Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
Кафедра «Вычислительная техника»

УТВЕРЖДАЮ
Заведующий кафедрой

_____ А.И. Легалов
подпись

«_____» _____ 2016 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Анализ геномных данных с использованием графических ускорителей

09.04.01 Информатика и вычислительная техника
09.04.01.01 Высокопроизводительные вычислительные системы

Научный руководитель	_____ подпись, дата	к.т.н., доцент	Д.А. Кузьмин
Выпускник	_____ подпись, дата		А.А. Сычев
Нормоконтроллер	_____ подпись, дата		В.И. Иванов
Рецензент	_____ подпись, дата	науч. сотр.	Ю.А. Путинцева

Красноярск 2016

ВВЕДЕНИЕ

Биоинформатика стала важной частью многих областей биологии. Инструменты биоинформатики помогают в сравнении генетических и геномных данных и, в целом, в понимании эволюционных аспектов молекулярной биологии. Изучение генома человека и других живых существ имеет важное прикладное значение. На основании результатов сборки генома конкретного человека возможна реализация персонифицированной медицины определения предрасположенности человека к различным болезням, создание индивидуальных лекарств и т.д. Именно биоинформатика изучает математические методы компьютерного анализа в сравнительной геномике [1].

Еще одно важное приложение исследования ДНК генетические заболевания. У особей, зараженных одним генетическим заболеванием, наблюдаются одинаковые изменения в ДНК, это можно использовать в медицине, как для теоретического исследования заболевания, так и для лечения от него.

Процесс получения и анализа генома можно разбить на три основных этапа:

- секвенирование молекул ДНК, содержащих информацию о геноме, при помощи специальных устройств секвенаторов;

- сборка геномной последовательности, при помощи специальных компьютерных программ;

- анализ и сравнение геномов, при помощи специальных компьютерных программ.

Задача разработки методов сборки геномных последовательностей является, в определенном смысле, центральной среди всех задач биоинформатики. Это объясняется тем, что без ее решения нельзя приступить к детальному изучению генома живого существа и его анализу.

1 Геном

В нуклеотидной последовательности ДНК записана генетическая информация обо всех признаках вида и особенностях особи, последовательность можно представить в виде текста, где буквами будут являться нуклеотиды.

Нуклеотид – это молекула, состоящая из азотистого основания, сахара и фосфатной группы. Следующие нуклеотиды входят в состав ДНК: аденин (A), цитозин (C), тимин (T), гуанин (G). В рамках настоящей работы под нуклеотидами мы будем понимать буквы алфавита {A,C,T,G}.

Геном – это совокупность всех последовательностей ДНК организма. В рамках данной работы под геномом мы будем подразумевать строку над четырехбуквенном алфавитом нуклеотидов {A,C,T,G} [2]. Длина генома может составлять от нескольких сотен до десятков миллиардов нуклеотидных оснований.

Гены занимают очень небольшую часть генома, но при этом составляют его основу. Информация, записанная в генах, подобна «инструкции» для изготовления белков, главных строительных кирпичиков нашего тела. «На плечах» генов лежит огромная ответственность за то, как будет выглядеть и работать каждая клетка и организм в целом. Они управляют нашей жизнью от момента зачатия до самого последнего вздоха, без них не функционирует ни один орган, не течет кровь, не бьется сердце, не работают печень и мозг.

2 Секвенирование

Метод чтения последовательности белков был разработан еще в начале 50-х годов, даже до открытия двойной спирали ДНК, умели немного читать короткие последовательности РНК, а последовательности ДНК не умели читать вообще.

Все попытки прочитать ДНК оказывались безуспешными и уже стало казаться, что это слишком сложная задача. Однако середине 70-х годов XX века произошел прорыв. Первый метод секвенирования, который учёные сумели применить для обработки целых геномов, был разработан британским ученым-химиком Фредериком Сенгером [4].

Для начала нужно определится с тем, что такое секвенирование. Секвенирование – это биологический процесс определения последовательности нуклеотидов. Существуют разные методы секвенирования, но нас интересует только то, что мы получаем в результате этого процесса. В процессе секвенирования молекулу ДНК разрезают на множество отрезков длинной от 100 до 1000 нуклеотидов, в результате считывания образуется множество фрагментов «текста» ДНК, такие фрагменты еще называют «ридами». Рид – это отдельная последовательность нуклеотидов, полученная в результате секвенирования.

В результате секвенирования по Сэнгеру получаются риды длиной около тысячи нуклеотидов, это очень хороший показатель. Риды такой длинны хорошо поддаются сборке и так как технология обладает высокой точностью (99.9%), можно говорить о низком проценте ошибок. Секвенирование по Сэнгеру доминировало на протяжении 25 лет, но его всё активнее вытесняют другие методы, и применяется оно всё реже. Именно при помощи этого метода секвенирования был впервые расшифрован геном человека. Однако, несмотря на то, что технология используется и в наши дни, она весьма дорогостоящая.

На протяжении последних нескольких лет одной из самых популярных технологий секвенирования является технология Illumina [5], которая позволяет получать достаточно точные данные (98%), при этом стоимость чтения одного нуклеотида в десятки раз меньше, чем стоимость данных, полученных по методу Сэнгера. Существует много разных методов, отличающихся довольно существенно, но нас интересует только результат секвенирования. На каждом цикле мы прочитываем одновременно очень большое число нуклеотидов из разных последовательностей. Но за это приходится платить тем, что участки

ДНК, которые мы можем прочесть, оказываются гораздо короче, чем в случае секвенирования по Сэнгеру – в зависимости от метода выбранного метода секвенирования риды обычно получаются длиной около 100-300 нуклеотидов.

В процессе чтения генома полученные данные представляют собой множество фрагментов геномной последовательности. В результате возникает задача ассемблирования, то есть сборки геномной последовательности из полученных фрагментов.

Сборка генома – это объединение большого числа ридов в одну или несколько длинных последовательностей, в результате чего должна быть восстановлена исходная последовательность генома.

3 Ассемблирование

После процесса секвенирования мы получаем множество небольших ридов. Теперь наша задача состоит в том, чтобы собрать исходную последовательность генома из ридов. Сборка происходит с помощью специально разработанных для этого программ – Ассемблеров.

Геномный ассемблер – это программа для восстановления исходной последовательности из большого числа ридов, путём объединения их в одну или несколько (контиги, скаффолды) длинных последовательностей [6]. Как правило, ассемблерам не удается восстановить весь геном целиком, но из множества коротких чтений они позволяют получить контиги или более крупные последовательности скаффолды. Контиг – это последовательность нуклеотидов, полученная путем объединения перекрывающихся ридов. Скаффолд – это упорядоченные последовательности контигов, разделенные промежутками.

Точная сборка этого «стройматериала» в геномную последовательность – проблема не из лёгких, и решаться она может двумя разными путями, в зависимости от наличия «эталонного» генома.

Референсная сборка – это общепринятый пример генетического кода того или иного организма, хранящийся в цифровом виде. Например, референс генома человека полученный вследствие проекта «Геном человека». Ассемблирование по референсу представляет собой задачу картирования полученных нами нуклеотидных последовательностей на эталонный геном.

Сборка de novo – это первичная сборка генетического кода организма. При de novo ассемблировании мы уже не можем опираться на какой либо референс, что делает эту задачу алгоритмически сложной и вычислительно затратной. Полученный результат сборки de novo далеко не всегда соответствует ожиданиям, что делает задачу создания новых геномных ассемблеров еще более актуальной.

В данной работе будем рассматривать именно de novo сборку генома. Существует множество совершенно разных ассеблеров для сборки de novo генома, но все их можно разделить на два вида: OLC ассеблеры, ассемблеры основанные на графах де Брюйна.

3.1 Ассемблеры основанные на графах де Брюйна

Из полученных фрагментов генома выделяются под последовательности фиксированной длины. То есть каждый рид можно представить как множество не больших последовательностей определённой длинны, называемые k-мерами. Основываясь на информации о перекрытиях между k-мерами строится граф. Все пары k-меров x и y, имеющих общую подстроку длины k-1, таких что последние k-1 нуклеотидов x совпадают с первыми k-1 нуклеотидами y, нужно соединить направленным ребром (x,y). Любой путь в построенном графе соответствует, какой либо геномной последовательности [7].

Рассмотрим работу алгоритма на примере генома из 14 символов. Если взять k=3, то получится граф де Брюйна, изображенный на рисунке 1 .

Из всех путей, начало и конец которых совпадают с парой чтений, вызывают интерес только те, которые укладываются в априорные границы длин

фрагментов, поэтому слишком короткие слишком длинные пути можно отбросить.

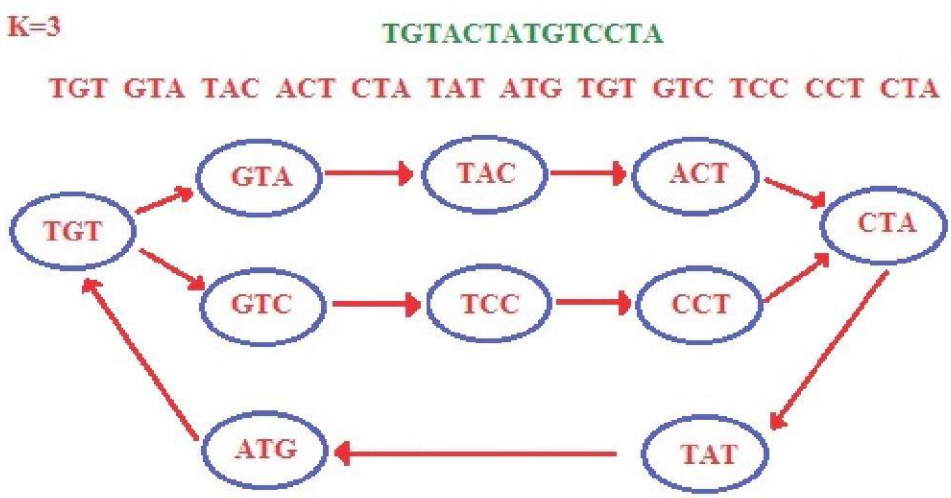

Рисунок 1 – Граф де Брюйна

Оставшиеся пути – «хорошие» кандидаты на роль пути, соответствующего фрагменту геномной последовательности в действительности. Если такой путь единственный, то можно с очень большой уверенностью сказать, что он соответствует реальной подстроке геномной последовательности. В результате задача о сборке генома сводится к задаче нахождения Эйлерова пути в графе. Эйлеров путь – это путь, проходящий по всем ребрам графа и только по одному разу.

