчитываются по формуле:

 (7.18)

где Но – норматив расходов на освоение (%).

*Ро = 14 342 255 ⋅ 10 / 100 = 1 434 225 (руб.).*

Затраты на сопровождение ПО (Рсi) рассчитываются по формуле:

 (7.19)

где Нс – норматив расходов на сопровождение (%).

*Рс = 14 342 255 ⋅ 20 / 100 = 2 868 450 (руб.).*

Общая сумма расходов на разработку ПО, включая расходы на освоение и сопровождение, будет следующая:

 (7.20)

*Сп = 14 342 255 + 1 434 225 + 2 868 450 = 18 644 932 (руб.).*

Прибыль от реализации ПО рассчитывается по формуле:

 (7.21)

где Поi – прибыль от реализации ПО заказчику;

Срi – себестоимость ПО;

Урпi – уровень рентабельности ПО (%).

*По = 18 644 932 ⋅ 15 / 100 = 2 796 740(руб.).*

Прогнозируемая цена ПО без налогов (Цпi) рассчитывается по формуле:

 (7.22)

*Цпi = 18 644 932 + 2 796 740 = 21 441 672 (руб.).*

Налог на добавленную стоимость (НДСi), рассчитывается по формуле:

, (7.23)

где Ндс – норматив НДС (%).

*НДС = 21 441 672 ⋅ 20 / 100 = 4 288 334 (руб.).*

Прогнозируемая отпускная цена (Цоi) рассчитывается по формуле:

 , (7.24)

*Цоi = 21 441 672 + 4 288 334 = 25 730 000 (руб.).*

Чистая прибыль, остающаяся в распоряжении предприятия (ПЧ) за вычетом налога на прибыль, определяется по формуле:

 (7.25)

*Пч = 2 796 740 – (2 796 740 ⋅ 24 / 100) = 2 125 522 (руб.).*

Чистая прибыль от реализации ПО, равная *2 125 522* рублей представляет собой экономический эффект от создания нового проекта.

Полученные показатели представлены в таблице 7.6.

* + - * 1. – Результаты расчетов

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | Обозначение | Единицы измерения | Сумма |
| Основная зар. плата | Зо | руб. | 4 521 172 |
| Дополнительная зар. плата | Зд | руб. | 452 117 |
| Отчисления в фонд социальной защиты населения | Зсз | руб. | 1 740 651 |
| Расходы по статье Материалы | М | руб. | 38 |
| Расходы по статье Машинное время | Рм | руб. | 831 600 |
| Расходы по статье Научные командировки | Рнк | руб. | 1 356 351 |
| Расходы по статье Прочие затраты | Пз | руб. | 904 234 |
| Затраты по статье Накладные расходы | Рн | руб. | 4 521 172 |
| Затраты на освоение ПО | Ро | руб. | 1 434 225 |
| Затраты на сопровождение ПО | Рс | руб. | 2 868 450 |
| Общая сумма расходов на разработку ПО | Сп | руб. | 18 644 932 |
| Прибыль от реализации ПО | По | руб. | 2 796 740 |
| Прогнозируемая цена ПО без налогов | Цп | руб. | 21 441 672 |
| Отчисления в Белгосстрах | Згс | руб. | 14 920 |
| Налог на добавленную стоимость | НДС | руб. | 5 045 547 |
| Прогнозируемая отпускная цена | Цо | руб. | 25 730 000 |
| Чистая прибыль | Пч | руб. | 2 125 522 |

## Оценка экономической эффективности применения ПО у пользователя

Для определения экономического эффекта от использования нового ПО у потребителя необходимо сравнить расходы по всем основным статьям сметы затрат на эксплуатацию нового ПО с расходами по соответствующим статьям базового варианта. При этом создание нового ПО окажется экономически целесообразным лишь в том случае, если все капитальные затраты окупятся за счет получаемой экономии в ближайшие 1–2 года.

Исходные данные для оценки экономической эффективности применение ПО у пользователя представлены в таблице 7.7.

Общие капитальные вложения (Ко) заказчика (потребителя), связанные с приобретением, внедрением и использованием ПО, рассчитываются по формуле:

 (7.26)

где Кпр – затраты пользователя на приобретение ПО по отпускной цене у разработчика с учетом стоимости услуг по эксплуатации (руб.);

Кос – затраты пользователя на освоение ПО (руб.);

Кс – затраты пользователя на оплату услуг по сопровождению ПО (руб.);

Ктс – затраты на доукомплектование ВТ техническими средствами в связи с внедрением нового ПО (руб.);

Коб – затраты на пополнение оборотных средств в связи с использованием нового ПО (руб.).

* + - * 1. – Исходные данные

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Наименование показателей | Обозначение | Единицы измерения | Значение показателя | |
| в базовом варианте | в новом варианте |
| Затраты пользователя на приобретение ПО | Кпр | руб. |  | 25 730 000 |
| Затраты на освоение ПО | Кос | руб. |  | 1 434 225 |
| Затраты на сопровождение ПО | Кс | руб. |  | 2 868 450 |
| Затраты на доукомплектование ВТ техническими средствами | Ктс | руб. |  | 1 434 225 |
|
| Затраты на пополнение оборотных средств в связи с эксплуатацией нового ПО | Коб | руб. |  | 100 000 |
|
| Время простоя сервиса | П1, П2 | мин | 11 | 1 |
| Стоимость одного часа простоя | Сп | руб. | 400 000 | 400 000 |
| Ставка налога на прибыль | Нп | % |  | 24 |

*Ко = 25 730 000 + 1 434 225 + 2 868 450 + 1 434 225+ 100 000 =*

*= 31 615 519 (руб.)*

Экономия за счет простоев сервиса (Сс):

 (7.27)

где П1, П2 – время простоя сервиса, обусловленное ПО, в день;

Дрг – плановый фонд работы сервиса (дней);

Сп – стоимость одного часа простоя (руб.).

*Сс = (11 - 1) ⋅ 225 ⋅ 400 000 / 60 = 15 000 000 (руб.)*

Общая годовая экономия текущих затрат, связанных с использованием нового ПО (Со), рассчитывается по формуле:

 (7.28)

*Со = 15 000 000 (руб.).*

Расчет экономического эффекта. Внедрение нового ПО позволит пользователю сэкономить на текущих затратах, то есть практически получить на эту сумму дополнительную прибыль. Для пользователя в качестве экономического эффекта выступает лишь чистая прибыль – дополнительная прибыль, остающаяся в его распоряжении (ΔПч), которые определяются по формуле:

где Нп – ставка налога на прибыль (%).

*ΔПч = 15 000 000 – 15 000 000 ⋅ 24 / 100 = 11 400 000(руб.).*

Данная получаемая прибыль остается в распоряжении пользователя и выступает в качестве экономического эффекта.

В процессе использования нового ПС чистая прибыль в конечном итоге возмещает капитальные затраты. Однако полученные при этом суммы прибыли и затрат по годам приводят к единому времени - расчетному году (за расчетный год принят 2010 год) путем умножения результатов и затрат за каждый год на коэффициент приведения (), который рассчитывается по формуле:

, (7.30)

где  – норматив приведения разновременных затрат и результатов;

 – расчетный год,  1;

 – номер года, результаты и затраты которого приводятся к расчетному (2010-1, 2011-2, 2012-3, 2013-4).

Норматив приведения разновременных затрат и результатов () для программных средств ВТ в существующей практике принимается равным 0.12.

Из приведенной ниже таблицы видно, что превышение результата над затратами происходит уже через 2 год.

* + - * 1. – Расчет экономического эффекта от использования нового ПО

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Показатели | Единицы измерения | 2010 год | 2011 год | 2012 год | 2013 год |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Прирост прибыли за счет экономии затрат (Пч) | руб. | 11 400 000 | 11 400 000 | 11 400 000 | 11 400 000 |
| То же с учетом фактора времени | руб. | 11 400 000 | 10 180 200 | 9 085 800 | 8 116 800 |
| Затраты на приобретение ПО (Кпр) | руб. | 25 778 619 | - | - | - |
| Затраты на освоение ПО (Кос) | руб. | 1 434 225 | - | - | - |
| Затраты на сопровождение ПО (Кс) | руб. | 2 868 450 | - | - | - |
| Затраты на доукомплектование ВТ техническими средствами (Ктс) | руб. | 1 434 225 | - | - | - |
| Затраты на пополнение оборотных средств (Коб) | руб. | 100 000 | - | - | - |
| Всего затрат | руб. | 31 615 519 | - | - | - |
| То же с учетом фактора времени | руб. | 31 615 519 | - | - | - |
| Превышение результата над затратами | руб. | -20 215 519 | 10 180 200 | 9 085 800 | 8 116 800 |
| То же с нарастающим итогом | руб. | -20 215 519 | -14 035 319 | -4 949 519 | 3 167 281 |
| Коэффициент  дисконтирования | - | 1 | 0,893 | 0,797 | 0,712 |

 – расчетный год;

 – 2011 год;

 – 2012 год;

 – 2013 год.

