ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

Факультет *Информационных Технологий и Программирования*

Направление (специальность) *Прикладная математика и информатика*

Квалификация (степень) *Магистр прикладной математики и информатики*

Кафедра *Компьютерных технологий* Группа *6539*

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Разработка алгоритма неточного поиска чтений в геноме с применением вычислений на видеокартах

Автор магистерской диссертации *Кириллова А.А.*

Научный руководитель *Шалыто А.А.*

Руководитель магистерской программы *Васильев В. Н.*

К защите допустить

Заведующий кафедрой *Васильев В. Н.*

“____” _____ 2015 г.

Санкт-Петербург, 2015 г.

Содержание

Введение

Биоинформатика – быстро развивающаяся область науки, стоящая на стыке информатики и биологии. Сборка генома является одной из важнейших задач биоинформатики, так как получение и последующий анализ генома человека и других живых существ может позволить реализовать персонифицированную медицину, выявлять наследственные заболевания и многое другое.

Появление задачи сборки генома обусловлено тем, что на сегодняшний день невозможно получить целый геном. Так, различные секвенаторы на выходе дают набор чтений довольно небольшой длины, которые несколько раз покрывают исследуемый геном. К тому же, полученные чтения содержат в себе ошибки, доля которых определяется характеристикам секвенатора. Таким образом, чтобы собрать геном, необходимо понять, как же склеить эти чтения в него, зная, что они перекрываются, но могут содержать ошибки. Эта задача может быть решена несколькими методами. Первый подход применяется, если есть уже известный геном очень похожего организма. В этом случае пытаются выровнять полученные чтения на него. Второй подход связан с поиском перекрытий между чтениями и последующим их анализом с помощью специальных алгоритмов.

На сегодняшний день существует множество секвенаторов, которые отличаются по различным параметрам, такими как цена, скорость секвенирования, длины чтений и доля ошибок. В основном все секвенаторы дают на выходе небольшие чтения, порядка нескольких сотен нуклеотидов. В 2013 году компания Pacific Biosciences выпустила секвенатор под названием «PacBio RS II»[1]. Этот секвенатор отличается тем, что дает на выходе, с одной стороны, длинные чтения (порядка 5-10 тысяч нуклеотидов), но, с другой стороны, они содержат около 15% ошибок. Для таких данных старые методы поиска перекрытий перестали работать эффективно. Поэтому, требуется разработка новых методов.

В данной работе разрабатывается такой метод, с использованием вычислений на GPU.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Биоинформатика

Биоинформатика начала выделяться в отдельную область науки в конце 60-х – начале 70-х годов прошлого века, когда ЭВМ стали активно применяться в биологии. К этому времени накопилось огромное количество экспериментальных данных по биологии, требующих осмысления и обработки. На тот момент, полученных новых данных было намного больше, чем человек мог обработать и выделить из них некоторые закономерности. Возникла необходимость в хранении такого огромного количества данных, и было понятно, что без специальных программ не обойтись. Биоинформатика, как и многие другие современные науки, развивается на стыке нескольких наук: математики, биологии, генетики и компьютерных технологий. Основная задача биоинформатики заключается в разработке программного обеспечения для анализа и хранения большого количества полученных биологических данных.

Биоинформатика ведет исследования в нескольких областях. Можно выделить следующие направления:

1) Геномная биоинформатика. Эта область биоинформатики, которая занимается тем, что применяет математические методы для исследования и анализа, сборки различных геномов.

2) Структурная биоинформатика. Эта область биоинформатики занимается тем, что изучает структуры, предсказывает их по имеющейся последовательности аминокислот.

3) Разработка программного обеспечения для управления биологическими системами информационной сложности.

1.2 Геном

Одной из основных областью исследования биоинформатики является анализ генетических последовательностей.

Два основополагающих свойства живых организмов – наследственность и изменчивость, закодированы в молекулы дезоксирибонуклеиновой кислоты (ДНК). ДНК определяет всю последующую программу развития организма. Почти все геномы построены из ДНК (исключение составляют некоторые вирусы).

ДНК – макромолекула, которая обеспечивает хранение, передачу через поколения, а так же реализацию генетической программы, для правильного функционирования живых организмов. ДНК содержит в себе информацию о структуре белков и различных видов РНК. В большинстве случаев ДНК состоит из двух полимерных цепочек, состоящих из соединенных в последовательность нуклеотидов. Каждый из них состоит из некоторого азотистого основания, фосфатной группы и дезоксирибозы. Азотистые основания, встречающиеся в ДНК, называются адеин (A), гуанин (G), тимин (T) и цитозин (C).

В молекулярной биологии многие объекты представляются в виде последовательности символов. Длина этих последовательностей может достигать миллиарды символов.

Расшифровка ДНК – решение многих проблем человеческого организма. Это помогает проводить профилактику врожденных болезней или заболеваний, к которым человек предрасположен изначально. При наличии подобной информации можно гораздо быстрее поставить диагноз и назначить лечение, а также спрогнозировать исход болезни с огромной точностью.

ДНК исследуется уже несколько десятилетий. Еще в середине XX века ученые установили, что отдельные участки молекул дезоксирибонуклеиновой кислоты – это и есть гены организма. Также в это

время стало известно, что между химической структурой участков молекул ДНК и самими молекулами белков существует взаимосвязь, согласно которой порядок расположения ДНК в белках соответствует порядку структурных единиц ДНК (нуклеотидов) в гене. Такое революционное открытие позволило ученым более детально исследовать загадки наследственности. Несмотря на то, что расшифровать ДНК сегодня возможно, на это уходит очень много времени из-за того, что в одной лишь молекуле скрыто огромное количество закодированной информации.

1.3 Методы секвенирования

Длина молекулы ДНК может варьироваться от нескольких миллионов, до нескольких миллиардов нуклеотидов. Секвенирование (*sequencing*) – это процесс получения последовательности нуклеотидов в молекуле ДНК. В настоящий момент существует множество различных технологий секвенирования, однако нет ни одного метода, который бы позволил прочитать ДНК длинной молекулы целиком. Все они работают по похожей методологии. Многократно клонируется длинная молекула ДНК, а потом «разрезается» в случайных местах. Так получается большое число небольших участков ДНК, каждый из которых потом читается по отдельности.