Минусы ассемблеров основанных на графах де Брюйна:

- Работают только с короткими ридами;
- Сильная зависимость от ошибок;
- Сильная зависимость от параметра k;
- Не корректно работают с большими геномами (более 10 млрд.н.о.).

3.2 Overlap-layout-consensus ассемблеры

Методы, использующие этот подход, пытаются составить исходную ДНК, путем поиска пересечений между полученными чтениями [6].

Условно работу OLC ассемблеров можно разбить на три этапа:

- поиск перекрытий;
- объединение перекрытий;
- построение консенсус последовательности генома.

Первое что мы делаем, это находим все возможные перекрытия. Перекрытие называется ситуация, когда часть контига X совпадает с частью контига Y. Для поиска перекрытий между двумя последовательностями используются разные математические алгоритмы. Например, алгоритм Нилдмана-Вунша, суффиксный массив, хэш-таблица.

Далее нужно объединить перекрытия (layout). Для объединения всех найденных перекрытий, как правило, используют различные виды графов [7]. Например, граф перекрытий, вершины которого - риды, а ребра это перекрытия ридов. Все ребра имеют свой вес, который зависит от величины сдвига перекрытий вершин графа. Чем больше вес ребра, тем больше вероятность, что это не случайный повтор. После того как граф построен его нужно упростить. Под процессом упрощения понимается объединение схожих путей, удаление ребер с малым весом, удаление лишних ответвлений и исправить ошибки.

Теперь по построенному графу нужно построить консенсус последовательность. Например, нахождение в графе Гамильтонова пути. Гамильтонов путь – это такой путь, который проходит через каждую вершину ровно один раз.

Не во всех случаях получается построить полную геномную последовательность, в большинстве случаев мы получим набор контигов или скаффолдов.

До появления методов секвенирования следующего поколения описанный подход был очень распространён для сборки геномов. OLC

ассемблеры хорошо себя зарекомендовали, при работе с большими ридами, но при работе с короткими ридами они уже не так эффективны. Ассемблеры этого типа уже не поддерживаются, соответственно отсутствует параррельная реализация.

4 Постановка задачи

Геномы растений имеют очень большой размер, так например геном сибирского кедра (лат. *pinus sibirica*), который исследуют в сибирском федеральном университете, имеет примерный размер 28 млрд. н.о.. Ассемблирование таких больших данных это настоящая проблема. Существующие ассемблеры не в состоянии собрать такой большой геном сразу. В частности, ассемблеры основанные на графах Де Брюйна не могут справится с таким количество данных, а OLC ассемблеры уже давно не поддерживаются и они изначально не были рассчитанные на такие объемы. В такой ситуации сборку больших геномов приходится осуществлять по частям. В результате мы получаем несколько наборов контигов, относящихся к данному геному, и перед нами стоит задача, сравнить и объединить данные полученные в результате разных сборок в единый набор контигов. Нам требуется программа способная объединить контиги из разных сборок.

Цель работы: Реализовать программу объединения контигов полученных в результате ассемблирования большого генома, по «частям», в единую сборку.

Требования к программе: Программа должна корректно работать с большими объёмами данных на параллельной архитектуре.

5 Поиск аналогов

Для реализации нашей задачи наиболее подходящим для нас являются ассемблеры, основанные на OLC подходе. Одним из типичных представителей, которых является CAP ассемблер [8].

Как и любой OLC ассемблер, CAP имеет три фазы работы: определение перекрытий, формирование перекрытий, составление консенсус последовательности. CAP вычисляет перекрытия между всеми входящими парами последовательностей, фильтрует данные и для каждого перекрытия устанавливает свой вес. Для поиска наилучшего перекрытия между парой последовательностей используется алгоритм Смита-Ватермана.

На втором этапе ассемблер строит граф, в него добавляем фрагменты с наилучшим перекрытием, полученные ранее, один раз.

И на последнем этапе по графу, исходя из параметров каждого перекрытия, находим наилучший путь и собираем консенсус последовательность.

CAP ассемблер хорошо работал с данными полученными в результате секвенирования по Сэнгеру. Для работы с данными секвенирования нового поколения ассемблер был доработан и получил название CAP3.

Данный ассемблер не справляется с большими геномами и не поддерживает современные параллельные архитектуры. Тем не менее, алгоритмы используемые в данном ассемблере могут быть использованы для реализации нашей задачи.

6 Выбор алгоритма

Требования к алгоритму сравнения контигов:

- алгоритм должен хорошо работать с длинными последовательностями;
- алгоритм должен хорошо распараллеливаться;
- алгоритм экономно использовать оперативную память.

Исходя из данных требований подход, основанный на графах, не подходит для нашей задачи, так как графы плохо работают с длинными последовательностями. Остановимся на OLC подходе поиска перекрытий. В CAP ассемблере используется алгоритм Смита-Ватермана. При этом поиск перекрытий между контигами это, по сути, задача поиска подстроки в строке

или задача вычисления редакционного расстояния, которую хорошо выполняет алгоритм Вагнера-Фишера.

Рассмотрим подробнее перечисленные алгоритмы с точки зрения применимости для решения нашей задачи.

6.1 Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние) между двумя строками – минимальное количество операций вставки одного символа на другой, необходимых для преобразования одной строки в другую [9].

Редакционное расстояние активно применяется:

- проверка орфографии и исправление ошибок ввода в поисковых системах и текстовых редакторах;

- в биоинформатике для определения схожести белков.

Редакционное предписание – это последовательность операций необходимых для получения из первой строки второй кратчайшим путем (рисунок 2). Существует четыре вида операций: I – вставка, D – удаление, M – совпадение, R –замена.

Рисунок 2 – Редакционное предписание

Для каждой операции можно определить свою цену. В общем случае:

- w(ε,b) – цена вставки символа b;

- w(a, ε) – цена удаления символа a;

- w(a,b) – цена замены символа a на символ b.

Решением задачи о редакционном расстоянии будет являться, последовательность замен, обладающая минимальной суммарной ценой.

Расстояние Левенштейна является частным случаем этой задачи при:

- w(a,a) = 0;
- w(a,b) при a ≠ b;
- w(ε,b) = 1;
- w(a, ε) = 1.

Задачу для произвольных w, решает алгоритм Вагнера-Фишера, рассмотрим его ниже. Будем считать, что все w неотрицательны.

Пусть S_1 и S_2 – две строки (длиной M и N соответственно) над некоторым алфавитом, тогда редакционное расстояние d(S_1,S_2) можно подсчитать по следующей рекуррентной формуле d(S_1,S_2) = D(M,N) где:

$$D(i,j) = \begin{cases} 0 & ; \ i = 0, \ j = 0 \\ i & ; \ j = 0, \ i > 0 \\ j & ; \ i = 0, \ j > 0 \\ \min(& \\ \quad D(i, j-1) + 1, & \\ \quad D(i-1, j) + 1, & ; \ j > 0, \ i > 0 \\ \quad D(i-1, j-1) + \mathrm{m}(S_1[i], S_2[j]) & \\ ) & \end{cases}, \quad (1)$$

где m(S_1[i],S_2[j]) – равна нулю, если S_1[i]= S_2[j] и единице в противном случае;

min() – возвращает наименьший из аргументов.

Здесь шаг по i означает удаление из первой строки, по j – вставку в первую строку, а шаг по обоим индексам означает схожесть символов или их различие.

6.2 Алгоритм Вагнера-Фишера

Алгоритм Вагнера-Фишера был предложен Р. Вагнером (R. A. Wagner) и М. Фишером (M. J. Fischer) в 1974 году [10].

Рассчитаем матрицу D, с помощью алгоритма, который мы разобрали выше. Для восстановления редакционного предписания требуется идти из правого нижнего угла (M,N) в левый верхний, и на каждом шаге искать минимальное из трёх значений:

- если минимально (D(i-1, j) + цена удаления символа S1[i]), добавляем удаление символа S1[i] и идём в (i-1, j);

- если минимально (D(i, j-1) + цена вставки символа S2[j]), добавляем вставку символа S1[i] и идём в (i, j-1);

- если минимально (D(i-1, j-1) + цена замены символа S1[i] на символ S2[j]), добавляем замену S1[i] на S2[j] (если они не равны; иначе ничего не добавляем), после чего идём в (i-1, j-1).

Здесь (i, j) – клетка матрицы, в которой мы находимся на данном шаге. Если минимальны два из трёх значений (или равны все три), это означает, что есть 2 или 3 равноценных редакционных предписания.

Для примера работы алгоритма сравним две последовательности “GGTACTTACG” и “AGGCTAGTTAC”. Согласно алгоритму построим матрицу (рисунок 3).