Таким образом, приобретение нового ПО для пользователя окажется экономически эффективным и целесообразным.

Для расчёта окупаемости и рентабельности ПС необходимо определить период, за который первоначальные капитальные вложения в проект будут возмещены за счет прибыли, получаемой от проекта.

Точный срок окупаемости проекта можно определить по формуле:

 (7.31)

где Ток – точный срок окупаемости проекта;

Тд – дробная часть шага периода окупаемости;

Тц – количество целых шагов расчетного периода.

Рассчитаем показатель, характеризующий эффективность проекта по уровню доходов на единицу затрат, то есть индекс рентабельности проекта (Ри).

 , (7.32)

где αt – коэффициент дисконтирования.

Данный показатель используется при принятии решений об инвестировании проектов. Условия принятия проекта для инвестирования:

Pи > 1 − проект следует принять;

Pи < 1 − проект следует отвергнуть;

Pи = 1 − проект не прибыльный и не убыточный.

*Ри = (11 400 000 + 10 180 200 +9 085 800 +8 116 800 )/*  31 615 519 *=1.095*

*.*

Так как Ри >1, то проект следует принять.

Проект характеризуется равномерным поступлением чистого дохода, поэтому срок окупаемости следует определять по формуле:

где – первоначальные инвестиции;

- среднегодовая прибыль.

= 20 215 519 / 9 695 700 = 2.26 (года)

Капитальные вложения в проект окупятся за счет чистой прибыли от реализации ПО менее чем через 3 года.

## Выводы

В процессе технико-экономического обоснования инвестиционного проекта по внедрению программного модуля параллельной обработки текста с использованием процессора специального назначение были получены следующие результаты:

1) интегральный экономический эффект от внедрения в производство изделия за 3 года составит 3 167 281руб;

2) срок окупаемости инвестиций составляет 2 года и 3 месяцев;

Полученные результаты свидетельствуют о том, что данный проект является экономически целесообразным и его реализация принесет предприятию коммерческий успех.

# ЗАКЛЮЧЕНИЕ

В результате проделанной работы было изучена проблема обработки текста и некоторые основные методы ее решения и подходы. Так же были рассмотрены различные парадигмы параллельного программирования в гетерогенной компьютерной системе. Была собрана информация для выбора оптимальных методов обработки текста и возможные их совершенствования под конкретную задачу, а так же архитектуры с использованием которой может реализовать эффективную разработку, поддержку, и высокое качество конечной программно-аппаратной системы. Была изучена и применена программно-аппаратная архитектура NVIDI CUDA, на основе которой было разработано программное средство, позволяющее эффективно обрабатывать большие объемы текстовой информации, за счет использования преимуществ многоядерных систем. В процессе разработки были использованы различные модификации шаблонов параллельного программирования, которые позволили добиться эффективного использования имеющихся аппаратных и энергетических ресурсов. Так же были экспериментально подтверждены теоретические предположения о возможных особенностях работы параллельных алгоритмов в гетерогенных компьютерных системах. Были получены высокие показатели эффективности использования гетерогенных компьютерных систем на примере использования AMD CPU и Nvidia GPU, как с точки зрения производительности, так и с экономической и энергетической точек зрения.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Data World Development Indicators. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://datafinder.worldbank.org/about-world-development-indicators?cid=GDP\_WDI.

[2] Michael, C. Fast Exact String Matching on the GPU/ Michael C., Schatz C. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.cbcb.umd.edu/software/cmatch/Cmatch.pdf

[3] Onsjo, M. Online Approximate String Matching with CUDA. . [Электронный ресурс]. – Электронные данные. – Режим доступа: http://odinlake.net/wordpress/wp-content/uploads/2009/03/pattmatch-report.pdf

[4] Hasan, L. Hardware Acceleration of Sequence Alignment Algorithms/ L. Hasan, Z. Al-Ars, S. Vassiliadis // 2007. International Conference on In Design & Technology of Integrated Systems in Nanoscale Era.

[5] Seamans, E. Fast Virus Signature Matching on the GPU/ A. Thomas, E. Seamans [Электронный ресурс]. – Электронные данные. – Режим доступа: http.developer.nvidia.com/GPUGems3/gpugems3\_ch35.html

[6] Regular Expression Matching on Graphics Hardware for Intrusion Detection / G. Vasiliadis [и др.] // 2009. Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection .

[7] Smith, R. XFA: Faster signature matching with extended automata. / R. Smith, C. Estan, S. Jha// Oakland 2008. IEEE Symposium on Security and Privacy.

[8] Evaluating GPUs for Network Packet Signature Matching. / R. Smith [и др.] // 2009. International Symposium on Performance Analysis of Systems and Software.

[9] Aho, A. Efficient String Matching: An Aid to Bibliographic Search. / A. Aho, M. Corasick. //Communications of the ACM. June, Volume 18 1975 г., стр. 333-340.

[10] Scarpazza, D. High-performance regular expression scanning on the Cell/B.E. processor./ D. Scarpazza, G. Russell.// 2009. International Conference on Supercomputing archive.

[11] Cox, A. Regular Expression Matching Can Be Simple And Fast. swtchboard. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://swtch.com/~rsc/regexp/regexp1.html.

[12] Thompson, K. Programming Techniques: Regular expression search algorithm/ K. Thompson // Communications of the ACM. June 1968, 6, стр. 419-422.

[13] IDC-digital-universe. EMC. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.emc.com/collateral/demos/microsites/idc-digital-universe/iview.htm.

[14] Amarasinghe, A. / Multicore Programming Primer. // A. Amarasinghe, Saman : MIT OpenCourseWare , 2007 г.

[15] NVIDIA. CUDA ProgrammingGuide. 2010 . [Электронный ресурс]. – Электронные данные. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/1\_1/NVIDIA\_CUDA\_Programming\_Guide\_1.1.pdf

[16] Parallel Nsight. NVIDIA. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://developer.nvidia.com/object/nexus.html.

[17] CUDA. NVIDIA . [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.nvidia.com/object/cuda\_home.html.

[18] OpenCL. Khronos Group. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.khronos.org/opencl/.

[19] NVIDIA OpenCL JumpStarе Guide. NVIDIA [Электронный ресурс]. – Электронные данные. – Режим доступа: http://developer.download.nvidia.com/OpenCL/NVIDIA\_OpenCL\_JumpStart\_Guide.pdf.

[20] Graham, R. Bound on multiprocessing timing anomalies./R. Graham// SIAM Journal on Applied Mathematics. March 1969 г., стр. 416-429.

[21] Brent, R. The parallel evaluation of general arithmetic expressions./ R. Brent // Journal of the ACM. April 1974 г., 21, стр. 201-206.

[22] . Gustafson, J. Fixed Time, Tiered Memory, and Superlinear Speedup/ J. Gustafson // Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5), 1990 г.

[23] Amdahl, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities./G. Amdahl //1967. AFIPS spring joing computer conference.

[24] AMD Processors Desktops. AMD. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://products.amd.com/en-us/DesktopCPUDetail.aspx?id=608&f1=AMD+Athlon%E2%84%A2+II+X2&f2=215&f3=2700&f4=512&f5=AM3&f6=C2&f7=45nm+SOI&f8=65+W&f9=4000&f10=False&f11=False&f12=True.

[25] Power Supply Calculator. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://extreme.outervision.com/PSUEngine.

[26] NVIDIA Power Trip. NVIDIA. [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.nvidia.com/object/product\_geforce\_gt\_220\_us.html.

[27] Палицын, В. А. Технико-экономическое обоснование дипломных проектов: Метод. Пособие для студ. всех специальностей БГУИР. В 2-ч частях. Ч. 4: Проекты программного обеспечения . Минск : БГУИР, 2006, стр. 76-90.