Таким образом, на выходе из секвенатора имеется набор чтений, которые являются кусками исходного генома. При этом чтения могу содержать ошибки вставки, удаления и замены. В биоинформатике ставится задача собрать исходный геном по данным, полученным на выходе секвенатора. На сегодняшний день существуют различные секвенаторы, отличающиеся по многим параметрам. Таким, как стоимость, время работы, длины получаемых чтений, процент ошибок в чтениях.

Существует несколько основных методологий секвенирования.

1.3.1 Метод Сэнгера (метод обрыва цепи)

Метод Сэнгера — метод секвенирования, известный так же под названием метода обрыва цепи[2]. Фредерик Сэнгер предложил этот метод впервые в 1977 году, за что в 1980 году был удостоен Нобелевской премии по химии. В настоящее время этот метод широко применяется, для определения нуклеотидной последовательности ДНК. Метод сочетает в себе простоту, точность и эффективность.

Есть различные вариации этого метода, но можно описать классический его вариант.

ДНК, как кислота, в водном растворе дает анион. Если же погрузить катод и анод в этот раствор, тогда фрагменты ДНК направятся к плюсу. Если заменить водный раствор на гель на водной основе, тогда скорость этих фрагментов будет обратно пропорциональной длине фрагментов. Таким образом, маленькие ниточки быстро заскользят через гелевую сетку, а длинные будут часто цепляться и ползти медленней. В итоге, те фрагменты ДНК, которые были в одной капле смеси изначально, выровняются по росту. Этот метод называется электрофорезом, он очень эффективен, когда нужно рассортировать куски ДНК по длине. Секвенирование по Сэнгеру основывается на этом методе.

Назначенный в прочтению раствор фрагмента выступает в качестве матрицы для реакции полимеризации ДНК. Реакцию с одной и той же матрицей проводят в четырех различных пробирках. Так же в каждую пробирку добавляют обрывающие синтез цепочки нуклеотиды. В каждую какой-нибудь один: A, T, G или C. Количество реагентов подбирают таким образом, что в каждой из пробирок получаются все фрагменты, заканчивающиеся на соответствующий нуклеотид. Далее, применяется метод высоковольтного электрофореза. В лунки на гели добавляют по капле из каждой пробирки и пропускают через них ток. Спустя какое-то время новые кусочки ДНК разделяются по размеру и располагаются узкими полосочками.

Каждая такая полосочка соответствует какому-либо определенному типу нуклеотидов. На одной из таких дорожек будут располагаться все буквы A, на второй – T, потом G и C. Получив такие дорожки, можно будет определить последовательность нуклеотидов в зависимости от расстояния до старта.

1.3.2 Метод дробовика

Обычные методы секвенирования применимы только к коротким чтениям, длина которых варьируется от 100 нуклеотидов до 1000. Метод дробовика[3] же позволяет секвенировать длинные участки ДНК. Идея состоит в том, чтобы длинные участки ДНК можно случайным образом разделить на небольшие фрагменты, к которым применимы другие методы секвенирования, к примеру, тот же метод Сэнгера. После секвенирования небольших участков, полученные фрагменты нужно собрать в соответствующие более длинные чтения. Это можно сделать при помощи специального программного обеспечения, которое ищет перекрытия между полученными участками. Таким образом можно получить довольно большой участок ДНК.

1.3.3 Методы секвенирования нового поколения

Методы нового поколения – общее название современных методов секвенирования[4]. Чтобы снизить стоимость процедуры секвенирования и увеличить ее производительность, в новых методах определения нуклеотидной последовательности ДНК проводится секвенирование миллионов ее фрагментов одновременно. Отличительной особенностью методов секвенирования ДНК нового поколения является их направленность

на получение нуклеотидных последовательностей целых геномов различных организмов, включая человека.

Секвенирование происходит в несколько этапов. Первый этап представляет собой фрагментацию длинных молекул ДНК ультразвуком или ферментативной рестрикцией до размера 50–1000 п.н. Далее следует стадия обработки концов ДНК-фрагментов, в результате которой достраиваются или удаляются выступающие концы фрагментов, а затем адаптерные олигонуклеотиды лигируются. Затем полученная библиотека для увеличения представительности каждого фрагмента ДНК амплифицируется. Последний шаг секвенирования – создание так называемых молекулярных колоний путем клональной амплификации каждого фрагмента библиотеки на твердой поверхности (подложка или микросфера). Подготовленная таким образом библиотека ДНК помещается в секвенатор, где чередуются стадии подачи реактивов для самлшл секвенирования, со стадиями сканирования поверхности и регистрации сигналов.

1.4 Сборка генома

Процесс сборки генома заключается в объединении большого числа чтений, полученных на выходе секвенатора, в одну или несколько длинных последовательностей, для того, что восстановить исходную последовательность ДНК, которая была секвенирована.

Эта задача является сложной вычислительной задачей, так как работать приходится с огромным количеством данных, в которых, к тому же, содержатся ошибки. Также задача осложняется тем, что различные геномы часто содержать большое число одинаковых повторяющихся последовательностей, которые называются геномными повторами. Длина этих повторов может достигать нескольких тысяч нуклеотидов, а встречаться они могу в тысяче различных мест.

Существует несколько основных подходов для сборки генома, на которых основываются все остальные.

Первый из них основывается на поиске перекрытий *Overlap-Layout-Consensus* (является эффективным для длинных чтений), второй основывается на графах *De Bruijn* (является эффективеным для коротких чтений).

1.4.1 Overlap-Layout-Consensus

В данном подходе, на выходе из секвенатора мы имеет миллионы маленьких фрагментов, длинной до 1000 нуклеотидов. Методы, использующие этот подход, пытаются составить исходной ДНК, путем поиска пересечений между полученными чтениями[5]. Перекрытия находятся, далее по ним происходит объединение и исправление ошибок в полученной строке. В процессе сборке данные шаги могут повторяться несколько раз.