		G	G	T	A	C	T	T	A	C	G
	0	1	2	3	4	5	6	7	8	9	10
A	**1**	1	2	3	3	4	5	6	7	8	9
G	2	**1**	1	2	3	4	5	6	7	8	8
G	3	2	**1**	2	3	4	5	6	7	8	8
C	4	3	**2**	2	3	3	4	5	6	7	8
T	5	4	3	**2**	3	4	3	4	5	6	7
A	6	5	4	3	**2**	3	4	4	4	5	6
G	7	6	5	4	3	**3**	4	5	5	5	5
T	8	7	6	5	4	3	**3**	4	5	6	6
T	9	8	7	6	5	4	4	**3**	4	5	6
A	10	9	8	7	6	5	5	4	**3**	4	5
C	11	10	9	8	7	6	6	5	4	**3**	**4**

Рисунок 3 – Матрица

Далее восстановим редакционное предписание. Пойдем из правого нижнего угла (M,N) в левый верхний, и на каждом шаге искать минимальное из

трёх значений. В итоге по матрице, мы можем увидеть какие символы у последовательностей совпадают.

6.3 Алгоритм Смита-Ватермана

Алгоритм Смита-Ватермана предназначен для получения локального выравнивания последовательностей, то есть для выявления сходных участков двух последовательностей. Алгоритм был опубликован в 1981 году [11].

Алгоритм Смита-Ватермана использует матрицу сходства F(i,j), где i и j изменяются от нуля до длины, соответственно, первой и второй строк. Это позволяет учитывать различные замены с различным весом. Также он позволяет учитывать разные добавления и удаления по-разному в зависимости от того, какой именно нуклеотид был добавлен или удалён.

Реализация алгоритма Смита-Ватермана: берётся матрица F(i,j), где элементы F(i,0) и F(0,j) инициализируются нулями. Расчет матрицы происходит по следующей формуле:

$$F_{i,j} = \max \begin{cases} F_{i-1,j-1} + S_{A_i},_{B_j} \\ F_{i-1,j} + D \\ F_{i,j-1} + I \\ 0 \end{cases}, \quad (2)$$

где S_A,S_B – сравниваемые строки;

D, I – штраф за пропуск;

max() – возвращает наибольший из аргументов.

Теперь максимальная степень сходства будет максимальным элементом матрицы. Если переходить от этого элемента по цепочке предыдущих, то путь закончится в каком-то нулевом элементе не обязательно левом верхнем. Ин-

дексы этих двух элементов равны индексам начал и концов подстрок: первые индексы в первой строке, вторые во второй.

Рассмотрим работу алгоритма на примере сравнения последовательностей «GGCTCAATCA» и «ACCTAAGG». Построим матрицу сходства и найдем перекрытия последовательностей (рисунок 4). Совпадение элементов последовательностей будем считать равным 2, несовпадение равным -1, штраф за пропуск равным -2.

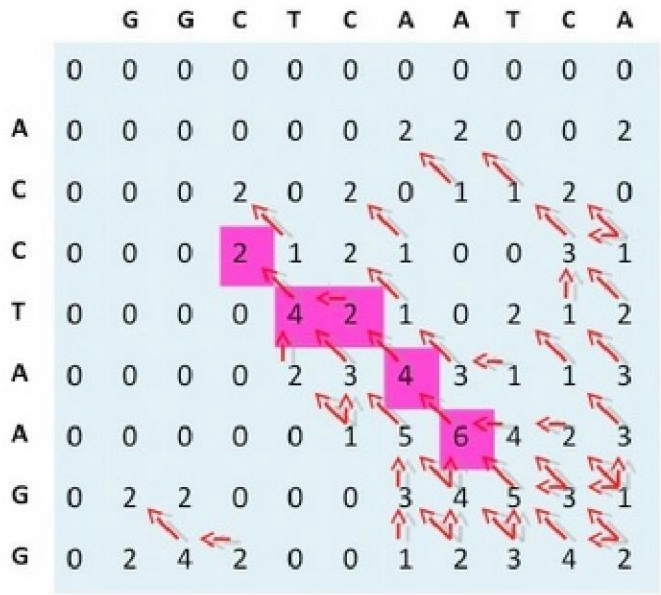

Рисунок 4 – Матрица сходства

Исходя из полученной матрицы можно сделать вывод, что лучшее найденное перекрытие имеет длину 2. Всего найдено два перекрытия: “CT” и “AA”.

6.4 Вывод

К требованиям поставленной задачи подходит алгоритм Смита-Ватермана. Он позволяет найти оптимальные выравнивания между последовательностями, с помощью которых мы можем определить наилучшее перекрытие и объединить последовательности. Алгоритм Вагнера-Фишера для

данной задачи не подходит, поскольку не дает нам данных по конкретным перекрытиям. С помощью алгоритма Вагнера-Фишера удобно искать различия строк над большим алфавитом. Алгоритмы отлично работают с любым типом данных. Процесс построения матрицы можно распараллелить, а значит сократить время выполнения алгоритмов. Для задачи поиска, сравнения и объединения последовательностей в данной работе буду использовать алгоритм Смита-Ватермана.

7 Архитектура NVIDIA CUDA

7.1 Определение

CUDA (Compute Unified Device Architecture) является расширением языка C, которое позволяет на писать расширяемое многопоточное приложение для устройств, поддерживающих данную архитектуру [12]. Программа с использованием CUDA содержит в себе ядро (kernel). Данная функция содержит вычисления, которые будут выполнены одним потоком, ядро исполняется одновременно в нескольких потоках. Потоки объединяются в иерархическую структуру (рисунок 5): блоки состоят из набора потоков, грид (grid) содержит в себе блоки потоков.

Каждому потоку назначается уникальный номер (состоит из номера потока в блоке, и номера блока в гриде). Поток получает свой уникальный номер через свои специальные переменные. При вызове функции ядра нужно явно обозначить число потоков внутри блока и число блоков внутри грида.

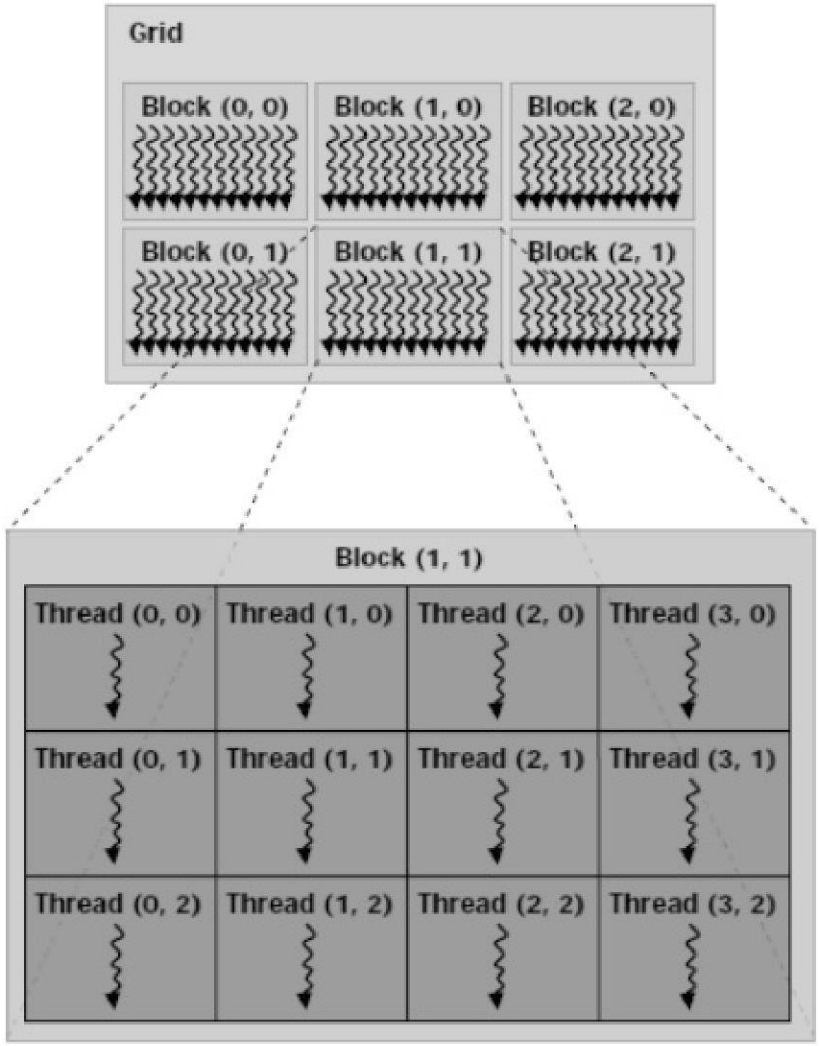

Рисунок 5 – Иерархия потоков в архитектуре CUDA

С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU [12]. CUDA предназначена для разработки GPGPU-приложений без привязки к графическим API и поддерживается многими GPU Nvidia.

Потоки, объединенные внутри блока, имеют средства коммуникации и синхронизации, такие как барьеры синхронизации и разделяемая память. Потоки, принадлежащие разным блокам, не имеют таких средств. Кроме разделяемой памяти блока имеется еще 4 типа памяти: глобальная память доступная всем потокам, внутренняя память потока, текстурная память и память для констант. Текстурная и константная память доступны только для

чтение, и обладают кэшем. Аппаратные средства, поддерживающие архитектуру CUDA, состоят из набора потоковых мультипроцессоров (Streaming Multiprocessor). Мультипроцессор содержит 8 скалярных процессоров (Scalar Processor), которые имеют общий доступ к разделяемой памяти, размером 16Кб. Все потоки внутри блока исполняются на одном мультипроцессоре. Мультипроцессор исполняет потоки группами по 32 потока, называемые варпами (warps). Таким образом, скорость параллельных вычислений может быть ограничена зависимостями по данным, а также ветвлениями.

