УДК 004.057

О.В. Аверин

РЕАЛИЗАЦИЯ ТОКЕНИЗАТОРА НА ГРАФИЧЕСКОМ ПРОЦЕССОРЕ

Нижегородский государственный технический университет им. Р. Е. Алексеева

Цель: Разработка эффективного алгоритма для графического процессора токенизации текстов на естественном языке.

Методология проведения работы: Применяются различные алгоритмы распараллеливания токенизации текста для SIMT архитектуры. Для проверки эффективности проводится ряд экспериментов.

Результаты и область применения: Использование графических процессоров для токенизации текста подтверждено экспериментально, а следовательно, полученный алгоритм может быть использован для решения широкого круга задач компьютерной лингвистики.

Выводы: Предложенный алгоритм позволяет значительно сократить время выполнения процедуры токенизации текста.

Ключевые слова: CUDA, параллельные вычисления, компьютерная лингвистика

Введение

Задачи обработки текстовой информации на естественных языках, такие как рубрикация, установление авторства, поиск фактов и т.д., являются весьма важными. Этот класс задач решается с помощью различных лексических анализаторов, для которых элементарной составляющей текста является, как правило, токен или лексема. Данный этап обработки текстовой информации может занимать значительную часть времени выполнения всего алгоритма.

Применение современных графических процессоров может значительно сократить время токенизации текста на естественном языке, без применения использования дорогостоящих многопроцессорных систем. Кроме того, для разработчиков стали доступны несколько языков высокого уровня для графических процессоров, таких как CUDA [1] и OpenCL [2], а следовательно, исчезла необходимость изучать особенности программирования графических процессоров для решения неграфических задач [3].

Многопоточные алгоритмы токенизации

Наиболее простым способом сокращения времени токенизации является переложение однопоточного алгоритма для центрального процессора на архитектуру графического процессора путём распараллеливания по входным данным.

Обозначим символы, которые могут входить в состав токена, – «1», а символы, разделяющие токены, – «0».

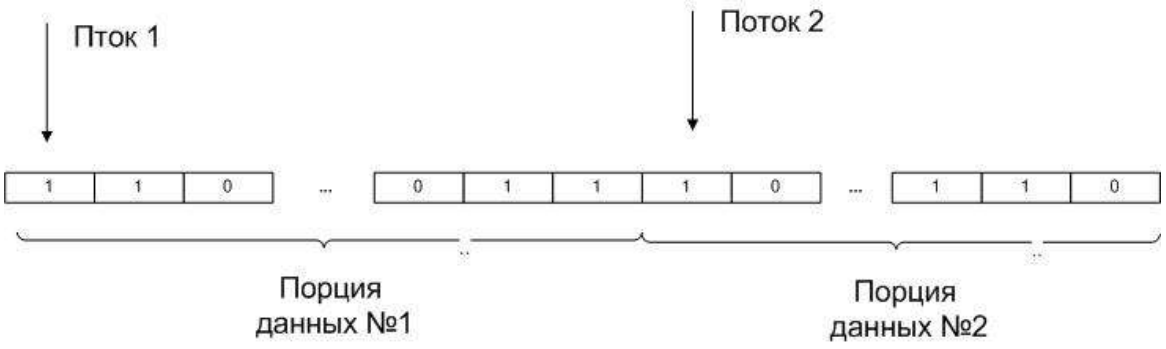

Рис. 1. Распараллеливание по входным данным

Таким образом, если видеокарта может параллельно выполнять *N* потоков, а обработать необходимо текст, состоящий из *M* символов, то каждый поток должен обработать *M/N* символов, а время вычислений может быть сокращено в *N* раз.

Однако такой алгоритм имеет ряд существенных недостатков:

1) Некоторые токены могут находиться на границе, разделяющей блоки данных, обрабатываемых разными потоками. Это повлёчет за собой постобработку результатов работы токенизатора и «склеивание» таких токенов.

2) При реализации данного алгоритма на графических процессорах компании NVIDIA значительный объём данных негативно скажется на производительности, так как объёмы «быстрых» видов памяти незначительны и размещение входных данных будет возможно только в глобальной памяти видеопроцессора, обладающей высокой латентностью. Если *M/N* > 128 байт, где *M* – количество символов во входном тексте, а *N* – количество потоков, то все обращения к глобальной памяти будут безколлизионными [1], а следовательно, время вычислений возрастет.

Первая проблема может быть решена путём перекрытия областей обработки данных потоками, если порции данных будут пересекаться *n* символами, где *n* – максимальная длина токена.