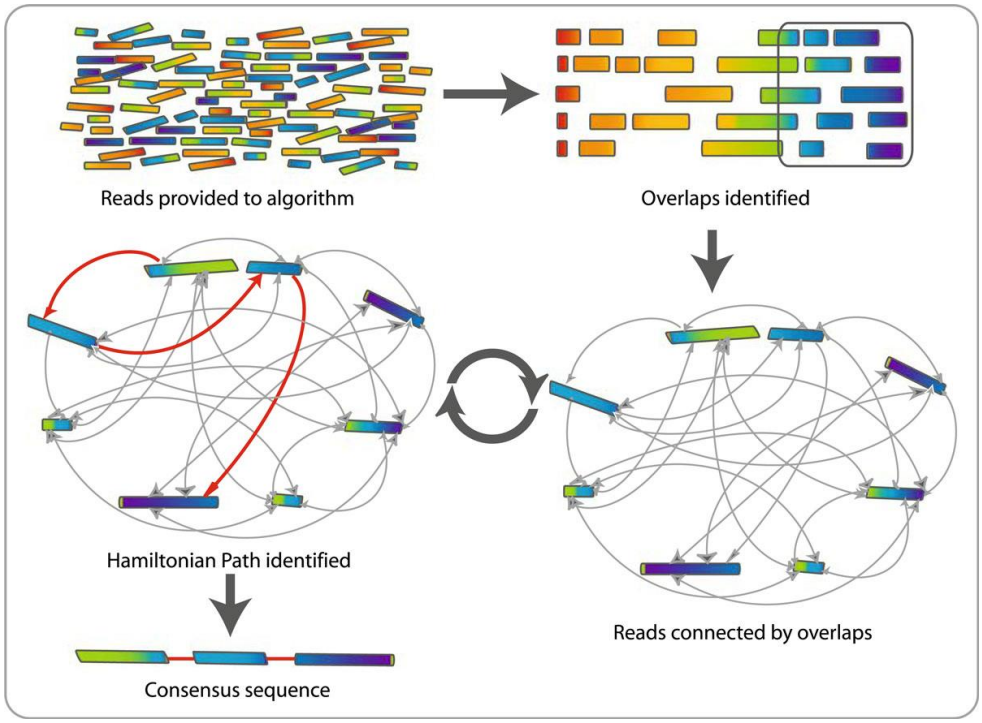

Рисунок 1 – Схематичное описание работы алгоритма[6].

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

1.4.2 Графы де Брюина

Данный подход может быть наиболее эффективно применен для набора небольших чтений – 300 нуклеотидов. Здесь применяются так называемые графы де Брюина[7]. Вершинами этого графа являются *k*-меры, из вершины u ведет ребро в вершину *v* тогда и только тогда, когда суффикс длины *(k-1)* вершины *u* является префиксом строки вершины *v*. Подграф, который используется сборщиками, выбирается на основании *k*-меров, содержащихся в чтениях, в качестве вершин и ребер между вершинам, которые показывают, что соответствующие *k*-меры идут друг за другом.

Для сборки генома в таком случаю используются различные алгоритмы на графах.

1.5 Картрирование чтений

Как было сказано ранее, на выходе секвенатора мы получаем не целый геном, а набор чтений небольшой длины (порядка нескольких сотен или тысяч нуклеотидов), которые в совокупности составляют оригинальный геном. Необходимо так же учитывать, что в полученных чтениях будут содержаться ошибки.

На сегодняшний момент уже известны геномы многих организмов, и многие компании имеют базу этих геномов. Когда исследуется некоторый новый геном, для сборки чтений можно использовать так называемый «референсный геном». Это уже известный геном организма близкого к исследуемому. Как показывает практика, подобные геномы сильно похожи, и при сборке можно пытать подобрать места для чтений, откуда они могли

быть взяты с наибольшей вероятностью. Этот метод анализа результатов секвенирования называется картрированием чтений.

Существуют несколько общих подходов к сборке генома. Они могут применяться с разной степенью эффективности с учетом различных параметров, которыми могут выступать длины выравниваемых чтений, процент ошибок, виды ошибок (многие поддерживают только ошибки вставки/удаления). Можно выделить два основных подхода, на которых базируются все остальные.

1.5.1 Основы подхода с использованием хеширования

Так как искать расстояние Левенштейна с каждой позиции в референсном геноме слишком затратно, было бы разумно ввести некоторую метрику, по которой можно было бы быстро предсказать степень похожести рассматриваемых подстрок. И производить выравнивание уже с отобранных позиций.

В данном подходе для поиска кандидатов вводится хеш-функция, трансформирующая строку в некоторый ключ, который и используется для быстрого поиска. Но, так как в чтениях содержатся ошибки, к тому же длина их может быть достаточно большой, а это затруднит вычисление значении хеш-функции, то в методе предлагается разделить чтения на более короткие участки, которые встречаются гораздо чаще.

Так, если чтение разделить, скажем, на четыре части и попробовать их наложить на референсный геном, то, если в чтении произошла одна ошибка, то всё равно три части будут полностью наложены на соответствующий участок генома. Аналогично, если произойдет две ошибки, то две части чтения будут полностью совпадать.

Таким образом для каждого чтения будет получен набор мест в геноме, куда оно может быть картрировано. И далее производится уже обычное

выравнивание двух подстрок, для определения, какая из позиций в геноме подходит наиболее лучшим образом.

Похожим образом работает, к примеру, SOAP[8].

Есть также модификация данного подхода. Чтение разделяют не на некоторые непересекающиеся части, а рассматривают все его *k*-меры. Это увеличивает чувствительность, как и увеличивает затраты времени. Подобным образом работают SHRiMP2[9].

Большую часть времени, однако, даже при таком подходе, обычно алгоритмы тратят не на поиск мест, откуда могут начинаться вхождения, а на саму проверку их окружения. Чаще всего для проверки окружения используется алгоритм Нидлмана — Вунша или различные его модификации. В некоторых программах, в целях оптимизации, сначала быстро просчитывают расстояние Эйлера – общее количество одинаковых букв между чтениями. Это помогает отсеять неподходящих кандидатов.

Также, данный подход может быть модифицирован для поиска перекрытий между чтениями. В таком случае можно использовать *k*-меры и пытаться найти подстроки максимальной длины между чтениями, которые больше всего похожи друг на друга. Это можно сделать, если посчитать для каждого из пары чтения все его *k*-меры и определить общие *k*-меры между полученными множествами. Так можно предположить, где чтения могут перекрываться.

1.5.2 Основы подхода с использованием суффиксов

При большом количестве повторов алгоритм на основе хеширования начинает плохо работать, так как получается очень большое количество мест в геному, куда могут быть наложены чтения, и проверка с каждого из них занимает много времени.

Для поиска вхождения чтений был придуман подход, базирующийся на использовании суффиксных деревьев и массивов. Эффективность его

работы не зависит от количества повторов, так как одинаковые участки просто «схлопываются» в суффиксном дереве. При условии отсутствия ошибок, подход работает очень быстро.