Рассмотрим подробнее типы памяти в CUDA устройствах:

- глобальная память, доступна на чтение и запись. Как правило, большого объёма. Имеет большую задержку и низкую пропускную способность. Пропуская способность, сильно зависит от способа доступа к памяти, обобщенный доступ может улучшить пропускную способность;

- внутренняя память потока, доступна на чтение и запись. Имеет не большой размер (16Кб), не кэшируется, характеризуется медленным доступом;

- память для констант, доступна только на чтение. Быстрая кэшируемая память не большого размера (64Кб). Скорость чтения зависит от количества потоков, одновременно обращающихся к разным адресам;

- текстурная память, доступна только на чтение. Кэшируемая память большого размера. Доступ к данной памяти выполняется только с помощью специальных инструкций. Чтение из текстурной памяти, быстрее чем чтение из глобальной или локальной памяти;

- внутренняя память блока, доступна на чтение и запись. Быстрая память не большого размера (64Кб). Память делится на банки для одновременного доступа из каждого потока. Скорость падает, если имеются конфликты доступа к банку;

- внутренние регистры потока, доступны на чтение и запись. Обладают наибольшей скорость доступа, но имеют крайне малый размер.

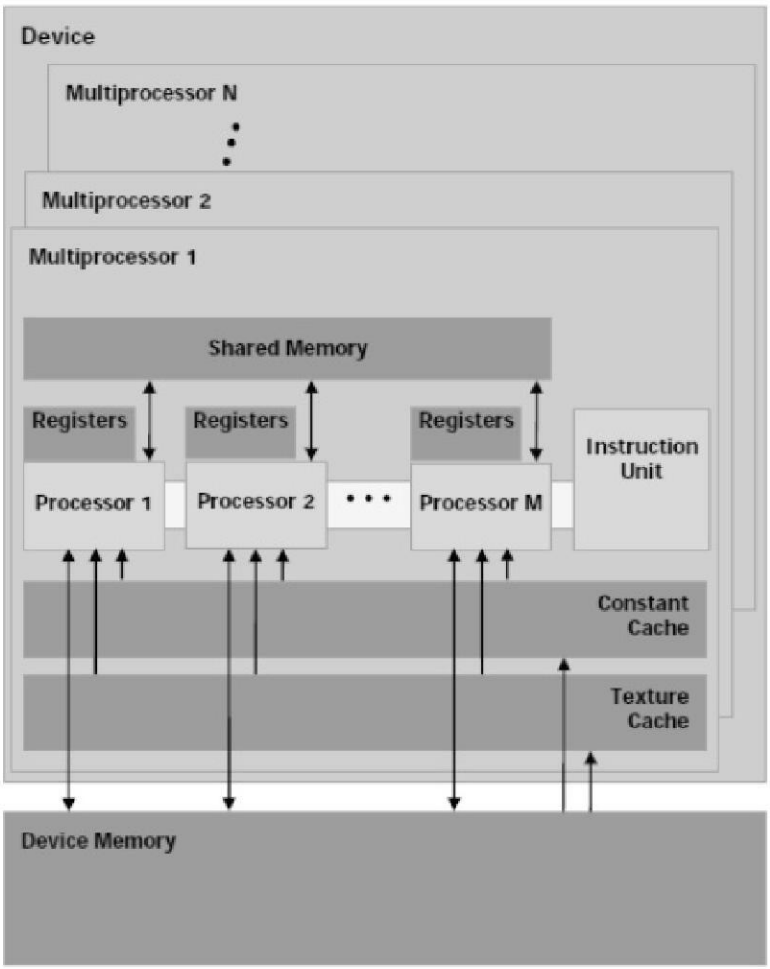

Рисунок 6 – Архитектура памяти Nvidia CUDA

7.2 Сравнение GPU с CPU

До последнего времени ключевым компонентом систем для высокопроизводительных вычислений, включая кластеры, был центральный процессор. Однако несколько лет назад у него появился серьезный конкурент – графический процессор (GPU) .

Высокая производительность GPU объясняется особенностями его архитектуры. В отличие от центрального процессора, который состоит из нескольких ядер, графический процессор изначально создавался как многоядерная структура, в которой количество компонентов измеряется сотнями [13]. Также есть существенная разница и в принципах работы – архитектура CPU предполагает последовательную обработку информации, а

GPU исторически предназначался для обработки компьютерной графики, поэтому рассчитан на массивно параллельные вычисления.

Каждая из этих двух архитектур имеет свои достоинства. Говорить об абсолютной замене CPU на GPU не имеет смысла – они не взаимозаменяют, а дополняют друг друга. CPU лучше работает с последовательными задачами, но при большом объеме обрабатываемой информации, с которой можно работать параллельно, очевидное преимущество имеет GPU.

8 Распаралелливание алгоритма

8.1 Вычисление побочной диагонали

Алгоритм заполнения матрицы можно распараллелить. Для этого согласно методологии канонического отображения необходимо разработать граф зависимости. Граф зависимости элементов матрицы представлен на рисунке 7. На рисунке 7 задано решетчатое дискретное пространство. В узлах решетки располагаются вычисления, которые могут быть назначены реальным или виртуальным вычислительным узлам (например, моделируемым нитями). Связи между вычислениями направленные и задают зависимости по данным в ходе вычисления. В данном пространстве можно выделить множество гиперплоскостей, каждая гиперплоскость объединяет множество вычислений, которые могут быть выполнены одновременно.

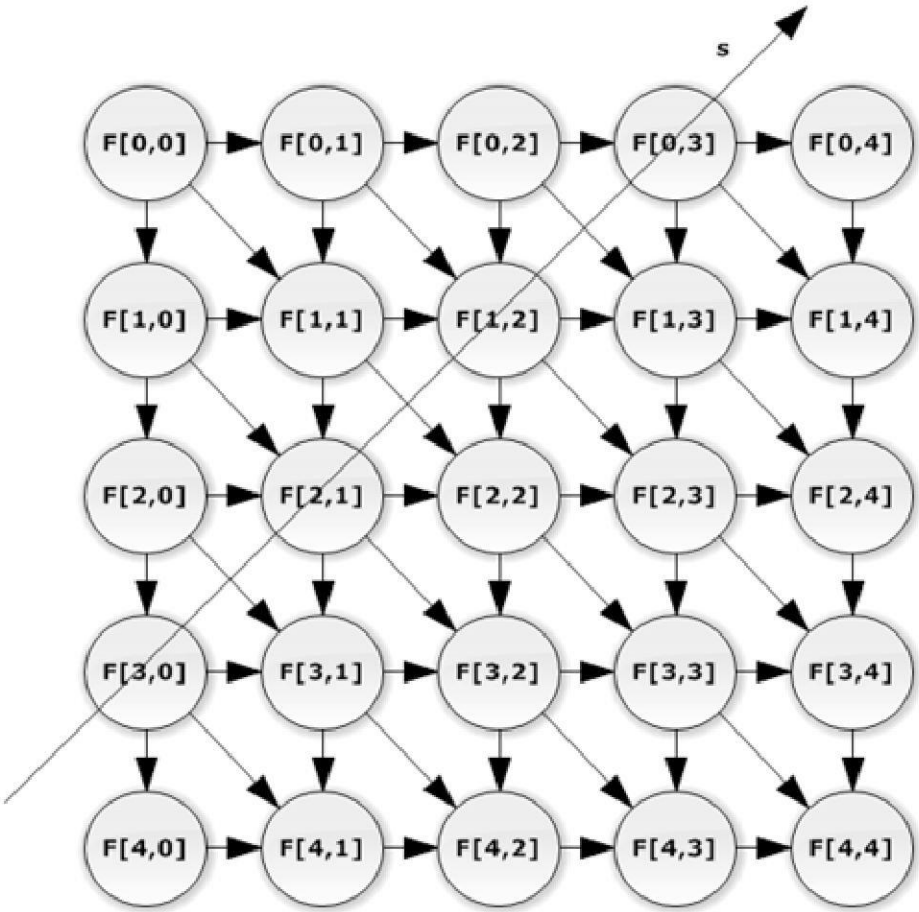

Рисунок 7 – Граф зависимости

Текущая гиперплоскость объединяет вычисления, которые производятся в текущий момент времени. Текущую гиперплоскость будем задавать с помощью вектора s. Далее согласно методологии канонического отображения необходимо составить граф потока сигналов (рисунок 8). На рисунке 8 решетчатое пространство, представленное на рисунке 7, спроецировано на гиперплоскость, параллельную вектору s. В узлах графа зависимости расположены процессорные элементы. Каждому процессорному элементу назначен столбец графа зависимости. Дугами показаны передачи сигналов между процессорными элементами. Данная схема плоха тем, что при использовании программной модели CUDA элементы матрицы будут сохраняться в глобальную память в отдельных транзакциях, так как они не расположены в непрерывном участке памяти.

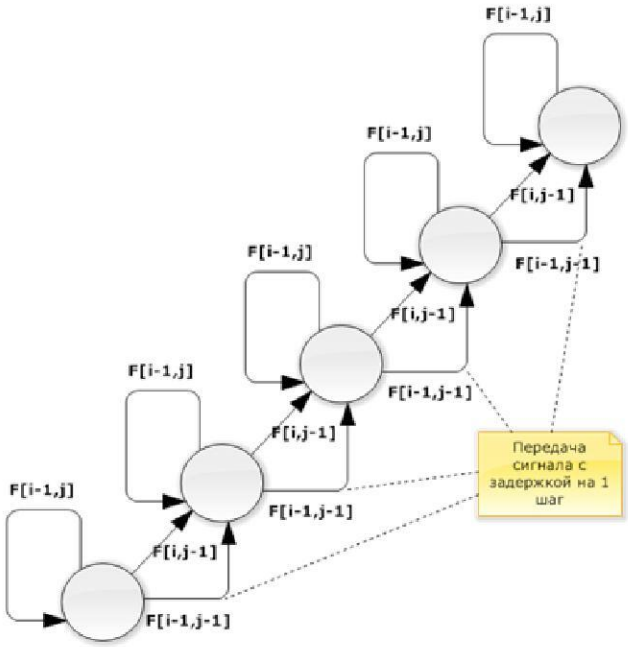

Рисунок 8 – Граф потока сигналов

Для оптимизации обращений к глобальной памяти элементы матрицы должны быть переупорядочены, как представлено на рисунке 9.