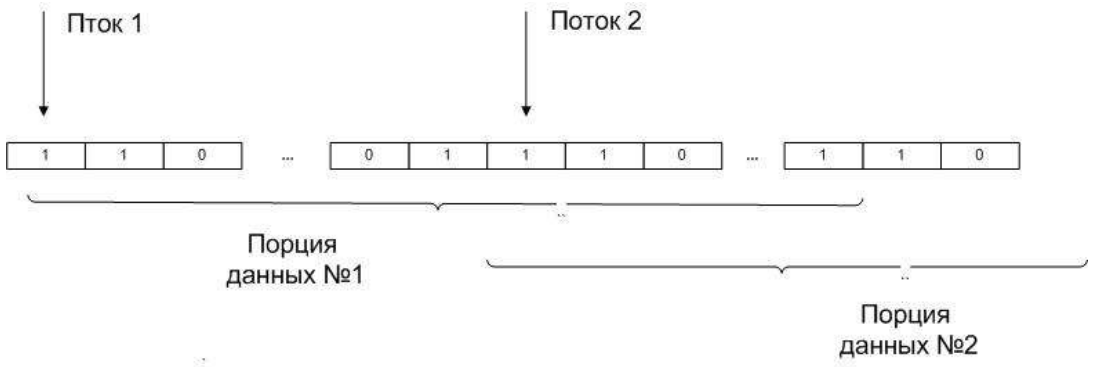

Рис. 2. Перекрытие порций данных

Вторая (при условии незначительных размеров порций данных для каждого потока) может быть решена путём копирования исходных данных в общую память (shared memory) мультипроцессора. Данная процедура может быть реализована очень быстро, благодаря коллизиям при обращениях к глобальной памяти.

Следует заметить, что для повышения производительности массив данных в общей памяти должен индексироваться так, что бы соседние символы находились в соседних блоках. Это дает возможность избавиться от коллизий при обращениях к общей памяти.

Данный алгоритм позволяет значительно ускорить процесс токенизации, но он также не лишен недостатков:

1. При значительном объёме входных данных необходим значительный объем общей памяти, а это снижает количество блоков потоков, которое может выполняться на одном мультипроцессоре одновременно, что снижает производительность.
2. Необходим контроль попадания токена на границу блоков данных.
3. Требуется дублирование данных в общей памяти мультипроцессоров.

Рассмотрим подробнее проблему вхождения токенов друг в друга. Пусть мультипроцессор может одновременно выполнять два потока, порция данных на один поток составляет четыре символа, токен может состоять из любых непробельных символов, а текст для токенизации состоит всего из одного слова – к примеру «молоко», тогда при простом распараллеливание по входным данным мы получим два токена «моло» и «ко».

Если увеличить порцию данных до шести символов, а размер перекрытия порций сделать равным двум, как это было предложено ранее, то результатом станут два токена – «мо-

локо» и «ко». Это может быть исправлено на этапе постобработки, однако это может значительно увеличить время работы алгоритма.

Для устранения отмеченных недостатков предлагается следующий алгоритм:

1. Каждый символ входного текста обрабатывается одним потоком.
2. В общей памяти каждого мультипроцессора каждый блок потоков загружает $N + M$ символов, где N – количество потоков в одном блоке, M – максимальная длина токена.
3. Таким образом, решается вопрос минимизации выделения общей памяти и максимизируется количество блоков потоков, обрабатываемых на одном мультипроцессоре.

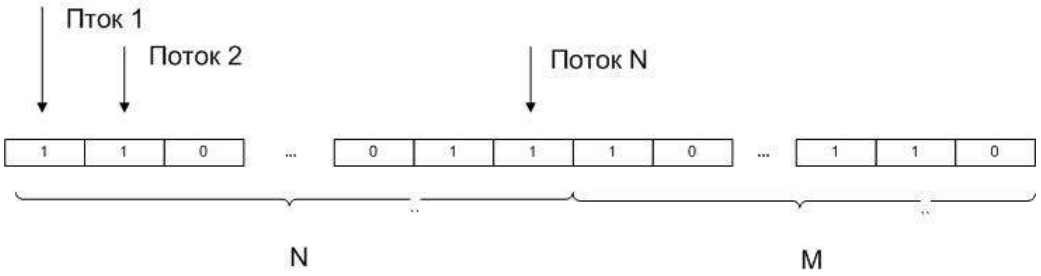

Рис. 3. Один поток на один символ

Токен начинается не с любого разрешённого символа, а с пары символов – неразрешённый затем разрешённый. Это позволит избавиться от постпроверки токенов на вхождение друг в друга.

Такой алгоритм токенизации для одного потока, может быть реализован в виде простого конечного автомата.

Разобьём все символы алфавита на четыре класса:

1) Символы, которые могут входить в токен на любой позиции.
2) Символы, которые не могут входить в токен.
3) Символы знаков препинания.
4) Символы, которые могут находиться только внутри токена, т.е. не могут находиться на первой и последней позициях.

Типы символов вычисляются по массиву в константной памяти, где индекс элемента соответствует ASCII-коду соответствующего символа. Например, символу ‘т’ соответствует ASCII-код равный 231, соответственно в 231-й ячейке массива будет храниться 1, т.е. это символ, который может входить в токен в любой позиции.

Рассмотрим все варианты комбинаций из двух символов. Для этого построим таблицу, где индексу строки соответствует номер класса первого символа, а номер столбца – классу второго символа.

Таким образом, мы получим шесть видов комбинаций:

1. Символ внутри токена.
2. Последний символ в токене.
3. Символ перед словом.
4. Знак препинания.
5. Игнорирующиеся символы.
6. Знак препинания, за которым следует начало слова.