# ПРИЛОЖЕНИЕ

**(обязательное)**

**Текст программы**

class State

{

public:

Transition Transitions[256];

void AddManyTransitions(char\* symbols, int nextStateId, int output);

State(void);

~State(void);

};

struct Transition

{

public:

int NextState;

int Output;

};

#define GetState(table, id) (table + id \* STATE\_SIZE)

#define GetTransaction(table, id, s) ( GetState(table, id)[s])

class TransitionsTable

{

public:

Transition\* Table;

int Size;

TransitionsTable(size\_t count)

{

Size = count \* STATE\_SIZE;

Table = new Transition[Size];

memset(Table, 0, Size \* sizeof(Transition));

}

~TransitionsTable()

{

if (Table != NULL)

{

delete Table;

}

}

};

#define TerminationSymbols " \n~!@#$%^&\*()\_+=-`{}[];':"",./<>?"

class WordFinder

{

private:

std::vector<State\*> \_states;

public:

void AddWord(std::string word, int id);

void AddWords(std::vector<std::string> words);

TransitionsTable\* Generate();

WordFinder(void);

~WordFinder(void);

};

State::State(void)

{

memset(Transitions, 0, STATE\_SIZE);

}

State::~State(void)

{

}

void State::AddManyTransitions( char\* symbols, int nextStateId, int output )

{

for (int i = 0; symbols[i] != 0; i++)

{

Transitions[symbols[i]].NextState = nextStateId;

Transitions[symbols[i]].Output = output;

}

}

WordFinder::WordFinder(void)

{

State\* state = new State();

\_states.push\_back(state);

}

WordFinder::~WordFinder(void)

{

}

void WordFinder::AddWord( std::string word, int id )

{

int stateId;

State\* state;

int currentStateId = 0;

State\* currentState;

for (int i = 0; word[i] != 0; i++)

{

currentState = \_states[currentStateId];

stateId = currentState->Transitions[word[i]].NextState;

if (stateId == 0)

{

state = new State();

\_states.push\_back(state);

stateId = \_states.size() - 1;

currentState->Transitions[word[i]].NextState = stateId;

}

currentStateId = stateId;

}

currentState->AddManyTransitions(TerminationSymbols, 0, id);

}

TransitionsTable\* WordFinder::Generate()

{

size\_t size = \_states.size();

TransitionsTable\* result = new TransitionsTable(size);

for (size\_t i = 0; i < size; ++i)

{

Transition \* transaction = GetState(result->Table,i);

memcpy( transaction, \_states[i]->Transitions, STATE\_SIZE );

}

return result;

}

void WordFinder::AddWords( std::vector<std::string> words )

{

for (int i = 0; i < words.size(); i++)

{

AddWord(words[i], i+1);

}

}

class Buffer

{

void\* \_hostMemory;

void\* \_deviceMemory;

size\_t \_size;

public:

Buffer(size\_t size);

Buffer(size\_t\* size);

Buffer(size\_t\* pcount, size\_t elemSize);

~Buffer(void);

size\_t GetSize();

void\* GetHost();

void\* GetDevice();

};

Buffer::Buffer(size\_t size)

{

\_hostMemory = NULL;

\_deviceMemory = NULL;

\_size = size;

}

Buffer::Buffer(size\_t\* psize)

{

\_hostMemory = NULL;

\_deviceMemory = NULL;

cudaMemcpy(&\_size, psize, sizeof(size\_t), cudaMemcpyDeviceToHost);

}

Buffer::Buffer(size\_t\* pcount, size\_t elemSize)

{

\_hostMemory = NULL;

\_deviceMemory = NULL;

cudaMemcpy(&\_size, pcount, sizeof(size\_t), cudaMemcpyDeviceToHost);

\_size \*= elemSize;

}

Buffer::~Buffer(void)

{

if (\_hostMemory != NULL)

{

delete[](\_hostMemory);

}

if (\_deviceMemory != NULL)

{

cudaFree(\_deviceMemory);

}

}

void\* Buffer::GetHost()

{

if (\_hostMemory == NULL)

{

\_hostMemory = new char[\_size];

}

if (\_deviceMemory != NULL)

{

cudaMemcpy(\_hostMemory, \_deviceMemory, \_size, cudaMemcpyDeviceToHost);

}

return \_hostMemory;

}

void\* Buffer::GetDevice()

{

if (\_deviceMemory == NULL)

{

cudaMalloc(&\_deviceMemory, \_size);

}

if (\_hostMemory != NULL)

{

cudaMemcpy(\_deviceMemory,\_hostMemory, \_size, cudaMemcpyHostToDevice);

}

return \_deviceMemory;

}

inline size\_t Buffer::GetSize()

{

return \_size;

}

class FileStruct

{

private:

char \* \_buffer;

char \* \_deviceBuffer;

size\_t \_size;

size\_t LoadFile(char \* path, char \* & buffer);

public:

char \* Name;

size\_t GetSize();

char \* GetHostBuffer();

char \* GetDeviceBuffer();

FileStruct(char \* fileName);

~FileStruct(void);

};

FileStruct::FileStruct(char\* fileName)

{

\_deviceBuffer = NULL;

\_buffer = NULL;

\_size = 0;

Name = fileName;

}

FileStruct::~FileStruct(void)

{

if (\_buffer != NULL)

{

free(\_buffer);

}

if (\_deviceBuffer != NULL)

{

cudaFree(\_deviceBuffer);

}

}

size\_t FileStruct::LoadFile(char\* path, char\* &buffer)

{

FILE \*file;

size\_t fileLen;

//Open file

file = fopen(path, "rb");

if (!file)

{

fprintf(stderr, "Unable to open file %s", path);

return 0;

}

//Get file length

fseek(file, 0, SEEK\_END);

fileLen = ftell(file);

fseek(file, 0, SEEK\_SET);

//Allocate memory

buffer=(char \*)malloc(fileLen+1);

if (!buffer)

{

fprintf(stderr, "Memory error!");

fclose(file);

return 0;

}

//Read file contents into buffer

fread(buffer, fileLen, 1, file);

buffer[fileLen] = 0;

fclose(file);

return fileLen;

}

char \* FileStruct::GetHostBuffer()

{

if (\_buffer == NULL)

{

\_size = this->LoadFile(Name, \_buffer);

}

return \_buffer;

}

size\_t FileStruct::GetSize()

{

if (\_buffer == NULL)

{

\_size = this->LoadFile(Name, \_buffer);

}

return \_size;

}

char \* FileStruct::GetDeviceBuffer()

{

if (\_deviceBuffer == NULL)

{

char\* buf = this->GetHostBuffer();

size\_t size = this->GetSize();

cudaMalloc((void\*\*) &\_deviceBuffer, size);

cudaMemcpy(\_deviceBuffer, buf, size, cudaMemcpyHostToDevice);

}

return \_deviceBuffer;

}

void deviceFindAllWordsPrepare(TransitionsTable \* transTable);

void deviceFindAllWords(char\* text, int len, Word\* words, int\* count, int \* allWords, int\* allCount);

#include <cuda\_runtime\_api.h>

#include <iostream>

#include "FileStruct.h"

#include "Word.h"

#include "..\WordFinder\WordFinderLib.h"

#include "..\cudpp\include\cudpp.h"

#include "Buffer.h"

#include "deviceWordsFinder.h"

\_\_global\_\_ void

device\_MarkAllWords(char\* text, int len, int\* terminatedSymbols)

{

// Block index

int bx = blockIdx.x;

int by = blockIdx.y;

// Thread index

int tx = threadIdx.x;

int ty = threadIdx.y;

extern \_\_shared\_\_ int sData[];

int idx = threadIdx.x + blockDim.x \* blockIdx.x;

int r;

char c = 0;

if (idx < len-1)

{

c = text[idx];

}

r = ((

(c == ' ')||

(c == '.')||

(c == ',')||

(c == '!')||

(c == '?'))

|| (idx==0));

sData[tx] = r;

\_\_syncthreads();

if (idx < len-1)

{

if (tx!=0)

{

int r0 = sData[tx-1];

r = (r0) && (!r);

}

else

if (idx!=0)

{

c = text[idx-1];

int rprev = (

(c == ' ')||

(c == '.')||

(c == ',')||

(c == '!')||

(c == '?'));

r = (rprev) && (!r);

}

terminatedSymbols[idx] = r;

}

}

\_\_device\_\_ Transition\* table;

\_\_global\_\_ void

device\_FindAllWords( Transition\* table, char\* text, int len, int\* position, size\_t\* count, int\* words)

{

// Block index

int bx = blockIdx.x;

int by = blockIdx.y;

// Thread index

int tx = threadIdx.x;

int ty = threadIdx.y;

int idx = threadIdx.x + blockDim.x \* blockIdx.x;

if (idx < \*count)

{

int state = 0;

int pos = position[idx];

int output;

//pos++;

Transition trans;

do

{

trans = GetTransaction(table, state, text[pos]);

pos++;

state = trans.NextState;

}

while((state != 0) && (pos < len));

words[idx] = trans.Output;

}

}

\_\_global\_\_ void

device\_NormalizeAllWords( unsigned int\* words, size\_t count)