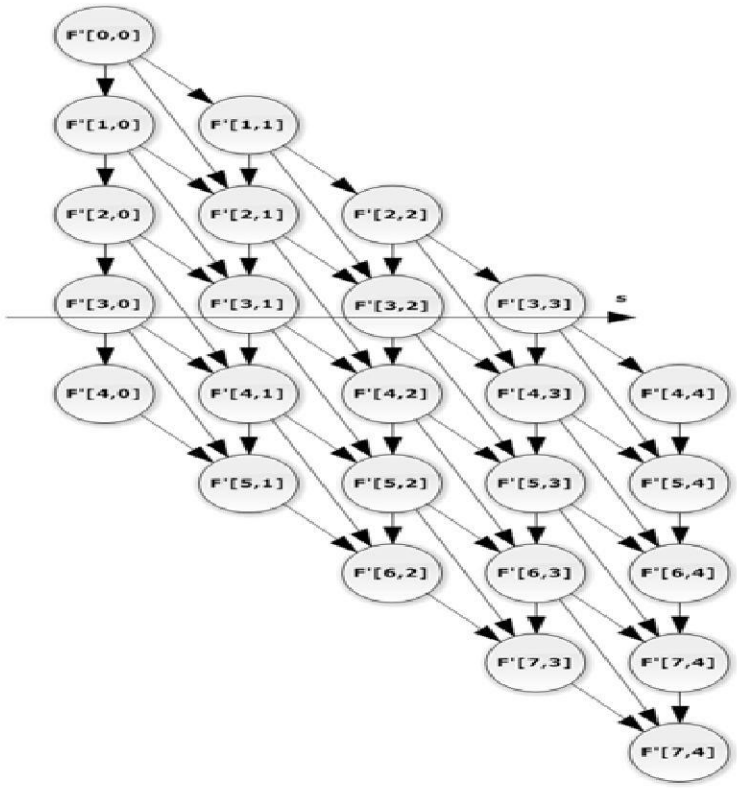

Рисунок 9 – Модифицированный граф зависимости

На рисунке 9 гиперплоскости вычислений расположены вдоль строк матрицы, поэтому при сохранении результатов вычислений в узлах решетки в глобальную память транзакции могут быть объединены, что ускорит работу алгоритма. Составим граф потока сигналов для данного решетчатого пространства.

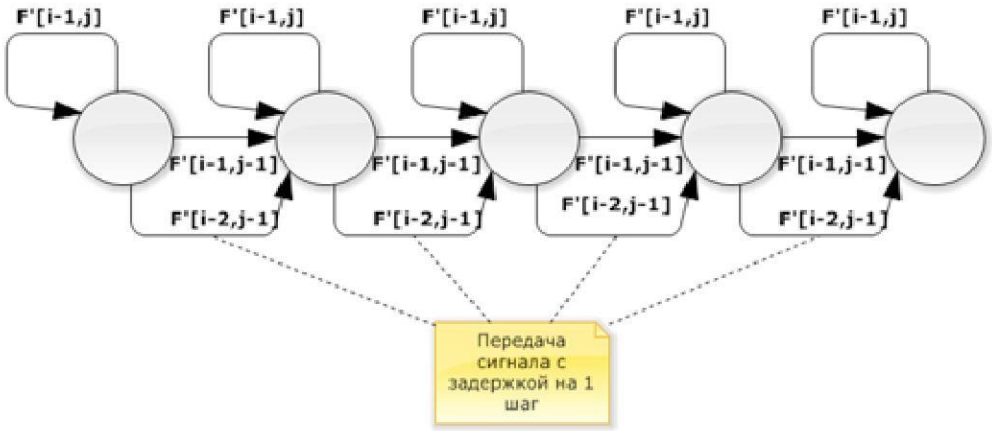

Рисунок 10 – Модифицированный граф потока сигналов

Как видно из рисунков оригинальный и модифицированный графы зависимости и соответствующие им графы потока сигналов изоморфны. Поэтому модель вычислений осталась прежней.

8.2 Перенумерация матрицы

Для того чтобы элементы матрицы располагались как представлено на рисунке 8 необходимо их перенумеровать. Нумерация – это отображение множества натуральных чисел на соответствующее множество [13]. Основное применение нумераций – построение эффективной модели представления данных в памяти [14].

В нашем случае под нумерацией понимается взаимно однозначное отображение из Z^2 в Z^2.

Введем отношение x:V→V', где $V \subset Z^2$ и $V' \subset Z^2$, то есть паре чисел (i,j) ставится в соответствие пара (i,j), значения которых могут быть вычислены по формуле:

$$\begin{cases} i' = i + j \\ j' = j \end{cases} \quad (3)$$

Обратная нумерация x^{-1}:V'→V задается формулой:

$$\begin{cases} i = i' - j' \\ j = j' \end{cases} \quad (4)$$

Данная нумерация переносит элементы матрицы с диагоналей параллельных побочной диагонали в строки, так что они находятся в соседних позициях.

Свойства нумерации:

а) номер строки в новой матрице совпадает с номером итерации цикла;

б) номера столбцов исходной и новой матриц совпадают;

в) число строк новой матрицы равно n+m-1;

г) число столбцов в матрицах совпадает.

Из свойств а и б следует, что во время вычисления значений матрицы нет необходимости вычислять пару (i,j), а значения соседних элементов, необходимых для вычисления, определяются по формулам:

$F_{i-1,j-1}=F'_{i-2,j-1}$,i>1,j>0 (5)

$F_{i-1,j}=F'_{i-1,j}$,i>0 (6)

$F_{i,j-1}=F'_{i-1,j-1}$,i>1,j>0 (7)

Из свойств в и г следует, что размер дополнительной памяти, необходимый для организации матрицы равен (n+m-1)*m-n*m=(m-1)*m слов. Поэтому для более эффективного расхода памяти и лучшей загрузки процессора необходимо чтобы выполнялось условие $n \geq m$.

8.3 Реализация алгоритма на CUDA

Рассмотрим реализацию алгоритма Смита-Ватермана на языке CUDA для семейства GPGPU Nvidia, поддерживающего данную технологию. Язык CUDA является расширением языка C. В него добавлены средства запуска ядра и API для работы и управления GPU, управления выполнением нитей ядра и передачей данных между нитями и между CPU и GPU. Программа на языке CUDA компилируется с помощью специального компилятора, который разделяет программный код для GPU и CPU. Программный код для GPU компилируется с данным компилятором в объектный или ассемблерный код. А программный код для CPU компилируется стандартным C компилятором, затем линковщик соединяет объектные файлы в программу. Каждая нить вычисляет отдельный столбец матрицы в цикле. Для вычисления значений матрицы на CUDA необходимо, чтобы нити работали синхронно. Для этого внутри цикла необходимо поставить барьерную синхронизацию. Однако для упрощения межпроцессорного взаимодействия можно синхронизировать только нити внутри блока, межблочная синхронизация стандартными средствами CUDA невозможна. Поэтому для вычисления элементов матрицы воспользуемся следующим методом, предложенным в [15]. Схема вычислений представлена на рисунке 11.

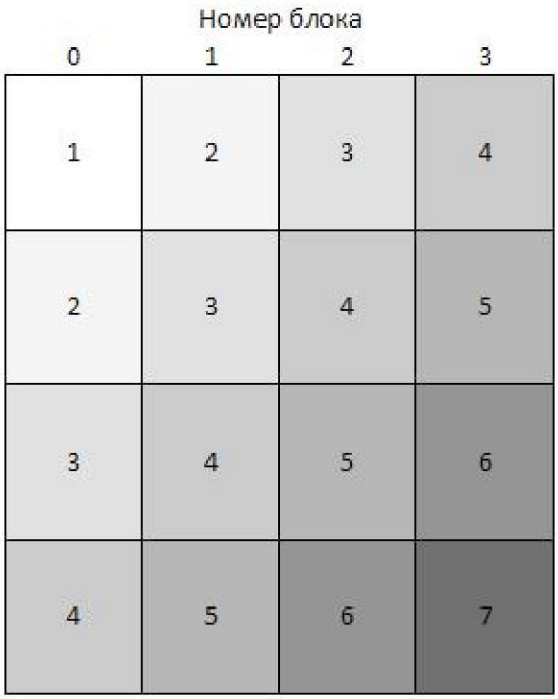

Рисунок 11 – Схема вычислений

Каждый блок из n нитей вычисляет фрагмент матрицы, после чего ядро перезапускается. Блоки, отмеченные на рисунке одинаковыми номерами, могут быть запущены в одном ядре. В данном случае получается параллелизм, как на уровне нитей, так и на уровне блоков. В алгоритме Смита-Ватермана для вычисления значения элементов матрицы методом динамического программирования необходимо знать значения трех соседних элементов матрицы, вычисленных на предыдущих шагах. Их необходимо прочитать из глобальной памяти GPU. Так как данная операция выполняется достаточно долго, можно передавать вычисленные значения напрямую между нитями, минуя глобальную память. Для этого необходимо воспользоваться разделяемой памятью. Разделяемая память имеет небольшой размер, и поэтому находится в кэше, что ускоряет доступ к ней. Однако полностью избавиться от чтения из глобальной памяти невозможно, т. к. в начале работы ядра необходимо проинициализировать разделяемую память, прочитав в нее из глобальной значения полученные ранее. Скорость выполнения программы зависит от размера блока. Чем больше блок, тем меньше число запусков ядра, что ускоряет

работу программы. Однако чем больше блок тем больше накладные расходы связанные с барьерной синхронизацией, и тем больший объем разделяемой памяти необходим блоку, из-за чего разделяемая память может не поместиться в кэш полностью.

8.4 Поиск перекрытий

Для поиска перекрытия потребуется результирующая матрица, построенная с помощью алгоритма Смита-Ватермана. Две строки S1 и S2 считаются равными, если имеется непрерывное выравнивание, проходящее строго по диагонали матрицы.