Суть алгоритма можно описать следующим образом. Референсный геном преобразуется по Барроузу-Уилеру. Далее, каждое чтение выравнивается каждый раз по одному нуклеотиду по преобразованному геному, таким образом сужая количество мест, куда оно может быть выравнено. Заметим, что преобразование по Барроузу-Уиллера позволяет хранить только верхнюю и нижнюю границы и быстро на каждом шаге находить следующий интервал, соответствующий рассматриваемому нуклеотиду. Для поддержки ошибок замены, добавляется возможность возврата к предыдущей позиции и выбора другого, не соответствующего рассматриваемому, элемента. Но, если количество ошибок велико, а так же присутствуют ошибки замены и вставки, то алгоритм начинает сильно терять в производительности, так как появляется очень большое ветвление.

Схожим образом работают BWA[10], Bowtie[11].

1.6 Bitap

Двоичный алгоритм поиска подстроки. Этот алгоритм[12] так же известен, как *Shift-Or* или *Baeza-Yates-Gonnet*. Существует несколько модификаций исходного алгоритма, к примеру, *Wu-Manber*.

Двоичный алгоритм поиска подстроки и различные его модификации довольно часто используются на практике, при поиске без индексации. Так, некоторая его модификация используется в unix-утилите agrep, которая выполняет те же функции, что и утилите grep, но ещё и поддерживает ошибки в запросах.

Идея алгоритма довольно простая, основной выигрыш здесь достигается некоторой оптимизации. А именно, алгоритм учитывает, что битовый сдвиг и побитовое ИЛИ являются атомарными операторами. И,

работая с битовыми данными, алгоритм позволяет производить одновременно 32-64 сравнения.

Существует множество его модификаций. Предполагается, что впервые алгоритм был разработан в 1964 году Балинтом Демелькином. В том варианте это был просто алгоритм поиска подстроки в строке, без учета ошибок. В 1992 году граждане Ricardo Baeza-Yates и Gaston Gonnet предложили некоторую модификацию алгоритма, которая могла находить ошибки замены и по сути вычисляла расстояние Хемминга. Несколько позже Sun Wu и Udi Manber предложили ещё одну модификацию алгоритма, которая уже вычисляла расстояние Левенштейна и поддерживала ошибки вставки и удаления. Именно на основе этой работы была написана первая версия утилиты agrep.

Алгоритм:

Введем операцию, которую назовем *Bitshift:*

$$R = \begin{matrix}0\\0\\0\\0\end{matrix} \xrightarrow{Bitshift} \begin{matrix}1\\0\\0\\0\end{matrix} \xrightarrow{Bitshift} \begin{matrix}1\\1\\0\\0\end{matrix} \xrightarrow{Bitshift} \begin{matrix}1\\1\\1\\0\end{matrix}$$

Определим следующие переменные:

- k — количество допускаемых ошибок;
- j — индекс символа в рассматриваемой строке;
- s_x — битовая маска символа *x*. Такая маска строится по запросу для каждого из символов рассматриваемого алфавита. Так, в позиции *i* в маске находится единичный бит, если в запросе на позиции *i* находится символ *x*;
- R^k - результирующий вектор, когда допускается наличие k ошибок. Совпадение или несовпадение запросу определяется последним битом вектора.

Тогда легко представить, что точное вхождение строки можно будет представить следующим образом:

$R_{j+1}^{k} = Bitshift(R_{j}^{k}) \wedge s_{x}$, где k=0

Дополнительную поддержку вставки, удаления и замены символа можно ввести следующим образом:

- Вставка: $R^{k}_{\text{вставки } j+1} = R_{j}^{k-1}$
- Удаление: $R^{k}_{\text{удаления } j+1} = Bitshift(R_{j+1}^{k-1})$
- Замена: $R^{k}_{\text{замены } j+1} = Bitshift(R_{j}^{k-1})$

Результирующее значение:

$R_{j+1}^{k} = (Bitshift(R_{j}^{k}) \wedge s_{x}) \vee R^{k}_{\text{вставки } j+1} \vee R^{k}_{\text{удаления } j+1} \vee R^{k}_{\text{замены } j+1}$

Асимптотическое время работы алгоритма $O(kn)$. Однако, на практике он работает значительно быстрее линейных методов. В основном, это достигается за счет битового параллелизма вычислений. Если брать 64-битную систему, то за одну операцию производится сравнение сразу на 64 битами. Особенно быстро он работает, если длины подстрок не превышают длину машинного слова. А так же на небольших алфавитах.

1.7 GPGPU и CUDA

GPGPU – техника использования графического процессора видеокарты, который обычно имеет дело с вычислениями только для компьютерной графики, чтобы выполнять некоторые общие вычисления, не связанные с графикой. Обычно такие вычисления проводит центральный процессор. GPU способен во много раз ускорить работу приложений.

CUDA[13] – это архитектура параллельных вычислений представленная компанией Nvidia[14]. Эта программно-аппаратная

архитектура позволяет увеличить вычислительную производительность, и приспосабливает использование графических процессоров для математических вычислений.

CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, которые могут выполняться на графических процессорах Nvidia. Архитектура CUDA позволяет разработчику управлять памятью графического ускорителя.

CPU состоит из нескольких мощных ядер, оптимизированных для последовательной обработки данных, в то время как GPU состоит из тысячи более мелких ядер, заточенных для обработки нескольких задач одновременно. За производительность GPU платит оптимизацией. Только программы специального вида можно исполнять эффективно.

В основном это те задачи, в которых требуется обработать большой объем данных одинаковым образом и при этом, эти данные не зависят между собой.

Логическая организация программ:

- Программу исполняют миллионы независимых потоков. Каждый поток обрабатывает часть входных данных.
- Потоки объединяются в блоки. В каждом блоке есть некоторая общая память, потоки в блоке всегда синхронизированны. Блок исполняется на одном мультипроцессоре.
- Блоки объединяются в сетку.
- Каждый поток знает свое положение в сетке.

В графических процессорах есть шесть видов памяти:

- Глобальная память. Доступна всем. Через неё идет обмен данных с CPU. Имеет довольно большой объем (6Гб), но очень медленная по сравнению с остальными.