{

// Block index

int bx = blockIdx.x;

int by = blockIdx.y;

// Thread index

int tx = threadIdx.x;

int ty = threadIdx.y;

int idx = threadIdx.x + blockDim.x \* blockIdx.x;

if (idx < count)

{

unsigned int r = (words[idx])?1:0;

words[idx] = r;

}

}

void deviceFindAllWordsPrepare(TransitionsTable \* transTable)

{

cudaMalloc((void\*\*)&table, transTable->FullSize);

cudaMemcpy(table, transTable->Table, transTable->FullSize, cudaMemcpyHostToDevice);

}

void deviceFindAllWords( char\* text, int len, Word\* words, int\* count, int \* allWords, int\* allCount)

{

// setup execution parameters

int threadsNum = 512;

dim3 threads(threadsNum, 1);

dim3 grid((len-1)/threadsNum+1,1);

int\* terminatedSymbols;

int num\_elements = len;

int mem\_size = sizeof( int) \* num\_elements;

int sharedMemSize = threadsNum \* sizeof(int);

cudaMalloc(&terminatedSymbols, mem\_size);

device\_MarkAllWords<<< grid, threads, sharedMemSize >>>(text, len, terminatedSymbols);

// allocate device memory output arrays

int\* d\_odata = NULL;

cudaMalloc( (void\*\*) &d\_odata, mem\_size);

CUDPPConfiguration config;

config.datatype = CUDPP\_INT;

config.algorithm = CUDPP\_COMPACT;

config.options = CUDPP\_OPTION\_FORWARD | CUDPP\_OPTION\_INCLUSIVE |CUDPP\_OPTION\_INDEX;

CUDPPHandle scanplan = 0;

CUDPPResult result = cudppPlan(&scanplan, config, len, 1, 0);

//Buffer wordsCountBuf(sizeof(size\_t));

size\_t\* pwordsCount;

cudaMalloc( (void\*\*) &pwordsCount, sizeof(size\_t));

cudppCompact(scanplan, d\_odata, pwordsCount, text,(unsigned int\*) terminatedSymbols, len);

//printf("Words count: %d \n", \*(int\*)(wordsCountBuf.GetHost()) );

//device\_WatchDebug<<< 1, 1 >>>((char\*)wordsCountBuf.GetDevice());

Buffer wordsId(pwordsCount, sizeof(int));

Buffer valid(pwordsCount, sizeof(int));

Buffer keyWordsId(pwordsCount, sizeof(int));

cudppDestroyPlan(scanplan);

device\_FindAllWords<<< grid, threads, sharedMemSize >>>(table, text, len, d\_odata, pwordsCount, (int\*) wordsId.GetDevice() );

scanplan = 0;

config.datatype = CUDPP\_INT;

config.algorithm = CUDPP\_COMPACT;

config.options = CUDPP\_OPTION\_FORWARD | CUDPP\_OPTION\_INCLUSIVE ;

int countWords;

cudaMemcpy( &countWords, pwordsCount, sizeof(int), cudaMemcpyDeviceToHost);

result = cudppPlan(&scanplan, config, (size\_t)(countWords), 1, 0);

unsigned int\* w = (unsigned int\*) wordsId.GetDevice();

cudaMemcpy( valid.GetDevice(), wordsId.GetDevice(), countWords\*sizeof(int), cudaMemcpyDeviceToDevice);

device\_NormalizeAllWords<<< grid, threads >>>((unsigned int \*)valid.GetDevice(), countWords);

cudppCompact(scanplan,( void\*) keyWordsId.GetDevice(), pwordsCount, wordsId.GetDevice(), (unsigned int \*)valid.GetDevice(), (size\_t)(countWords));

cudppDestroyPlan(scanplan);

cudaFree(d\_odata);

cudaMemcpy(count, pwordsCount, sizeof(int), cudaMemcpyDeviceToDevice);

//cudaFree(pwordsCount);

}

template <class T, unsigned int blockSize, bool nIsPow2>

\_\_global\_\_ void

FUNC(reduce6)(T \*g\_idata, T \*g\_odata, unsigned int n)

{

SharedMemory<T> smem;

T \*sdata = smem.getPointer();

// perform first level of reduction,

// reading from global memory, writing to shared memory

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockSize\*2) + threadIdx.x;

unsigned int gridSize = blockSize\*2\*gridDim.x;

sdata[tid] = 0;

// we reduce multiple elements per thread. The number is determined by the

// number of active thread blocks (via gridDim). More blocks will result

// in a larger gridSize and therefore fewer elements per thread

while (i < n)

{

sdata[tid] += g\_idata[i];

// ensure we don't read out of bounds -- this is optimized away for powerOf2 sized arrays

if (nIsPow2 || i + blockSize < n)

sdata[tid] += g\_idata[i+blockSize];

i += gridSize;

}

\_\_syncthreads();

// do reduction in shared mem

if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } \_\_syncthreads(); }

if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } \_\_syncthreads(); }

if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } \_\_syncthreads(); }

#ifndef \_\_DEVICE\_EMULATION\_\_

if (tid < 32)

#endif

{

if (blockSize >= 64) { sdata[tid] += sdata[tid + 32]; EMUSYNC; }

if (blockSize >= 32) { sdata[tid] += sdata[tid + 16]; EMUSYNC; }

if (blockSize >= 16) { sdata[tid] += sdata[tid + 8]; EMUSYNC; }

if (blockSize >= 8) { sdata[tid] += sdata[tid + 4]; EMUSYNC; }

if (blockSize >= 4) { sdata[tid] += sdata[tid + 2]; EMUSYNC; }

if (blockSize >= 2) { sdata[tid] += sdata[tid + 1]; EMUSYNC; }

}

// write result for this block to global mem

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

#define STATE\_SIZE (sizeof(Transition) \* 256)

class State

{

public:

Transition Transitions[256];

void AddManyTransitions(char\* symbols, int nextStateId, int output);

State(void);

~State(void);

};

State::State(void)

{

memset(Transitions, 0, STATE\_SIZE);

}

State::~State(void)

{

}

void State::AddManyTransitions( char\* symbols, int nextStateId, int output )

{

for (int i = 0; symbols[i] != 0; i++)

{

Transitions[symbols[i]].NextState = nextStateId;

Transitions[symbols[i]].Output = output;

}

}

#define GetState(table, id) ((table) + (id) \* STATE\_SIZE)

#define GetTransaction(table, id, s) ( GetState((table), (id))[(s)])

int host\_FindAllWords(Transition\* table, char\* text, Word\* words )

{

int wordsCount = 0;

int state = 0;

Transition trans;

for (int i = 0; text[i] != 0; ++i)

{

trans = GetTransaction(table, state, text[i]);

if (trans.Output != 0)

{

Word word;

word.Id = trans.Output;

word.Pos = i;

words[wordsCount++] = word;

}

state = trans.NextState;

}

return wordsCount;

}

inline bool

isPowerOfTwo(int n)

{

return ((n&(n-1))==0) ;

}

inline int

floorPow2(int n)

{

#ifdef WIN32

// method 2

return 1 << (int)logb((float)n);

#else

// method 1

// float nf = (float)n;

// return 1 << (((\*(int\*)&nf) >> 23) - 127);

int exp;

frexp((float)n, &exp);

return 1 << (exp - 1);

#endif

}

#define BLOCK\_SIZE 256

float\*\* g\_scanBlockSums;

unsigned int g\_numEltsAllocated = 0;

unsigned int g\_numLevelsAllocated = 0;

void preallocBlockSums(unsigned int maxNumElements)

{

assert(g\_numEltsAllocated == 0); // shouldn't be called

g\_numEltsAllocated = maxNumElements;

unsigned int blockSize = BLOCK\_SIZE; // max size of the thread blocks

unsigned int numElts = maxNumElements;

int level = 0;

do

{

unsigned int numBlocks =

max(1, (int)ceil((float)numElts / (2.f \* blockSize)));

if (numBlocks > 1)

{

level++;

}

numElts = numBlocks;

} while (numElts > 1);

g\_scanBlockSums = (float\*\*) malloc(level \* sizeof(float\*));

g\_numLevelsAllocated = level;

numElts = maxNumElements;

level = 0;

do

{

unsigned int numBlocks =

max(1, (int)ceil((float)numElts / (2.f \* blockSize)));

if (numBlocks > 1)

{

CUDA\_SAFE\_CALL(cudaMalloc((void\*\*) &g\_scanBlockSums[level++],

numBlocks \* sizeof(float)));

}

numElts = numBlocks;

} while (numElts > 1);