Перекрытием строк S1 и S2 буду считать непрерывное выравнивание, проходящее по диагонали от последнего столбца к первой строке или от последней строки к первому столбцу.

Возможно перекрытие двух типов:

а) перекрытие начала первой строки и конца второй строки;

б) перекрытие конца первой строки и начала второй строки.

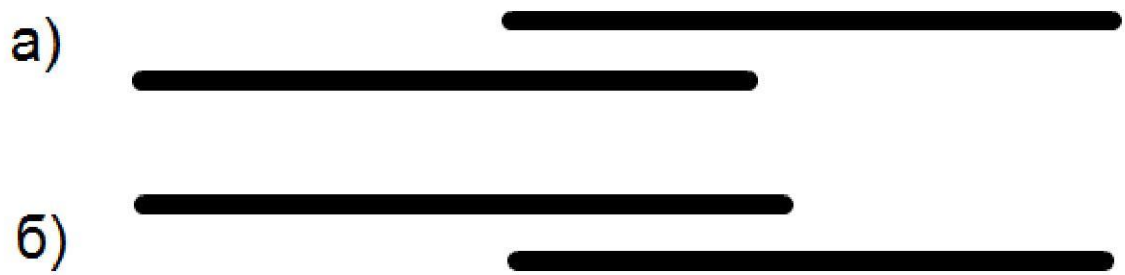

Рисунок 12 – Типы перекрытий

Рассмотрим детальный поиск перекрытия на примере двух строк: «AGGTCAAAATTC» и «ACACCTTTGGAGGTCA».

Построю результирующую матрицу (рисунок 13), где совпадение элементов считаю равным 2, несовпадение равным -1, штраф за пропуск

равным -1. Строки имеют перекрытие равное шести элементам, найти его можно с помощью построенной матрицы.

		A	C	A	C	C	T	T	T	G	G	A	G	G	T	C	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	0	0	0	0	0	0	2	1	0	0	0	2
G	0	1	1	1	1	0	0	0	0	2	2	1	4	3	2	1	1
G	0	0	0	0	0	0	0	0	0	2	2	3	3	6	5	4	3
T	0	0	0	0	0	0	2	2	2	1	3	3	2	5	8	7	6
C	0	0	2	1	2	2	1	1	1	1	2	2	2	4	7	10	9
A	0	2	1	4	3	2	1	0	0	0	1	4	3	3	6	9	12
A	0	2	1	3	3	2	1	0	0	0	0	3	3	2	5	8	11
A	0	2	1	3	2	2	1	0	0	0	0	2	2	2	4	7	10
A	0	2	1	3	2	1	1	0	0	0	0	2	1	1	3	6	9
T	0	1	1	2	2	1	3	3	2	1	0	1	1	0	3	5	8
T	0	0	0	1	1	1	3	5	5	4	3	2	1	0	2	4	7
C	0	0	2	1	3	3	2	4	4	4	3	2	1	0	1	4	6

Рисунок 13 – Результирующая матрица

Нужно найти максимальный элемент в последнем столбце матрицы и от этого элемента пройти по диагонали до строки (рисунок 13).

Цена за совпадение элементов в построенной матрице равно 2, можно с уверенностью сказать, что перекрытие составляет 6 элементов, так как 12/2=6. В этом примере перекрытие находится между началом первой строки и концом второй строки.

Если требуется найти перекрытие между концом первой строки и началом второй, то нужно на йти максимальный элемент матрицы в последней строке и пройти по диагонали в первый столбец. Пройденный путь и будет являться перекрытием.

8.5 Анализ данных

Для тестирования корректности работы алгоритма и программы было выбрано модельное растение Arabidopsis Thaliana, чья длина генома составляет около 157млн. н.о.. На вход программы подается файл, содержащий множество

контигов, которые требуется сравнить, найти в них перекрытия и объединить. Проведу анализ входных данных (таблица 1) и построю график распределения длин контигов (рисунок 14), чтобы провести анализ затрат ресурсов требующихся при работе программы.

Таблица 1 – Анализ входных данных

Наименование характеристики	Значение
Число контигов	106 500
Минимальная длинна контига	200
Максимальная длинна контига	73 000
Суммарная длинна всех контигов	121,4 млн.

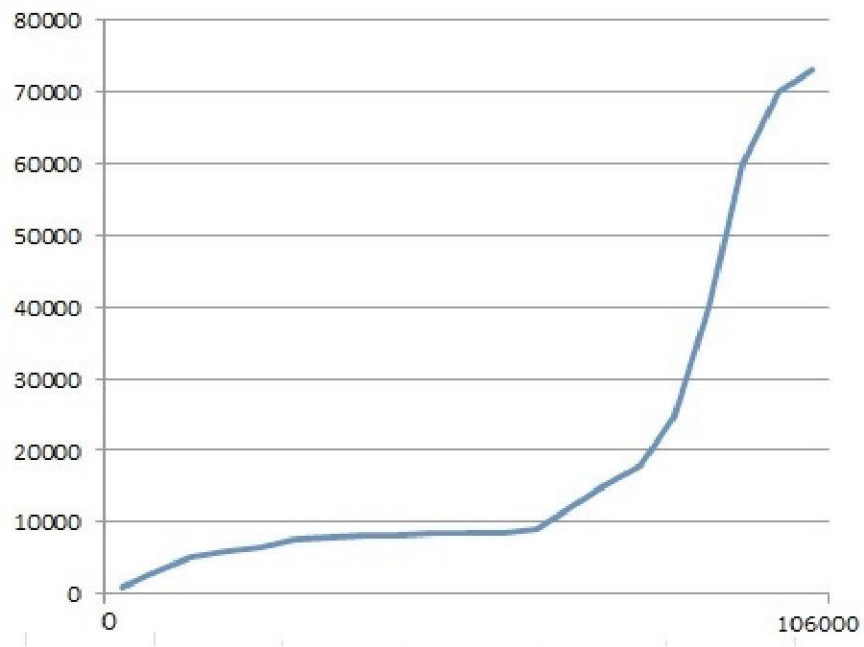

Рисунок 14 – График распределения длин контигов

Требуется проверить, способен ли реализованный алгоритм, справится с такими данными.

Разработка и тестирование прототипа программы производилось на платформе Microsoft Visual C++ и графическом ускорителе NVIDIA GeForce GTX 460. Итоговый вариант программы будет работать на гибридном счетном

сервере IBM dx360 M4 с двумя GPU NVIDIA Teska K20. Сервер включает в себя два процессора Intel Xeon E5-2690, с частотой 2,9 ГГц и два ускорителя Tesla K20. Объем ОЗУ сервера составляет 192 Гб. Объем ОЗУ GPU ограничен в 5 Гб. Операционная система Linux, CentOS 7.15.03 x64. Компиляторы NVCC и GCC. Архитектура приложения – x64.

Перед началом работы алгоритма, все контиги из файла будут помещены в оперативную память. Рассчитаю количество оперативной памяти сервера, необходимое для размещения всех последовательностей. Последовательности хранятся в массивах типа Char. Так как для хранения одного символа типа Char требуется 1 байт памяти, то для хранения 121 млн. н.о. потребуется примерно 116 Мб памяти. Памяти сервера вполне хватит для такого объёма данных.

Для работы функции fill_matrix_kernel (листинг А.2) требуется, чтобы сравниваемые последовательности S1, S2 и результирующая матрица F располагались непосредственно в памяти GPU. Последовательности S1 и S2 размерностью n и m будут потреблять память примерно равную n*sizeof(int) байт и m*sizeof(int) байт. Результирующая матрица F займет n*m*sizeof(unsigned short int) байт памяти.

Максимальная длинна контига может составлять около 73000 н.о., рассчитаем затраты памяти GPU для сравнения двух последовательностей такой длинны.

Необходимая память для последовательностей составит 73000*2*4 байт, а для матрицы 73000*73000*2 байт. Общие затраты памяти GPU будут примерно равны 570Кб + 9Гб. Большие объёмы не позволительны для GPU. Максимальные длины сравниваемых последовательностей не должны превышать 73000 н.о.

Несмотря на колоссальные объёмы памяти, даже при таких больших последовательностях переполнение переменных происходить не будет.

9 Результаты работы алгоритма

Эксперименты проведены на платформе NVIDIA GeForce GTX 460. На рисунке 15 показаны графики времени выполнения ядра в зависимости от размера входных данных и размера блока. Из рисунка видно, что наиболее оптимальный размер блока составляет 256 потоков. При увеличении блока увеличиваются накладные расходы времени на организацию работы блока.