- Разделяемая память. Относится к быстрому типу памяти. Является общей для всех потоков на блок. Имеет довольно небольшой объем. Используется для минимизации обращений в глобальную память.
- Константная память. Быстрый тип памяти. Особенностью является возможность записи с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти.
- Регистровая память. Является самой быстрой из всех видов. Нет явных способов размещения в регистровой памяти. Компилятор всю работу берет на себя.
- Локальная память. Память доступная внутри потока. Довольно медленная. Используется при необходимости, когда не хватает регистровой.
- Текстурная память. Предназначена для работы с текстурами.

1.8 Постановка задачи

В этой работе ставится задача разработки алгоритма поиска перекрытий между парами длинных чтений с использование вычислений на GPU.

Общая формулировка задачи:

Есть множество длинных чтений. Для каждой пары чтений из этого множества требуется найти локальные перекрытия. Чтения перекрываются, если у них есть общие подстроки довольно большой длинны (параметр), с редакционным расстоянием, не превышающим порядка 15%(параметр) от длины подстроки.

Требуется реализовать придуманный алгоритм. Протестировать его и сравнить с аналогами.

1.9 Вывод по главе 1

В данной главе был произведен обзор предметной области. Были рассмотрены существующие технологии секвенирования, описана задача сборки генома и подходы к её решению.

Был дан краткий обзор техники GPGPU и архитектуры CUDA.

Была выполнена постановка задачи данной работы.

2 ПРЕДЛАГАЕМЫЙ ПОДХОД

В данной главе приводится описание предложенного алгоритма поиска перекрытий длинных чтений и его реализации.

2.1 Формальная постановка задачи

На выходе секвенатора получается большое множество длинных чтений. Обычно, для увеличения точности и качества сборки, делают 20-40 ыкратное покрытие генома чтениями, разбивая его с различных позиций. Потом среди всех полученных чтений ищут перекрытия.

Чтобы найти перекрытия, необходимо попарно сравнить между собой все чтения из полученного множества. По сути, нужно сравнить это множество с самим собой. Тогда можно переформулировать задачу в терминах двух блоков. Задача сведется к тому, что заданы два блока длинных чтений, и для каждой пары чтений, первое из которых содержится в первом блоке, а второе во втором блоке, требуется найти локальное перекрытие. Такая постановка задачи является более удобной и предпочтительной. Так как геномы многих организмов имеют очень большую длину, то для хранения полученного множества чтений и его обработки может не хватить памяти. Чтобы решить эту проблему, множество можно разбить на несколько более маленьких. Тогда можно будет попарно сравнить полученные блоки и объединять результаты.

Определим задачу следующим образом.

На вход дается два блока чтений: A и B. Длина чтения может варьироваться от 1000 до 25000 нуклеотидов, в среднем порядка 10000 нуклеотидов. Чтение задается как строка, где нуклеотид представляется одним из символов ‘A’, ‘T’, ’G’, ‘C’. Для каждого чтения $a \in A$ и $b \in B$

нужно найти подстроки длины, большей чем τ, расстояние Левенштейна между которыми не превышает $\varepsilon\%$ от длины подстроки.

2.2 Общее описание алгоритма

Как говорилось в первой главе, если мы хотим для двух чтений найти перекрытие, то выполнять выравнивание с каждой пары позиций слишком затратно. Если длина первого чтений n, а второго m, то было бы необходимо запускать алгоритм выравнивания с $O(nm)$ позиций. Поэтому, необходимо сначала придумать, как искать позиции, с которых следует начинать выравнивание, сформировать список кандидатов для выравнивания, который будет передан на следующий этап работы программы.

Таким образом, задача была разбита на несколько независимых этапов:

- Фильтрация. Поиск кандидатов для сравнения.
- Выравнивание. Неточное сравнение двух строк.
- Объединение результатов.

Первый этап реализовывался на основе метода поиска общих k-меров для пары чтений. Но к стандартному подходу были применены различные модификации, которые позволили сильно оптимизировать алгоритм, и к тому же перенести часть вычислений на GPU. Второй стандартный подход на основе суффиксных массивов в рамках поставленной задачи не может быть эффективно реализован, так как в отличие от стандартных 2-3 процентов ошибок секвенирования, в полученных чтениях их содержится около 15 процентов. А, так как в своем решении я хочу поддерживать ошибки вставки и удаления, то ветвление получилось бы очень большим, и подход был бы крайне не эффективен. Поэтому, было решено остановиться на первом подходе, модифицировать его под поставленную задачу и разработать алгоритм, переносящий все вычисления на GPU.

Для конкретного выравнивания был реализован алгоритм на основе известного алгоритма *Bitap*. Асимптотическое время работы этого алгоритма $O(nd)$, однако, на практике он работает на порядок быстрее линейных методов сравнения. На этапе фильтрации был сформирован список кандидатов, которые определяли подстроки, которые необходимо сравнить. Алгоритм фильтрации был разработан таким образом, что длины подстрок, которые нужно было сравнить, не превышали 64 символов. Таким образом, битовые маски строк, которые использует алгоритм *Bitap,* можно было представить в виде 64-битного числа. Благодаря этому за одну операцию, по сути, проводилось 64 сравнения, что позволило очень сильно уменьшить время работы.

Объединение результатов производилось на основе найденных общих *k*-меров и полученных результатов неточного сравнения подстрок.

2.3 Фильтрация

Целью этого этапа является сокращение числа кандидатов для сравнения. Необходимо найти позиции в чтениях, откуда могло бы начинаться выравнивание.

Краткое описание алгоритма:

1. Для каждого чтения из блока захешировать все его *k*-меры.
2. Сформировать список для каждого блока всех его *k*-меров с информацией о номере чтения, где он был найден, и позиции, с которой начинался.
3. Объединить два полученных списка по одинаковым *k*-мерам. Получится список, который будет содержать два номера чтений (первый номер – номер чтения блока *A*, второй номер – номер чтения блока *B*), а так же позиции в этих чтениях, с которых будет начинаться общий *k*-мер.

4. Для каждой пары чтений посчитать количество совпавших нуклеотидов (на основе знаний о совпавших *k*-мерах и учитывая смещение в позициях совпавших *k*-меров, оно должно быть примерно одинаковым).
5. Если количество одинаковых нуклеотидов больше, чем некоторая заданная константа алгоритма, тогда предполагается, что между заданными чтениями есть локальное выравнивание. В список кандидатов для сравнения добавляются все промежутки между найденными общими *k*-мерами для подсчета числа ошибок в них. Если длина промежутка превышает 64 символа, то промежуток дополнительно подразбивается.