CUT\_CHECK\_ERROR("preallocBlockSums");

}

void deallocBlockSums()

{

for (int i = 0; i < g\_numLevelsAllocated; i++)

{

cudaFree(g\_scanBlockSums[i]);

}

CUT\_CHECK\_ERROR("deallocBlockSums");

free((void\*\*)g\_scanBlockSums);

g\_scanBlockSums = 0;

g\_numEltsAllocated = 0;

g\_numLevelsAllocated = 0;

}

void prescanArrayRecursive(float \*outArray,

const float \*inArray,

int numElements,

int level)

{

unsigned int blockSize = BLOCK\_SIZE; // max size of the thread blocks

unsigned int numBlocks =

max(1, (int)ceil((float)numElements / (2.f \* blockSize)));

unsigned int numThreads;

if (numBlocks > 1)

numThreads = blockSize;

else if (isPowerOfTwo(numElements))

numThreads = numElements / 2;

else

numThreads = floorPow2(numElements);

unsigned int numEltsPerBlock = numThreads \* 2;

// if this is a non-power-of-2 array, the last block will be non-full

// compute the smallest power of 2 able to compute its scan.

unsigned int numEltsLastBlock =

numElements - (numBlocks-1) \* numEltsPerBlock;

unsigned int numThreadsLastBlock = max(1, numEltsLastBlock / 2);

unsigned int np2LastBlock = 0;

unsigned int sharedMemLastBlock = 0;

if (numEltsLastBlock != numEltsPerBlock)

{

np2LastBlock = 1;

if(!isPowerOfTwo(numEltsLastBlock))

numThreadsLastBlock = floorPow2(numEltsLastBlock);

unsigned int extraSpace = (2 \* numThreadsLastBlock) / NUM\_BANKS;

sharedMemLastBlock =

sizeof(float) \* (2 \* numThreadsLastBlock + extraSpace);

}

// padding space is used to avoid shared memory bank conflicts

unsigned int extraSpace = numEltsPerBlock / NUM\_BANKS;

unsigned int sharedMemSize =

sizeof(float) \* (numEltsPerBlock + extraSpace);

#ifdef DEBUG

if (numBlocks > 1)

{

assert(g\_numEltsAllocated >= numElements);

}

#endif

// setup execution parameters

// if NP2, we process the last block separately

dim3 grid(max(1, numBlocks - np2LastBlock), 1, 1);

dim3 threads(numThreads, 1, 1);

// make sure there are no CUDA errors before we start

CUT\_CHECK\_ERROR("prescanArrayRecursive before kernels");

// execute the scan

if (numBlocks > 1)

{

prescan<true, false><<< grid, threads, sharedMemSize >>>(outArray,

inArray,

g\_scanBlockSums[level],

numThreads \* 2, 0, 0);

CUT\_CHECK\_ERROR("prescanWithBlockSums");

if (np2LastBlock)

{

prescan<true, true><<< 1, numThreadsLastBlock, sharedMemLastBlock >>>

(outArray, inArray, g\_scanBlockSums[level], numEltsLastBlock,

numBlocks - 1, numElements - numEltsLastBlock);

CUT\_CHECK\_ERROR("prescanNP2WithBlockSums");

}

// After scanning all the sub-blocks, we are mostly done. But now we

// need to take all of the last values of the sub-blocks and scan those.

// This will give us a new value that must be sdded to each block to

// get the final results.

// recursive (CPU) call

prescanArrayRecursive(g\_scanBlockSums[level],

g\_scanBlockSums[level],

numBlocks,

level+1);

uniformAdd<<< grid, threads >>>(outArray,

g\_scanBlockSums[level],

numElements - numEltsLastBlock,

0, 0);

CUT\_CHECK\_ERROR("uniformAdd");

if (np2LastBlock)

{

uniformAdd<<< 1, numThreadsLastBlock >>>(outArray,

g\_scanBlockSums[level],

numEltsLastBlock,

numBlocks - 1,

numElements - numEltsLastBlock);

CUT\_CHECK\_ERROR("uniformAdd");

}

}

else if (isPowerOfTwo(numElements))

{

prescan<false, false><<< grid, threads, sharedMemSize >>>(outArray, inArray,

0, numThreads \* 2, 0, 0);

CUT\_CHECK\_ERROR("prescan");

}

else

{

prescan<false, true><<< grid, threads, sharedMemSize >>>(outArray, inArray,

0, numElements, 0, 0);

CUT\_CHECK\_ERROR("prescanNP2");

}

}

void prescanArray(float \*outArray, float \*inArray, int numElements)

{

prescanArrayRecursive(outArray, inArray, numElements, 0);

}

#define NUM\_BANKS 16

#define LOG\_NUM\_BANKS 4

#ifdef ZERO\_BANK\_CONFLICTS

#define CONFLICT\_FREE\_OFFSET(index) ((index) >> LOG\_NUM\_BANKS + (index) >> (2\*LOG\_NUM\_BANKS))

#else

#define CONFLICT\_FREE\_OFFSET(index) ((index) >> LOG\_NUM\_BANKS)

#endif

template <bool isNP2>

\_\_device\_\_ void loadSharedChunkFromMem(float \*s\_data,

const float \*g\_idata,

int n, int baseIndex,

int& ai, int& bi,

int& mem\_ai, int& mem\_bi,

int& bankOffsetA, int& bankOffsetB)

{

int thid = threadIdx.x;

mem\_ai = baseIndex + threadIdx.x;

mem\_bi = mem\_ai + blockDim.x;

ai = thid;

bi = thid + blockDim.x;

// compute spacing to avoid bank conflicts

bankOffsetA = CONFLICT\_FREE\_OFFSET(ai);

bankOffsetB = CONFLICT\_FREE\_OFFSET(bi);

// Cache the computational window in shared memory

// pad values beyond n with zeros

s\_data[ai + bankOffsetA] = g\_idata[mem\_ai];

if (isNP2) // compile-time decision

{

s\_data[bi + bankOffsetB] = (bi < n) ? g\_idata[mem\_bi] : 0;

}

else

{

s\_data[bi + bankOffsetB] = g\_idata[mem\_bi];

}

}

template <bool isNP2>

\_\_device\_\_ void storeSharedChunkToMem(float\* g\_odata,

const float\* s\_data,

int n,

int ai, int bi,

int mem\_ai, int mem\_bi,

int bankOffsetA, int bankOffsetB)

{

\_\_syncthreads();

// write results to global memory

g\_odata[mem\_ai] = s\_data[ai + bankOffsetA];

if (isNP2) // compile-time decision

{

if (bi < n)

g\_odata[mem\_bi] = s\_data[bi + bankOffsetB];

}

else

{

g\_odata[mem\_bi] = s\_data[bi + bankOffsetB];

}

}

template <bool storeSum>

\_\_device\_\_ void clearLastElement(float\* s\_data,

float \*g\_blockSums,

int blockIndex)

{

if (threadIdx.x == 0)

{

int index = (blockDim.x << 1) - 1;

index += CONFLICT\_FREE\_OFFSET(index);

if (storeSum) // compile-time decision

{

// write this block's total sum to the corresponding index in the blockSums array

g\_blockSums[blockIndex] = s\_data[index];

}

// zero the last element in the scan so it will propagate back to the front

s\_data[index] = 0;

}

}

\_\_device\_\_ unsigned int buildSum(float \*s\_data)

{

unsigned int thid = threadIdx.x;

unsigned int stride = 1;

// build the sum in place up the tree

for (int d = blockDim.x; d > 0; d >>= 1)

{

\_\_syncthreads();

if (thid < d)

{

int i = \_\_mul24(\_\_mul24(2, stride), thid);

int ai = i + stride - 1;

int bi = ai + stride;

ai += CONFLICT\_FREE\_OFFSET(ai);

bi += CONFLICT\_FREE\_OFFSET(bi);

s\_data[bi] += s\_data[ai];

}

stride \*= 2;

}

return stride;

}

\_\_device\_\_ void scanRootToLeaves(float \*s\_data, unsigned int stride)

{

unsigned int thid = threadIdx.x;

// traverse down the tree building the scan in place

for (int d = 1; d <= blockDim.x; d \*= 2)

{

stride >>= 1;

\_\_syncthreads();

if (thid < d)

{

int i = \_\_mul24(\_\_mul24(2, stride), thid);

int ai = i + stride - 1;

int bi = ai + stride;

ai += CONFLICT\_FREE\_OFFSET(ai);

bi += CONFLICT\_FREE\_OFFSET(bi);

float t = s\_data[ai];

s\_data[ai] = s\_data[bi];

s\_data[bi] += t;

}

}

}

template <bool storeSum>

\_\_device\_\_ void prescanBlock(float \*data, int blockIndex, float \*blockSums)

{

int stride = buildSum(data); // build the sum in place up the tree

clearLastElement<storeSum>(data, blockSums,

(blockIndex == 0) ? blockIdx.x : blockIndex);

scanRootToLeaves(data, stride); // traverse down tree to build the scan

}

template <bool storeSum, bool isNP2>

\_\_global\_\_ void prescan(float \*g\_odata,

const float \*g\_idata,

float \*g\_blockSums,

int n,

int blockIndex,

int baseIndex)

{

int ai, bi, mem\_ai, mem\_bi, bankOffsetA, bankOffsetB;

extern \_\_shared\_\_ float s\_data[];

// load data into shared memory

loadSharedChunkFromMem<isNP2>(s\_data, g\_idata, n,

(baseIndex == 0) ?