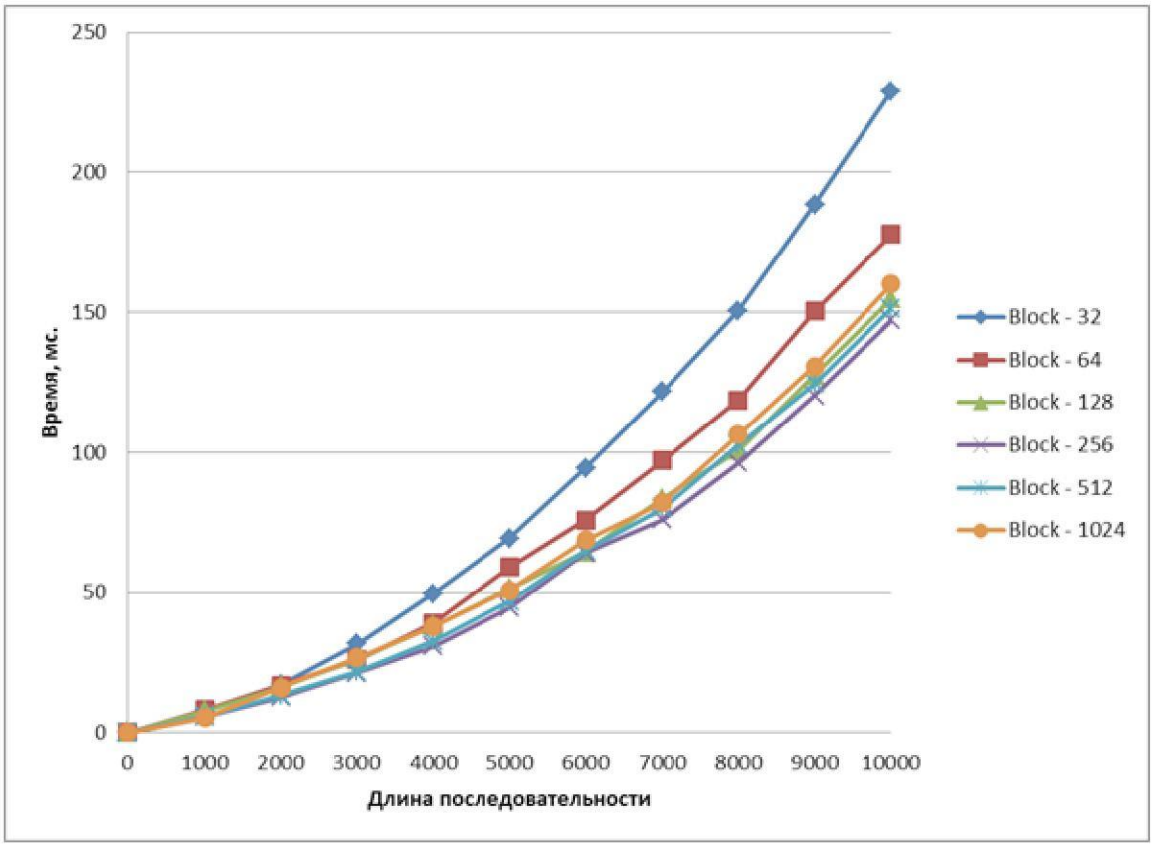

Рисунок 15 – Время выполнения алгоритма на GPU

Для сравнения времени работы алгоритма, проведем аналогичный эксперимент на CPU Intel Core i3. График изображенный на рисунке 16 доказывает эффективность работы алгоритма на GPU.

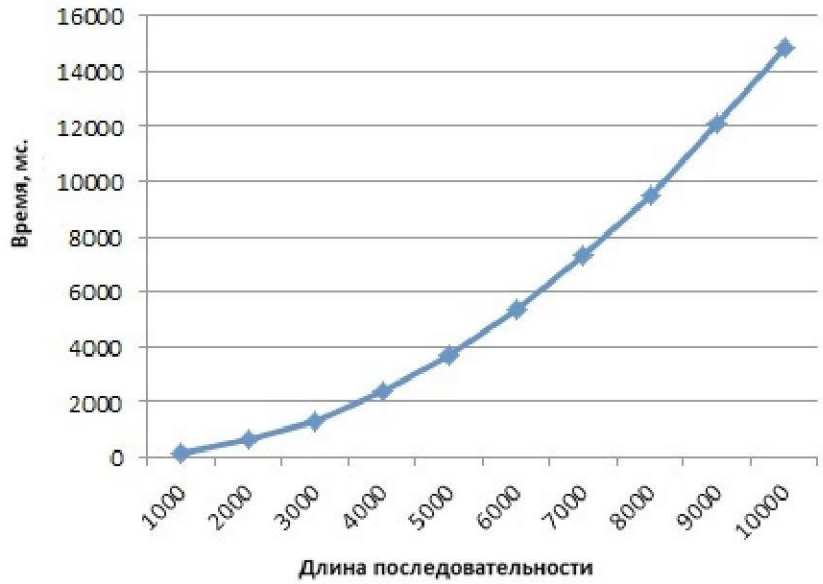

Рисунок 16 – Время выполнения алгоритма на GPU

Размер памяти требуемой алгоритмам имеет одинаковую асимптотику $O(n^2)$,что ограничивает его использование на больших последовательностях. Однако память является менее критичным ресурсом, чем время. Обойти данное ограничение можно, либо увеличив количество памяти на GPU, либо используя кластеры из GPU.

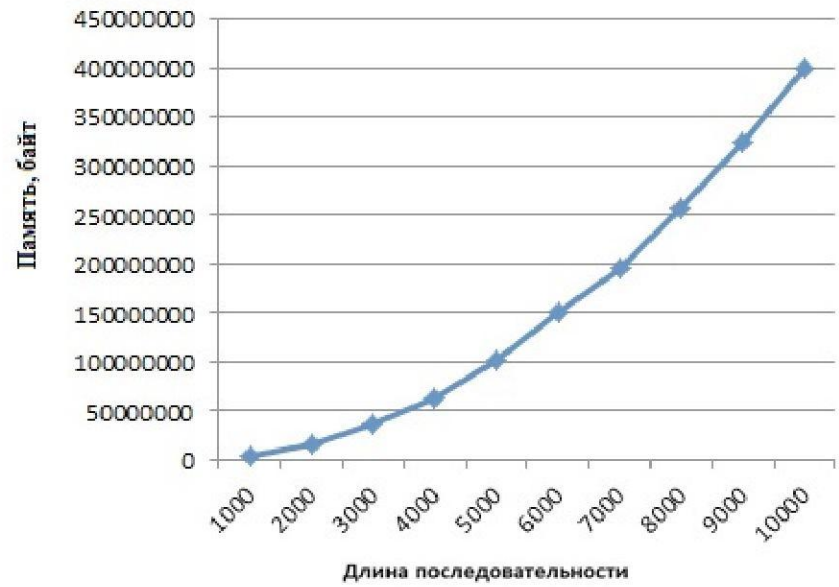

Рисунок 17 – Размер памяти требуемый алгоритмом

ЗАКЛЮЧЕНИЕ

В результате работы была разработана программа для сравнения и объединения длинных контигов. Алгоритм Смита-Ватермана применяемый для определения перекрытий контигов, был успешно реализован на архитектуре CUDA. Реализация алгоритма показала свое быстродействие в ходе экспериментов.

Созданное ПО, позволяет сократить расходы времени на ассемблирование длинных контигов. Также программа является единственным, в своем роде, решением для сборки больших геномов хвойных растений.

Все поставленные задачи выполнены успешно в результате выполнения настоящей дипломной работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Несговорова, Г. П. Биоинформатика: пути развития и перспективы / Г. П. Несговорова // Информатика в науке и образовании : сб. науч. тр. / Новосибирск. Ин-т систем информатики им. А. П. Ершова. – Новосибирск, 2012. – С. 70–90.

2. Кирьянов, К. Г. Почему биологические алфавиты имеют 4 и 20 букв ? / К. Г. Кирьянов, О. Л. Лебедев // Биофизика. – 1995. – № 40. – С. 536–538 с.

3. Сетубал, Ж. Введение в вычислительную молекулярную биологию / Ж. Сетубал, Ж. Мейданис. – Москва-Ижевск: НИЦ «Регулярная и хаотическая динамика», Институт компьютерных исследований, 2007. – 420 с.

4. Mount, D. W. Bioinformatics: Sequence and Genome Analysis / D. W. Mount. – Cold Spring Harbor, NY : Cold Spring Harbor Laboratory Press, 2004. – 547 p.

5. Illumina, Inc. [сайт]. – Режим доступа: http://www.illumina.com

6. Глик, Б. Молекулярная биотехнология. Принципы и применение / Б. Глик, Дж. Пастернак – Москва: Мир, 2002. – 589 с.

7. Александров, А. В. Метод сборки контигов геномных последовательностей на основе совместного применения графов де Брюина и графов перекрытий / А. В. Александров, С. В. Казаков, С. В. Мельников, А. А. Сергушичев // Научно-технический вестник информационных технологий, механики и оптики. – 2012. – No 6. – С. 93–98.

8. The National Center for Biotechnology Information, NCBI [сайт]. – Режим доступа: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC310812.html.

9. Левенштейн, В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов / В. И. Левенштейн // Доклады Академий Наук СССР. – 1965. – С. 845–848.

10. Wagner, R. A. The string-to-string correction problem / R. A. Wagner, M. J. Fischer // Journal of the ACM. – 1974. – № 21. – P. 168–173.

11. Smith, T. Identification of common molecular subsquenses / T. Smith, M. Waterman // Journal Molecular Biology. – 1981. – № 147. – P. 195–197.

12. Боресков, А. В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA : учеб. пособие / А. В. Боресков. – Москва : Издательство Московского университета, 2012. – 336 с.

13. Ершов, Ю. Л. Теория нумераций / Ю. Л. Ершов. – Москва, 1977. – 416 с.

14. Линьков, В. М. Нумерационные методы проектирования систем управления данными : Монография / В. М. Линьков. – Пенза : Издательство Пензенского государственного технического университета, 1994. – 156 с.

15. A performance study of general-purpose pplications on graphics processors using cuda. [сайт]. – Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/summary?oi=10.1.1.143.4849

16. Боресков, А. В. Основы работы с технологией CUDA : учебное пособие / А. В. Боресков, А. А. Харламов. – Москва : ДМК Пресс, 2010. – 230 с.

17. Гергель, В. П. Современные языки и технологии параллельного программирования : учебник для студентов вузов / В. П. Гергель. – Москва : Издательство Московского университета, 2012. – 406 с.

18. Касьянов, В. Н. Графы в программировании: обработка, визуализация, применение : монография / В. Н. Касьянов, В. А. Евстигнеев. – Санкт-Петербург : БХВ-Петербург, 2003. – 1104 с.

19. Практикум по методам параллельных вычислений : учебник для студентов вузов / А. В. Старченко [и др.] ; ред. А. В. Старченко. – Москва : Изд-во МГУ, 2010. – 199 с.