Данный алгоритм был реализован за линейное время работы на CPU, а так же использовал линейный объем памяти. Такой оценки времени помогло добиться перенос части вычислений на GPU, а именно *RadixSort*, которая не раз использовалась в реализации алгоритма. Так же, реализованный алгоритм позволяет находить не только перекрытия, но охватывает больший круг решаемых задач. К примеру, находя локальные перекрытия, мы находим повторы.

2.3.1 Хеширование *k*-меров

Число *k* – параметр алгоритма (принимает значения в диапазоне 14-18), длина *k*-меров, которые будут хешироваться. От этого параметра сильно зависит точность и время работы алгоритма (анализ зависимости можно посмотреть в главе 3). Чем больше *k*, тем быстрее работает алгоритм и тем меньше точность. Если выбрать *k* небольшим, то количество совпавших *k*-меров будет большое, а значит, алгоритм будет работать дольше, но при этом шанс не пропустить перекрытия повышается, чем если бы взяли большее *k*.

Исходя из этого, мы можем ввести ограничение, что длина *k*-мера не должна превышать 30 символа. Так как каждый символ *k*-мера может

принимать всего одно из четырех значений, каждый из них мы можем однозначно закодировать 2 битами. И тогда, при использовании битового сдвига может представить k-мер как 64 целочисленное беззнаковое число, которое и будет являться хешем. Причем, идя по чтению a, нам не нужно пересчитывать каждый раз хеш, достаточно просто сделать битовый сдвиг, закодировать нужный символ и применить маску, чтобы удалить лишние символы:

$$KMerMask = (1l \ll 2 * K) - 1$$
$$H_i = (H_{i-1} \ll 2 \mid a_{i+K-1}) \,\&\, KMerMask$$

2.3.2 Формирование списка *k*-меров каждого блока

Одновременно с *k*-мером нам нужно закодировать сразу номер чтения и позицию, в котором он встречается. Это мы можем сделать, используя знание о том, что длина чтения не превышает 25 тысяч. Поэтому, чтобы закодировать позицию достаточно 15 бит. 17 бит достаточно, чтобы закодировать номер чтения, если учесть, что в среднем чтение состоит из 5-10 тысяч нуклеотидов. Таким образом, мы можем закодировать номер чтения и позицию как 32-значное число, отдав первые 17 бит под номер чтения, а оставшиеся 15 под позицию в нем. Так как номер чтения перекодировать не нужно при построении хеша для его *k*-меров, то он меняться не будет. А позиция будет каждый раз просто инкрементироваться на единицу, что будет соответствовать увеличению на единицу просто самого числа:

$$V_0 = (a \ll 15)$$
$$V_i = V_{i-1} + 1$$

Такое хеширование для чтения *a* будет работать за *O(|a|)*.

Применив это хеширование для всех чтений каждого из блоков, можно составить списки пар «ключ-значение»: L_A и L_B, где в качестве ключа будет

выступать хеш *k*-мера, а в качестве значения – закодированные номер чтения в блоке и его позиция. Составить эти списки можно за один проход по всем чтениям блока.

2.3.3 Получение списка общих *k*-меров

На предыдущем шаге были получены два списка L_A и L_B, содержащие пары «ключ-значение», где ключом является хеш *k*-мера, а значением – номер чтения и позиция начала вхождения *k*-мера в чтение. На данном этапе задача заключается в том, чтобы объединить всеми возможными способами значения списков L_A и L_B для элементов с одинаковыми ключами. Так как хочется сделать все это за линейное время на CPU, полный перебор вариантов и сравнения ключей не подходит. Был разработан подход, позволяющий этого избежать:

1. Отсортируем по ключу каждый из списков. Я использовала сортировку, встроенную в библиотеку для CUDA *thrust*[15], *radix sort*[16]. Это очень быстрая сортировка, разработанная в рамках проекта *back40computing*[17], выполняющаяся на GPU. Сортировка написана нескольких вариантах. Я использовала сортировку по паре «ключ-значение», где ключ является 64 битным числом, а значение – 32 битным.
2. Теперь легко можно объединить два списка на CPU за линейное время. Обычное слияние двух отсортированных массивов. Достаточно иметь два указателя на каждый из массивов и идти по ним параллельно, формирую новый список. При этом формировать список будем только из значений, конкретные *k*-меры уже не интересны, мы знаем, что они полностью совпадают, а информация о совпавших *k*-мерах нам уже не понадобится. В итоге получается список *L={(a,b,i,j):Kmer(a,i)=Kmer(b,i)}*.

2.3.4 Подсчет числа общих нуклеотидов

Рассмотрим для начала самый простой вариант алгоритма, который подсчитывает число общих *k*-меров между каждой парой чтений. Это облегчит понимание и покажет основную идею алгоритма. Далее, будет показано, как этот алгоритм можно улучшить и доработать под поставленную задачу, уменьшив время работы и увеличив точность.

Алгоритм:

1. Отсортируем полученный список по координатам *a, b*.
2. Подсчитываем количество совпавших *k*-меров для каждой пары чтений. С учетом того, что список отсортирован, это можно будет сделать за линейное время.

В таком виде алгоритм обладает рядом недостатков. Во-первых, необходимо учитывать случайные совпадения *k*-меров, которые могут вовсе не являться локальными перекрытиями. Во-вторых, не совсем правильно подсчитывать число общих *k*-меров и на основании этого делать вывод о возможном наличии локального перекрытия. Далее приведено, как решается каждая из этих проблем.

2.3.4.1 Разбиение по диагоналям

Необходимо учитывать, что *k*-меры могут совпадать случайно, а так же иногда встречаются повторы. И, даже, если между парой чтений есть большое число общих *k*-меров, не все они могут содержаться в одном перекрытии. Идея решения этой проблемы основывается на том, что если между парой чтений есть перекрытие, то относительное смещение для общих *k*-меров, содержащихся в нем остается примерно одинаковым. Если в чтении *a* *k*-мер из перекрытия начался с позиции *i*, а в чтении *b* с позиции *j*, то смещение будет равно *i*-*j*. Предполагается, что и для остальных *k*-меров из

перекрытия смещение будет примерно такое же. Это предположение основывается на том, что, хоть ошибки вставки и удаления присутствуют в чтениях, однако их не так много.