\_\_mul24(blockIdx.x, (blockDim.x << 1)):baseIndex,

ai, bi, mem\_ai, mem\_bi,

bankOffsetA, bankOffsetB);

// scan the data in each block

prescanBlock<storeSum>(s\_data, blockIndex, g\_blockSums);

// write results to device memory

storeSharedChunkToMem<isNP2>(g\_odata, s\_data, n,

ai, bi, mem\_ai, mem\_bi,

bankOffsetA, bankOffsetB);

}

\_\_global\_\_ void uniformAdd(float \*g\_data,

float \*uniforms,

int n,

int blockOffset,

int baseIndex)

{

\_\_shared\_\_ float uni;

if (threadIdx.x == 0)

uni = uniforms[blockIdx.x + blockOffset];

unsigned int address = \_\_mul24(blockIdx.x, (blockDim.x << 1)) + baseIndex + threadIdx.x;

\_\_syncthreads();

// note two adds per thread

g\_data[address] += uni;

g\_data[address + blockDim.x] += (threadIdx.x + blockDim.x < n) \* uni;

}

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

flipFloats(uint \*values, uint numValues)

{

uint index = \_\_umul24(blockDim.x\*4, blockIdx.x) + threadIdx.x;

if (index < numValues) values[index] = floatFlip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatFlip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatFlip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatFlip<true>(values[index]);

}

/\*\* @brief Undoes the flips from flipFloats

\*

\* Uses floatUnflip function to unflip bits.

\* @param[in,out] values Values to be manipulated

\* @param[in] numValues Number of values to be unflipped

\*\*/

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

unflipFloats(uint \*values, uint numValues)

{

uint index = \_\_umul24(blockDim.x\*4, blockIdx.x) + threadIdx.x;

if (index < numValues) values[index] = floatUnflip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatUnflip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatUnflip<true>(values[index]);

index += blockDim.x;

if (index < numValues) values[index] = floatUnflip<true>(values[index]);

}

/\*\* @brief Optimization for sorts of WARP\_SIZE or fewer elements

\*

\* @param[in,out] keys Keys to be sorted.

\* @param[in,out] values Associated values to be sorted (through keys).

\* @param[in] numElements Number of elements in the sort.

\*/

template <bool flip>

\_\_global\_\_

LAUNCH\_BOUNDS(WARP\_SIZE)

void radixSortSingleWarp(uint \*keys,

uint \*values,

uint numElements)

{

volatile \_\_shared\_\_ uint sKeys[WARP\_SIZE]; //remove class distinctions

volatile \_\_shared\_\_ uint sValues[WARP\_SIZE];

volatile \_\_shared\_\_ uint sFlags[WARP\_SIZE];

sKeys[threadIdx.x] = floatFlip<flip>(keys[threadIdx.x]);

sValues[threadIdx.x] = values[threadIdx.x];

\_\_EMUSYNC; // emulation only

for(uint i = 1; i < numElements; i++)

{

uint key\_i = sKeys[i];

uint val\_i = sValues[i];

sFlags[threadIdx.x] = 0;

uint temp, tempval;

if( (threadIdx.x < i) && (sKeys[threadIdx.x] > key\_i) )

{

temp = sKeys[threadIdx.x];

tempval = sValues[threadIdx.x];

sFlags[threadIdx.x] = 1;

#ifdef \_\_DEVICE\_EMULATION\_\_

}

\_\_EMUSYNC;

if( (threadIdx.x < i) && (sKeys[threadIdx.x] > key\_i) )

{

#endif

sKeys[threadIdx.x + 1] = temp;

sValues[threadIdx.x + 1] = tempval;

sFlags[threadIdx.x + 1] = 0;

}

if(sFlags[threadIdx.x] == 1 )

{

sKeys[threadIdx.x] = key\_i;

sValues[threadIdx.x] = val\_i;

}

\_\_EMUSYNC; // emulation only

}

keys[threadIdx.x] = floatUnflip<flip>(sKeys[threadIdx.x]);

values[threadIdx.x] = sValues[threadIdx.x];

}

/\*\*

\* @param[in,out] keys Keys to be sorted

\* @param[in] numElements Total number of elements to be sorted

\*\*/

template <bool flip>

\_\_global\_\_

LAUNCH\_BOUNDS(WARP\_SIZE)

void radixSortSingleWarpKeysOnly(uint \*keys,

uint numElements)

{

volatile \_\_shared\_\_ uint sKeys[WARP\_SIZE];

volatile \_\_shared\_\_ uint sFlags[WARP\_SIZE];

sKeys[threadIdx.x] = floatFlip<flip>(keys[threadIdx.x]);

\_\_EMUSYNC; // emulation only

for(uint i = 1; i < numElements; i++)

{

uint key\_i = sKeys[i];

sFlags[threadIdx.x] = 0;

uint temp;

if( (threadIdx.x < i) && (sKeys[threadIdx.x] > key\_i) )

{

temp = sKeys[threadIdx.x];

sFlags[threadIdx.x] = 1;

#ifdef \_\_DEVICE\_EMULATION\_\_

}

\_\_EMUSYNC;

if( (threadIdx.x < i) && (sKeys[threadIdx.x] > key\_i) )

{

#endif

sKeys[threadIdx.x + 1] = temp;

sFlags[threadIdx.x + 1] = 0;

}

if(sFlags[threadIdx.x] == 1 )

{

sKeys[threadIdx.x] = key\_i;

}

\_\_EMUSYNC; // emulation only

}

keys[threadIdx.x] = floatUnflip<flip>(sKeys[threadIdx.x]);

}

/\*\* @brief sorts all blocks of data independently in shared memory.

\* GPUs than it is on compute version 1.2 GPUs.

\*

\* @param[out] keysOut Output of sorted keys

\* @param[out] valuesOut Output of associated values

\* @param[in] keysIn Input of unsorted keys in GPU

\* @param[in] valuesIn Input of associated input values

\* @param[in] numElements Total number of elements to sort

\* @param[in] totalBlocks The number of blocks of data to sort

\*/

template<uint nbits, uint startbit, bool fullBlocks, bool flip, bool loop>

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

radixSortBlocks(uint4\* keysOut, uint4\* valuesOut,

uint4\* keysIn, uint4\* valuesIn,

uint numElements, uint totalBlocks)

{

extern \_\_shared\_\_ uint4 sMem[];

uint4 key, value;

uint blockId = blockIdx.x;

while (!loop || blockId < totalBlocks)

{

uint i = blockId \* blockDim.x + threadIdx.x;

uint idx = i << 2;

// handle non-full last block if array is not multiple of 1024 numElements

if (!fullBlocks && idx+3 >= numElements)

{

if (idx >= numElements)

{

key = make\_uint4(UINT\_MAX, UINT\_MAX, UINT\_MAX, UINT\_MAX);

value = make\_uint4(UINT\_MAX, UINT\_MAX, UINT\_MAX, UINT\_MAX);

}

else

{

// for non-full block, we handle uint1 values instead of uint4

uint \*keys1 = (uint\*)keysIn;

uint \*values1 = (uint\*)valuesIn;

key.x = (idx < numElements) ? floatFlip<flip>(keys1[idx]) : UINT\_MAX;

key.y = (idx+1 < numElements) ? floatFlip<flip>(keys1[idx+1]) : UINT\_MAX;

key.z = (idx+2 < numElements) ? floatFlip<flip>(keys1[idx+2]) : UINT\_MAX;

key.w = UINT\_MAX;

value.x = (idx < numElements) ? values1[idx] : UINT\_MAX;

value.y = (idx+1 < numElements) ? values1[idx+1] : UINT\_MAX;

value.z = (idx+2 < numElements) ? values1[idx+2] : UINT\_MAX;

value.w = UINT\_MAX;

}

}

else

{

key = keysIn[i];

value = valuesIn[i];

if (flip)

{

key.x = floatFlip<flip>(key.x);

key.y = floatFlip<flip>(key.y);

key.z = floatFlip<flip>(key.z);

key.w = floatFlip<flip>(key.w);

}

}

\_\_syncthreads();

radixSortBlock<nbits, startbit>(key, value);

// handle non-full last block if array is not multiple of 1024 numElements

if(!fullBlocks && idx+3 >= numElements)

{

if (idx < numElements)

{

// for non-full block, we handle uint1 values instead of uint4

uint \*keys1 = (uint\*)keysOut;

uint \*values1 = (uint\*)valuesOut;

keys1[idx] = key.x;

values1[idx] = value.x;

if (idx + 1 < numElements)

{

keys1[idx + 1] = key.y;

values1[idx + 1] = value.y;

if (idx + 2 < numElements)

{

keys1[idx + 2] = key.z;

values1[idx + 2] = value.z;

}

}

}

}

else

{

keysOut[i] = key;

valuesOut[i] = value;

}

if (loop)

blockId += gridDim.x;

else

break;

}

}

/\*\* @brief Computes the number of keys of each radix in each block stores offset.