Таблица 1

Пары символов

Виды комбинаций				
	1	2	3	4
1	1	1	2	2
2	1	4	4	4
3	3	5	5	5
4	6	4	4	4

Для повышения производительности данная таблица помещается в константную память видеопроцессора.

Необходимо отметить, что для подобного алгоритма будет свойственно избыточное выделение памяти, так как токен может начинаться с каждой позиции во входном тексте. Блоки потоков не могут быть синхронизированы между собой, а следовательно, необходимо резервировать память под количество токенов, равное количеству потоков, которое в свою очередь равно количеству символов во входном тексте, а следовательно, большая часть памяти будет соответствовать символам, с которых не начинаются токены, для удаления которых потребуется постобработка.

Если она будет выполняться на *CPU*, то потребуется совершить копирование значительного количества «пустых данных» из памяти графического процессора в оперативную память. Это процедура довольно медленная, и для сокращения времени её выполнения необходимо минимизировать объём данных.

Обработка ошибок и исключений

Рассмотрим ошибки, порождаемые предложенной ранее классификацией символов. Очевидно, что некоторые символы могут быть как разделителями между токенами, так и их частью –, например, «точка», которая может выступать как терминальный символ естественного языка, так и частью *URL* или символом, разделяющим целую и дробную части числа.

Для решения этой проблемы реализуется последующая проверка последовательностей токенов. В ходе выделения токены классифицируются как числовые, кириллические, латинские, смешанные и т.д. Затем определяются последовательности, которые могут быть объединены в один токен.

Данные последовательности записываются в виде векторов фиксированной длинны в константную память видеопроцессора. В первый элемент вектора помещается действительная длина последовательности.

Стоит отметить, что подобный подход имеет ряд недостатков:

1. Ограничена максимальная длинна последовательности.
2. Часть выделенной константной памяти не будет использована.

При проверке последовательностей распараллеливание по данным происходит аналогично процедуре поиска токенов.

Результаты испытаний

Для тестирования производительности предложенных алгоритмов использовалось следующее оборудование и программное обеспечение:

1. Процессор Intel Core 2 Duo E6700.
2. Оперативная память DDR 2 объёмом 4ГБ.
3. Видеопроцессор GeForce GTS 250.
4. Видеопроцессор GeForce GTS 450.
5. Операционная система Windows 7 x64.

Все эксперименты проводились на случайных текстах размером порядка одного Гбайта.

Таблица 2

Результаты тестирования

Условия эксперимента	Производительность Мбайт/с
Аналогичный алгоритм на CPU (один поток)	10.1
GeForce GTS 250	55.2
GeForce GTS 450	85.3

Следует отметить, что при использовании видеопроцессора одновременно для расчётов средствами технологии CUDA и отображения информации на мониторе, его производительность значительно снижается. По этой причине для отображения использовалась интегрированная видеокарта.

Выводы

Проведено исследование, которое наглядно показывает, что область применения современных графических процессоров не ограничивается обработкой видео. Этот класс устройств позволяет значительно повысить производительность обработки текстовой информации, что немаловажно для широкого круга задач корпусной лингвистики, где объёмы исходных данных значительны.

Необходимо заметить, что предложенный алгоритм реализует довольно простые правила токенизации естественного языка, что влечёт за собой появление ошибок и неточностей, что делает его малопригодным для синтаксического анализа текста. Однако при решении таких задач, как поиск ключевых слов и словосочетаний частотными методами для рубрикации и определения тональности текста, а также при определении авторства – данными ошибками можно пренебречь.

Очевидно, что при использовании более мощных видеопроцессоров последних поколений (GeForce 400 и 500) производительность данных алгоритмов значительно возрастёт, однако и использование относительно старых устройств позволяет получить показатели выше, чем у современных центральных процессоров.

Библиографический список

1. **Nvidia, C.** Compute Unified Device Architecture Programming Guide/ C. Nvidia. – Santa Clara, CA. 2012.
2. **Munshi, A.** The OpenCL specification version 1.0. Khronos OpenCL Working Group. 2009.
3. **Harris, M.** GPGPU: General-purpose computation on graphics hardware / M. Harris, D. Luebke // In: Proceedings of the International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2005. Courses. – Los Angeles, California. 2005.

Дата поступления
в редакцию 08.02.2013

O.V. Averin

GPU TOKENIZER

Nizhny Novgorod state technical university n.a. R.E. Alexeev

Purpose: Effective algorithm development for text tokenization graphical processor with native language.
Design/methodology/approach: Varity of algorithms for parallel text tokenization are to be implemented for SIMT architecture. Row of experiments is to be applied to verify effectiveness and performance.
Results and scope of use: Possibility of graphical processors usage for text tokenization has been successfully confirmed within held experiments and therefore developed algorithm can be applied for the wide range of computer linguistic tasks.
Summary: Introduced algorithm allows to significantly speedup text tokenization procedure.

Key words: CUDA, parallel computing, computer linguistics.