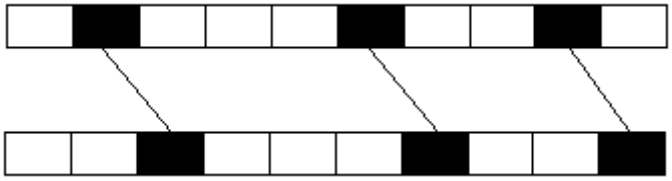

Рисунок 1 – Пример совпавших *k*-меров у двух чтений. Черным цветом обозначены совпавшие *k*-меры. Относительное смещение *k*-меров примерно сохраняется.

И если вдруг появляется *k*-мер, у которого оказалось совсем другое смещение, то в рамках поиска перекрытия было бы полезно его не учитывать.

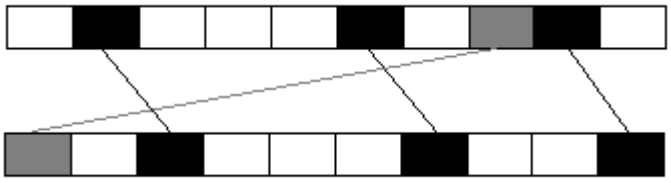

Рисунок 2 – Серым цветом помечен *k*-мер, смещение которого сильно отличается от смещений остальных *k*-меров.

К тому же, возможно такая ситуация, когда для пары чтений существует несколько локальных выравниваний, которые являются независимыми. Хотелось бы их тоже разделить и обрабатывать отдельно.

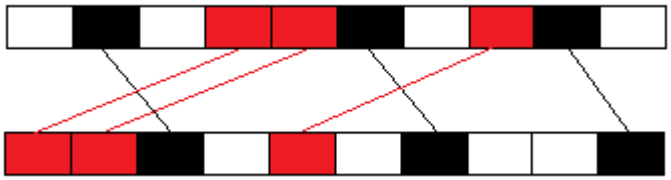

Рисунок 3 – Два различных перекрытия.

Было решено решить эту проблему разбиением всех *k*-меров для пары чтений по диагоналям в зависимости от их смещения. Выбиралась ширина диагонали s (в плане вычислений удобно брать 64), и высчитывалось максимальное число диагоналей, исходя из ширины и максимальной длины чтений. Потом, каждый *k*-мер некоторый пары чтений a и b, где в чтении a *k*-мер начинался с позиции i, а в b с позиции j, помещался в следующую диагональ: $d = \lfloor (i - j) / s \rfloor$. Все остальные вычисления проводились для каждой из диагоналей.

2.3.4.2 Подсчет числа общих пар оснований

Подсчитывать просто число общих *k*-меров не совсем корректно, даже если делать это для каждой диагонали в отдельности. Так, эти *k*-меры могут идти подряд, а в дальнейшем рассматриваемые чтения могут и не пересекаться. Более объективным является подсчет общих элементов между парой чтений на диагонали. Так, если, к примеру, три *k*-мера подряд совпадут, то количество общих элементов будет $k + 2$. Если же эти три *k*-мера не пересекаются, то количество элементов уже $3k$.

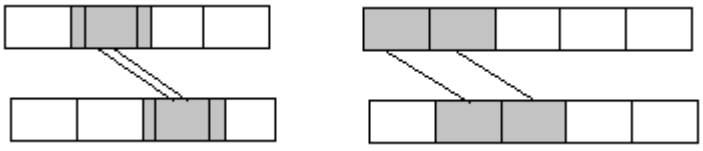

Рисунок 4 – Варианты расположения совпавших *k*-меров.

Реализовывается эта эвристика следующим образом. Полученный список после объединения по общим *k*-мерам мы будем сортировать не только по a и b, а ещё и по i. и тогда, идя по списку, i будет только увеличиваться для каждой пары чтений. Это позволит нам хранить для каждой диагонали некоторый индекс – позицию в чтении, в которой

заканчивает последний найденный общий *k*-мер. При рассмотрении следующего общего *k*-мера, в диагонали смотрится эта правая граница, в которой закончился предыдущий *k*-мер. На основании этих данных мы можем сделать вывод о пересечении предыдущего *k*-мера и рассматриваемого. И посчитать количество нуклеотидов, которые совпали в этих *k*-мерах и не были посчитаны ранее. После каждого шага мы сдвигаем правую границу в соответствии с позицией, в которой закончился рассматриваемый *k*-мер.

2.3.5 Построение списков кандидатов

После того, как мы сформировали и отсортировали по *a, b* и *i* список общих *k*-меров, мы может взять пару чтений, посчитать, как описывалось выше количество общих нуклеотидов. Если это количество будет больше, чем некоторая константа, являющаяся параметром алгоритма, тогда считаем, что, вероятно, *k*-меры из рассматриваемой диагонали образуют локальное перекрытие. И нам нужно это проверить.

Начинать сравнение с каждой из позиций в диагонали, в обе стороны от *k*-мера не очень эффективно. Во-первых, многие элементы сравнения могут повторяться (если два *k*-мера идут подряд), а во-вторых, если начинаться неточное выравнивание с каждой из позиций в диагонали, то не очень понятно, когда нужно остановиться. Если ставить ограничение на длину сравниваемого участка, то могут остаться не просмотренные участки между найденными *k*-мерами.

То, что исходный список отсортирован так же по *i*, дает нам возможность для каждой диагонали хранить некоторую правую границу, место, где закончился последний найденный *k*-мер. Так и будем идти по каждой диагонали. Если в какой-то момент позиция начала вхождения рассматриваемого *k*-мера оказывается больше, чем запомненная правая граница для соответствующей диагонали, то это означает, что между

рассматриваемым *k*-мером и предыдущем *k*-мером, есть участок, который содержит ошибки. Мы можем вычислить этот участок. На этот участок можно ввести ограничение на максимальную длину, при которой мы пытаемся его выровнять. Для удобства и улучшения эффективности работы алгоритма выравнивания, можно подразбить его на небольшие участки по 64 символа или меньше. Таким образом мы обработаем все участки между найденными *k*-мерами, каждый участок в рамках диагонали будет просмотрен только один раз. В то же время, участки, для которых заранее известно, что они совпадают (объединенные *k*-меры), просматриваться не будут. После того, как для каждой диагонали будет сформирован списков участков между найденными *k*-мерами, для получения более точных результатов, от крайних *k*-меров можно посмотреть выравнивания в различные стороны, на определенную длину, на несколько участков по 64 символа.