\*

\* GPUs than it is on compute version 1.2 GPUs.

\*

\* @param[in] keys Input keys

\* @param[out] counters Radix count for each block

\* @param[out] blockOffsets The offset address for each block

\* @param[in] numElements Total number of elements

\* @param[in] totalBlocks Total number of blocks

\*\*/

template<uint startbit, bool fullBlocks, bool loop>

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

findRadixOffsets(uint2 \*keys,

uint \*counters,

uint \*blockOffsets,

uint numElements,

uint totalBlocks)

{

extern \_\_shared\_\_ uint sRadix1[];

\_\_shared\_\_ uint sStartPointers[16];

uint blockId = blockIdx.x;

while (!loop || blockId < totalBlocks)

{

uint2 radix2;

uint i = blockId \* blockDim.x + threadIdx.x;

// handle non-full last block if array is not multiple of 1024 numElements

if(!fullBlocks && ((i + 1) << 1 ) > numElements )

{

// handle uint1 rather than uint2 for non-full blocks

uint \*keys1 = (uint\*)keys;

uint j = i << 1;

radix2.x = (j < numElements) ? keys1[j] : UINT\_MAX;

j++;

radix2.y = (j < numElements) ? keys1[j] : UINT\_MAX;

}

else

{

radix2 = keys[i];

}

sRadix1[2 \* threadIdx.x] = (radix2.x >> startbit) & 0xF;

sRadix1[2 \* threadIdx.x + 1] = (radix2.y >> startbit) & 0xF;

// Finds the position where the sRadix1 entries differ and stores start

// index for each radix.

if(threadIdx.x < 16)

{

sStartPointers[threadIdx.x] = 0;

}

\_\_syncthreads();

if((threadIdx.x > 0) && (sRadix1[threadIdx.x] != sRadix1[threadIdx.x - 1]) )

{

sStartPointers[sRadix1[threadIdx.x]] = threadIdx.x;

}

if(sRadix1[threadIdx.x + SORT\_CTA\_SIZE] != sRadix1[threadIdx.x + SORT\_CTA\_SIZE - 1])

{

sStartPointers[sRadix1[threadIdx.x + SORT\_CTA\_SIZE]] = threadIdx.x + SORT\_CTA\_SIZE;

}

\_\_syncthreads();

if(threadIdx.x < 16)

{

blockOffsets[blockId\*16 + threadIdx.x] = sStartPointers[threadIdx.x];

}

\_\_syncthreads();

// Compute the sizes of each block.

if((threadIdx.x > 0) && (sRadix1[threadIdx.x] != sRadix1[threadIdx.x - 1]) )

{

sStartPointers[sRadix1[threadIdx.x - 1]] =

threadIdx.x - sStartPointers[sRadix1[threadIdx.x - 1]];

}

if(sRadix1[threadIdx.x + SORT\_CTA\_SIZE] != sRadix1[threadIdx.x + SORT\_CTA\_SIZE - 1] )

{

sStartPointers[sRadix1[threadIdx.x + SORT\_CTA\_SIZE - 1]] =

threadIdx.x + SORT\_CTA\_SIZE - sStartPointers[sRadix1[threadIdx.x + SORT\_CTA\_SIZE - 1]];

}

if(threadIdx.x == SORT\_CTA\_SIZE - 1)

{

sStartPointers[sRadix1[2 \* SORT\_CTA\_SIZE - 1]] =

2 \* SORT\_CTA\_SIZE - sStartPointers[sRadix1[2 \* SORT\_CTA\_SIZE - 1]];

}

\_\_syncthreads();

if(threadIdx.x < 16)

{

counters[threadIdx.x \* totalBlocks + blockId] =

sStartPointers[threadIdx.x];

}

if (loop)

blockId += gridDim.x;

else

break;

}

}

/\*\*@brief Reorders data in the global array.

\*

threshold is higher on compute version 1.1 and earlier

\* GPUs than it is on compute version 1.2 GPUs.

\*

\* @param[out] outKeys Output of sorted keys

\* @param[out] outValues Output of associated values

\* @param[in] keys Input of unsorted keys in GPU

\* @param[in] values Input of associated input values

\* @param[in] blockOffsets The offset address for each block

\* @param[in] offsets Address of each radix within each block

\* @param[in] sizes Number of elements in a block

\* @param[in] numElements Total number of elements

\* @param[in] totalBlocks Total number of data blocks to process

\*

\* @todo Args that are const below should be prototyped as const

\*\*/

template<uint startbit, bool fullBlocks, bool manualCoalesce, bool unflip, bool loop>

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

reorderData(uint \*outKeys,

uint \*outValues,

uint2 \*keys,

uint2 \*values,

uint \*blockOffsets,

uint \*offsets,

uint \*sizes,

uint numElements,

uint totalBlocks)

{

\_\_shared\_\_ uint2 sKeys2[SORT\_CTA\_SIZE];

\_\_shared\_\_ uint2 sValues2[SORT\_CTA\_SIZE];

\_\_shared\_\_ uint sOffsets[16];

\_\_shared\_\_ uint sBlockOffsets[16];

uint \*sKeys1 = (uint\*)sKeys2;

uint \*sValues1 = (uint\*)sValues2;

uint blockId = blockIdx.x;

while (!loop || blockId < totalBlocks)

{

uint i = blockId \* blockDim.x + threadIdx.x;

// handle non-full last block if array is not multiple of 1024 numElements

if(!fullBlocks && (((i + 1) << 1) > numElements))

{

uint \*keys1 = (uint\*)keys;

uint \*values1 = (uint\*)values;

uint j = i << 1;

sKeys1[threadIdx.x << 1] = (j < numElements) ? keys1[j] : UINT\_MAX;

sValues1[threadIdx.x << 1] = (j < numElements) ? values1[j] : UINT\_MAX;

j++;

sKeys1[(threadIdx.x << 1) + 1] = (j < numElements) ? keys1[j] : UINT\_MAX;

sValues1[(threadIdx.x << 1) + 1] = (j < numElements) ? values1[j] : UINT\_MAX;

}

else

{

sKeys2[threadIdx.x] = keys[i];

sValues2[threadIdx.x] = values[i];

}

if (!manualCoalesce)

{

if(threadIdx.x < 16)

{

sOffsets[threadIdx.x] = offsets[threadIdx.x \* totalBlocks + blockId];

sBlockOffsets[threadIdx.x] = blockOffsets[blockId \* 16 + threadIdx.x];

}

\_\_syncthreads();

uint radix = (sKeys1[threadIdx.x] >> startbit) & 0xF;

uint globalOffset = sOffsets[radix] + threadIdx.x - sBlockOffsets[radix];

if (fullBlocks || globalOffset < numElements)

{

outKeys[globalOffset] = floatUnflip<unflip>(sKeys1[threadIdx.x]);

outValues[globalOffset] = sValues1[threadIdx.x];

}

radix = (sKeys1[threadIdx.x + SORT\_CTA\_SIZE] >> startbit) & 0xF;

globalOffset = sOffsets[radix] + threadIdx.x + SORT\_CTA\_SIZE - sBlockOffsets[radix];

if (fullBlocks || globalOffset < numElements)

{

outKeys[globalOffset] = floatUnflip<unflip>(sKeys1[threadIdx.x + SORT\_CTA\_SIZE]);

outValues[globalOffset] = sValues1[threadIdx.x + SORT\_CTA\_SIZE];

}

}

else

{

\_\_shared\_\_ uint sSizes[16];

if(threadIdx.x < 16)