20. Волков, Р. GPGPU-технологии – сравнительный анализ и краткое описание достоинств и недостатков / Р. Волков, И. Починок // Мир компьютерной автоматизации. – 2014. – № 1. – С.34-41.

21. Langmead, B. Fast gapped-read alignment with Bowtie 2 / B. Langmead, S. Salzberg // Nature Methods. – 2012. – № 9. – P. 357–359.

22. Сандерс, Дж. Технология CUDA в примерах: введение в программирование графических процессоров / Дж. Сандерс, Э. Кэндрот; перевод с англ. А. А. Слинкина, научные редактор А. В. Боресков. – Москва : ДМК Пресс, 2011. – 232 с.

ПРИЛОЖЕНИЕ А

Листинг А.1 – Функция чтения входных данных

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda_runtime.h>
#include "dna_sequence.h"
#include "dna_alignment.h"
#include "config.h"

static cudaDeviceProp prop;

static cudaError_t gpu_init()
{
        cudaError_t err = cudaSuccess;
        if ((err = cudaSetDevice(0)) != cudaSuccess)
                return err;
        cudaGetDeviceProperties(&prop, 0);
        return cudaSuccess;
}

static void gpu_Properties()
{
        printf("ASCII string identifying device: %s\n"
                        "Global memory available on device in bytes: %u\n"
                        "Shared memory available per block in bytes: %u\n"
                        "32-bit registers available per block: %d\n"
                        "Warp size in threads: %d\n"
                        "Maximum pitch in bytes allowed by memory copies: %u\n"
                        "Maximum number of threads per block: %d\n"
                        "Maximum size of each dimension of a block: { %d, %d, %d
}\n"
                        "Maximum size of each dimension of a grid: { %d, %d, %d
}\n"
                        "Clock frequency in kilohertz: %d\n"
                        "Constant memory available on device in bytes: %u\n"
                        "Major compute capability: %d\n"
                        "Minor compute capability: %d\n"
                        "Number of multiprocessors on device: %d\n"
                        "Peak memory clock frequency in kilohertz: %d\n"
                        "Global memory bus width in bits: %d\n"
                        "Size of L2 cache in bytes: %d\n"
                        "Maximum resident threads per multiprocessor: %d\n",
                        prop.name, prop.totalGlobalMem, prop.sharedMemPerBlock,
prop.regsPerBlock,
                        prop.warpSize, prop.memPitch, prop.maxThreadsPerBlock,
                        prop.maxThreadsDim[0], prop.maxThreadsDim[1],
prop.maxThreadsDim[2],
                        prop.maxGridSize[0], prop.maxGridSize[1],
prop.maxGridSize[2],
                        prop.clockRate, prop.totalConstMem, prop.major,
prop.minor,
                        prop.multiProcessorCount, prop.memoryClockRate,
prop.memoryBusWidth,
                        prop.l2CacheSize, prop.maxThreadsPerMultiProcessor);
}

int main(int argc, char **argv)
{
```

```
        int result;
        cudaError_t status = gpu_init();
        if (status != cudaSuccess) {
                printf("%s\n", cudaGetErrorString(status));
                return status;
        }
        gpu_Properties();

        printf("\nDNA alignment time tests\n");
        for (unsigned threads_per_block = prop.warpSize;
                threads_per_block <= prop.maxThreadsPerBlock; threads_per_block
<<= 1) {
                        printf("\nthreads per block: %d\n\n", threads_per_block);

                for (unsigned i = 1; i <= 10; i++) {
                        char *s1 = (char*) malloc(i * 1000 + 1);
                        char *s2 = (char*) malloc(i * 1000 + 1);

                        memcpy(s1, Y13113, i * 1000);
                        s1[i * 1000] = '\0';
                        memcpy(s2, Y16067, i * 1000);
                        s2[i * 1000] = '\0';

                        printf("bp: %u\n", i* 1000);

                        result = dna_global_alignment_on_gpu(threads_per_block,
s1, s2, 0, 0);
                        printf("dna_global_alignment_on_gpu: %d\n", result);

                        result =
dna_global_alignment_on_gpu_opt(threads_per_block, s1, s2, 0, 0);
                        printf("dna_global_alignment_on_gpu_opt: %d\n", result);

                        free(s1);
                        free(s2);
                }
        }
        return 0;
}
```

Листинг А.2 – Функция вызова ядра построения матрицы сравнения на GPU

```
#include "dna_alignment.h"
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <math_functions.h>
#include <string.h>
#include <stdlib.h>
#include "config.h"
#include <stdio.h>

static __host__ __device__ int s(char a, char b)
{
	return (a == b) ? 1 : -1;
}

__global__ void fill_matix_kernel(const char *seq1, const  char *seq2, int *f,
int step, int str_count)
{
	extern __shared__ int f01 [];
	int *f11 = &f01[blockDim.x];
	int m = gridDim.x * blockDim.x;
	int j = blockIdx.x * blockDim.x + threadIdx.x;

	int i_start = (step - blockIdx.x) * blockDim.x;
	int i_end = i_start + blockDim.x;

	char c = seq2[j];

	if ((i_start >= 0) && (i_start < str_count)) {
		int f_;
		int f10 =	(i_start > 0) ? f[(i_start - 1) * m + j] : -(j +
1) * D;
		f01[threadIdx.x] = ((blockIdx.x > 0) && (i_start + threadIdx.x <
str_count))
			? f[(i_start + threadIdx.x) * m +  blockIdx.x *
blockDim.x - 1]
			: -(i_start + threadIdx.x + 1) * D;
		f11[threadIdx.x] =	((i_start > 0) && (j > 0)) ? f[(i_start -
1) * m + (j - 1)] :
								(j > 0) ? -j * D : -i_start
* D;
		__syncthreads();
		int i = i_start - threadIdx.x;
		for (int step = 0; step < (2 * blockDim.x - 1); i++, step++) {
			if ((i >= i_start) && (i < i_end)) {
				f_ = max(max(
									f01[i - i_start] -
D,
									f11[threadIdx.x] +
s(seq1[i], c)),
									f10 - D);
				f[i * m + j] = f_;
				f10 = f_;
			}
			__syncthreads();
			if ((i >= i_start) && (i < i_end))
				f11[threadIdx.x] = f01[i - i_start];
			__syncthreads();
			if ((i >= i_start) && (i < i_end))
				f01[i - i_start] = f_;
```

```
                        __syncthreads();
                }
        }
}

static void find_alignment(const char *seq1, int seq1_len,
        const char *seq2, int seq2_len,
        char *seq1_out, char *seq2_out, int *f, int col_count)
{
        int n = seq1_len;
        int m = seq2_len;

        for (int i = n - 1, j = m - 1, pos = n + m - 1; (i > 0) && (j > 0); pos-
-) {
                int f10 =       (i > 0) ? f[(i - 1) * col_count + (j)] : -(j + 1)
* D;
                int f11 =       ((i > 0) && (j > 0)) ? f[(i - 1) * col_count + (j
- 1)] :
                                                (j > 0) ? -j * D : -i * D;

                if(f[i * col_count + j] == f10 - D) {
                        seq1_out[pos] = seq1[i--];
                        seq2_out[pos] = '-';
                }
                else if(f[i * col_count + j] == f11 + s(seq1[i], seq2[j])) {
                        seq1_out[pos] = seq1[i--];
                        seq2_out[pos] = seq2[j--];
                }
                else {

                        seq1_out[pos] = '-';
                        seq2_out[pos] = seq2[j--];
                }
        }
}

int dna_global_alignment_on_gpu(unsigned threads_per_block, const char *seq1,
const char *seq2, char *seq1_out, char *seq2_out)
{
        cudaEvent_t start, end;
        int n = strlen(seq1);
        int m = strlen(seq2);

        char *gpu_seq1;
        char *gpu_seq2;
        int *gpu_f;

        int block_count = (m + threads_per_block) / threads_per_block;
        int col_count = block_count * threads_per_block;

        int sblock_count = (n + threads_per_block) / threads_per_block;
        int str_count = sblock_count * threads_per_block;

        int step_count = block_count + sblock_count - 1;

        CUDA_ASSERT(cudaMalloc(&gpu_seq1, str_count));
        CUDA_ASSERT(cudaMemcpy(gpu_seq1, seq1, n, cudaMemcpyHostToDevice));
        CUDA_ASSERT(cudaMalloc(&gpu_seq2, col_count));
        CUDA_ASSERT(cudaMemcpy(gpu_seq2, seq2, m, cudaMemcpyHostToDevice));

        CUDA_ASSERT(cudaMalloc(&gpu_f, str_count * col_count * sizeof(int)));

        cudaEventCreate(&start);
        cudaEventCreate(&end);
```

```
        cudaEventRecord(start);
        for (int step = 0; step < step_count; step++) {
                fill_matix_kernel<<<block_count, threads_per_block,
threads_per_block * 2 * sizeof(int)>>>(gpu_seq1, gpu_seq2, gpu_f, step,
str_count);
                cudaThreadSynchronize();
        }
        cudaEventRecord(end);
        cudaEventSynchronize(end);

        float t;
        cudaEventElapsedTime(&t, start, end);
        printf("dna_alignment_on_gpu: time %f ms; mem %u bytes\n", t, str_count
* col_count * sizeof(int));

        cudaFree(gpu_seq1);
        cudaFree(gpu_seq2);

        int *f = (int*) malloc(n * col_count * sizeof(int));
        cudaMemcpy(f, gpu_f, n * col_count * sizeof(int),
cudaMemcpyDeviceToHost);
        cudaFree(gpu_f);

        if ((seq1_out != 0) && (seq2_out != 0))
                find_alignment(seq1, n, seq2, m, seq1_out, seq2_out, f,
col_count);

        int res = f[col_count * (n - 1) + m - 1];
        free(f);
        return res;
}
```

СОДЕРЖАНИЕ