{

sOffsets[threadIdx.x] = offsets[threadIdx.x \* totalBlocks + blockId];

sBlockOffsets[threadIdx.x] = blockOffsets[blockId \* 16 + threadIdx.x];

sSizes[threadIdx.x] = sizes[threadIdx.x \* totalBlocks + blockId];

}

\_\_syncthreads();

const uint halfWarpID = threadIdx.x >> 4;

const uint halfWarpOffset = threadIdx.x & 0xF;

const uint leadingInvalid = sOffsets[halfWarpID] & 0xF;

uint startPos = sOffsets[halfWarpID] & 0xFFFFFFF0;

uint endPos = (sOffsets[halfWarpID] + sSizes[halfWarpID]) + 15 -

((sOffsets[halfWarpID] + sSizes[halfWarpID] - 1) & 0xF);

uint numIterations = endPos - startPos;

uint outOffset = startPos + halfWarpOffset;

uint inOffset = sBlockOffsets[halfWarpID] - leadingInvalid + halfWarpOffset;

for(uint j = 0; j < numIterations; j += 16, outOffset += 16, inOffset += 16)

{

if( (outOffset >= sOffsets[halfWarpID]) &&

(inOffset - sBlockOffsets[halfWarpID] < sSizes[halfWarpID]))

{

if(blockId < totalBlocks - 1 || outOffset < numElements)

{

outKeys[outOffset] = floatUnflip<unflip>(sKeys1[inOffset]);

outValues[outOffset] = sValues1[inOffset];

}

}

}

}

if (loop)

{

blockId += gridDim.x;

\_\_syncthreads();

}

else

break;

}

}

/\*\* @brief Sorts all blocks of data independently in shared memory.

\* @param[out] keysOut Output of sorted keys GPU main memory

\* @param[in] keysIn Input of unsorted keys in GPU main memory

\* @param[in] numElements Total number of elements to sort

\* @param[in] totalBlocks Total number of blocks to sort

\*

\*/

template<uint nbits, uint startbit, bool fullBlocks, bool flip, bool loop>

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

radixSortBlocksKeysOnly(uint4\* keysOut, uint4\* keysIn, uint numElements, uint totalBlocks)

{

extern \_\_shared\_\_ uint4 sMem[];

uint4 key;

uint blockId = blockIdx.x;

while (!loop || blockId < totalBlocks)

{

uint i = blockId \* blockDim.x + threadIdx.x;

uint idx = i << 2;

// handle non-full last block if array is not multiple of 1024 numElements

if (!fullBlocks && idx+3 >= numElements)

{

if (idx >= numElements)

{

key = make\_uint4(UINT\_MAX, UINT\_MAX, UINT\_MAX, UINT\_MAX);

}

else

{

// for non-full block, we handle uint1 values instead of uint4

uint \*keys1 = (uint\*)keysIn;

key.x = (idx < numElements) ? floatFlip<flip>(keys1[idx]) : UINT\_MAX;

key.y = (idx+1 < numElements) ? floatFlip<flip>(keys1[idx+1]) : UINT\_MAX;

key.z = (idx+2 < numElements) ? floatFlip<flip>(keys1[idx+2]) : UINT\_MAX;

key.w = UINT\_MAX;

}

}

else

{

key = keysIn[i];

if (flip)

{

key.x = floatFlip<flip>(key.x);

key.y = floatFlip<flip>(key.y);

key.z = floatFlip<flip>(key.z);

key.w = floatFlip<flip>(key.w);

}

}

\_\_syncthreads();

radixSortBlockKeysOnly<nbits, startbit>(key);

// handle non-full last block if array is not multiple of 1024 numElements

if(!fullBlocks && idx+3 >= numElements)

{

if (idx < numElements)

{

// for non-full block, we handle uint1 values instead of uint4

uint \*keys1 = (uint\*)keysOut;

keys1[idx] = key.x;

if (idx + 1 < numElements)

{

keys1[idx + 1] = key.y;

if (idx + 2 < numElements)

{

keys1[idx + 2] = key.z;

}

}

}

}

else

{

keysOut[i] = key;

}

if (loop)

blockId += gridDim.x;

else

break;

}

}

\*

\* @param[out] outKeys Output result of reorderDataKeysOnly()

\* @param[in] keys Keys to be reordered

\* @param[in] blockOffsets Start offset for each block

\* @param[in] offsets Offset of each radix within each block

\* @param[in] sizes Number of elements in a block

\* @param[in] numElements Total number of elements

\* @param[in] totalBlocks Total number of blocks

\*/

template<uint startbit, bool fullBlocks, bool manualCoalesce, bool unflip, bool loop>

\_\_global\_\_ void

LAUNCH\_BOUNDS(SORT\_CTA\_SIZE)

reorderDataKeysOnly(uint \*outKeys,

uint2 \*keys,

uint \*blockOffsets,

uint \*offsets,

uint \*sizes,

uint numElements,

uint totalBlocks)

{

\_\_shared\_\_ uint2 sKeys2[SORT\_CTA\_SIZE];

\_\_shared\_\_ uint sOffsets[16];

\_\_shared\_\_ uint sBlockOffsets[16];

uint \*sKeys1 = (uint\*)sKeys2;

uint blockId = blockIdx.x;

while (!loop || blockId < totalBlocks)

{

uint i = blockId \* blockDim.x + threadIdx.x;

// handle non-full last block if array is not multiple of 1024 numElements

if(!fullBlocks && (((i + 1) << 1) > numElements))

{

uint \*keys1 = (uint\*)keys;

uint j = i << 1;

sKeys1[threadIdx.x << 1] = (j < numElements) ? keys1[j] : UINT\_MAX;

j++;

sKeys1[(threadIdx.x << 1) + 1] = (j < numElements) ? keys1[j] : UINT\_MAX;

}

else

{

sKeys2[threadIdx.x] = keys[i];

}

if (!manualCoalesce)

{

if(threadIdx.x < 16)

{

sOffsets[threadIdx.x] = offsets[threadIdx.x \* totalBlocks + blockId];

sBlockOffsets[threadIdx.x] = blockOffsets[blockId \* 16 + threadIdx.x];

}

\_\_syncthreads();

uint radix = (sKeys1[threadIdx.x] >> startbit) & 0xF;

uint globalOffset = sOffsets[radix] + threadIdx.x - sBlockOffsets[radix];

if (fullBlocks || globalOffset < numElements)

{

outKeys[globalOffset] = floatUnflip<unflip>(sKeys1[threadIdx.x]);

}

radix = (sKeys1[threadIdx.x + SORT\_CTA\_SIZE] >> startbit) & 0xF;

globalOffset = sOffsets[radix] + threadIdx.x + SORT\_CTA\_SIZE - sBlockOffsets[radix];

if (fullBlocks || globalOffset < numElements)

{

outKeys[globalOffset] = floatUnflip<unflip>(sKeys1[threadIdx.x + SORT\_CTA\_SIZE]);

}

}

else

{

\_\_shared\_\_ uint sSizes[16];

if(threadIdx.x < 16)

{

sOffsets[threadIdx.x] = offsets[threadIdx.x \* totalBlocks + blockId];

sBlockOffsets[threadIdx.x] = blockOffsets[blockId \* 16 + threadIdx.x];

sSizes[threadIdx.x] = sizes[threadIdx.x \* totalBlocks + blockId];

}

\_\_syncthreads();

const uint halfWarpID = threadIdx.x >> 4;

const uint halfWarpOffset = threadIdx.x & 0xF;

const uint leadingInvalid = sOffsets[halfWarpID] & 0xF;

uint startPos = sOffsets[halfWarpID] & 0xFFFFFFF0;

uint endPos = (sOffsets[halfWarpID] + sSizes[halfWarpID]) + 15 -

((sOffsets[halfWarpID] + sSizes[halfWarpID] - 1) & 0xF);

uint numIterations = endPos - startPos;

uint outOffset = startPos + halfWarpOffset;

uint inOffset = sBlockOffsets[halfWarpID] - leadingInvalid + halfWarpOffset;

for(uint j = 0; j < numIterations; j += 16, outOffset += 16, inOffset += 16)

{

if( (outOffset >= sOffsets[halfWarpID]) &&

(inOffset - sBlockOffsets[halfWarpID] < sSizes[halfWarpID]))

{

if(blockId < totalBlocks - 1 || outOffset < numElements)

{

outKeys[outOffset] = floatUnflip<unflip>(sKeys1[inOffset]);

}

}

}

}

if (loop)

{

blockId += gridDim.x;

\_\_syncthreads();

}

else

break;

}

}