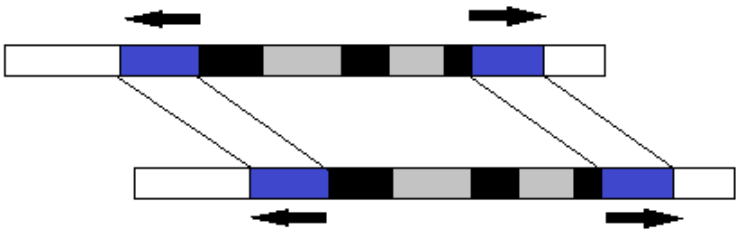

Рисунок 5-Просмотр окружения по бокам от крайних найденных *k*-меров.

Количество таких участков *p* является настраиваемым параметром алгоритма.

2.4 Реализация фильтрации

В данном пункте будут описаны некоторые детали реализации, которые пояснят, как был реализован описанный алгоритм.

На первом этапе поиска *k*-меров чтения, создаются два массива для каждого из блоков, соответствующих друг другу по индексу. Первый массив

содержит значения типа *int64*, которые будут являться ключом при дальнейшей сортировке, второй *int32*, которые будут выступать в роли значений соответствующих ключей. Для каждого чтения из блока мы может просто идти указателем от начала и до конца, пересчитывая каждый раз за время $O(1)$ хеш. Таким образом, время работы хеширования будет линейным.

Когда массивы сформированы, они отправляются на цифровую сортировку. Так как ключ соответствует некоторому *k*-меру, то длины всех ключей будут одинаковыми и сортировка будет работать особенно быстро.

После сортировки обоих списков, они отправляются на объединение. Ставятся два указателя на начала обоих списков и продвигаясь вперед определяются для пары чтений позиции, где их *k*-меры совпадают. Каждое найденное совпадение записывают в новый список, который является массивом значение *int64*. В список необходимо поместить четыре позиции: номер чтения *a*, номер чтения *b*, позиция начала вхождения *i k*-мера в чтении *a*, позиция начала вхождения *j* в чтении *b*. Как уже описывалось выше, эти числа можно уместить в *int64*. В старших 17 разрядах будет содержаться номер чтения *a*, потом номер *b*, потом позиция *i*, потом позиция *j*.

Отправим полученный массив на цифровую сортировку. Таким образом получим отсортированный сначала по *a, b*, потом по *i* массив элементов. Далее, идём по списку и для каждой пары чтений *a, b* сначала подсчитываем количество общих нуклеотидов в каждой диагонали, и если их оказывается довольно много, по алгоритму, описанному выше, находим промежутки, которые необходимо проверить, добавляя их в массив, который будет далее отправлен на выравнивание.

2.5 Выравнивание двух строк

На этом этапе задача ставилась следующая. Есть две подстроки a и b. И есть позиция, с которой начинается общий k-мер: a_i, b_j. Нужно найти подстроки в a и b, включающие этот k-мер, с редакционным расстоянием, не превышающим заданное значение.

Используя ту особенность, что чтения состоят всего из четырех различных символов, был разработан алгоритм, который использовал битовые маски этих символов, которые были в свою очередь представлены 64-битным числом. Что позволило очень быстро производить сравнение строк при помощи исключительно битовых операций.

Для выравниваний был реализован алгоритм *Bitap*, описанный в главе 1. Этот алгоритм работал очень эффективно на представленных данных в силу их некоторых особенностей. А именно, из-за того, что алфавит сравниваемых строк состоит всего из четырех символов. Так же, кандидаты для выравнивания были подготовлены таким образом, что их длины не превышали 64 символов.

На вход в cuda ядро попадают данные:

- Номер чтения блока *A*;
- Номер чтения блока *B*;
- Позиция чтения блока *A*, с которой начинается общий *k*-мер;
- Позиция чтения блока *B*, с которой начинается общий *k*-мер;
- Длина подстроки, которую требуется сравнить, блока *A*;
- Длина подстроки, которую требуется сравнить, блока *B*.

Далее, ядро подгружает соответствующие входным данным подстроки чтений. Заметим, что длины этих подстрок не будут превышать 64 символов. Для одной из подстрок далее строятся четыре маски, соответствующие каждому из нуклеотидов. Если в маске, соответствующей нуклеотиду, скажем, ‘G’, стоит единичка в позиции i, то это значит, что в подстроке на i-й

позиции стоял нуклеотид ‘G’. Так как длина подстроки не превышает 64 символов, то вся маска помещается в *long long int*. Тогда подстроке соответствуют 4 числа *long long int*, которые являются масками. После подготовки, указанный алгоритм выполняется и выдает ответ в виде количества ошибок вставки, удаления и замены, найденных между двумя сравниваемыми строками.

Таким образом, мы наиболее эффективно реализуем описанный алгоритм, используя вычислительные способности GPU.

2.6 Объединение результатов

После выполнения выравнивания, ядро записывает количество найденных ошибок (расстояние Левенштейна) для каждого исследуемого промежутка.

Далее, мы идем по списку, сформированному при объединение общих *k*-меров, после сортировки. Мы так же находим полностью совпавшие участки, но теперь, мы знаем сколько ошибок содержится в промежутке между совпавшими участками. Теперь мы объединяем не только общие *k*-меры, но и делаем объединение по этим участком, высчитывая количество ошибок каждый раз на нём. Все объединения делаем в рамках диагоналей, чтобы была возможность найти разные перекрытия. После объединения всех участков, мы считаем длину найденного перекрытия. И, если, длина этого перекрытия получилась больше ещё одного параметра алгоритма τ, а при этом количество ошибок $< \varepsilon$, то мы выводим найденное перекрытие.

2.7 Параметры алгоритма

Для настройки точности работы, чувствительности или времени работы в алгоритме используются различные параметры. Меняя эти параметры

можно добиться желаемых характеристик. Чем выше будет точность, тем дольше будет работать алгоритм. И наоборот, можно выбрать такие значения, которые позволят на порядок быстрее обрабатывать данные, но и точность будет сильно меньше. Но можно найти компромисс и выбирать средние значения. В 3 главе будет приведен анализ различных значений параметров.

В программе используются следующие параметры:

1) Параметр *k*. Этот параметр задает длину *k*-мера, которые мы будет рассматривать, хешировать. Так как возникновение ошибки любой из позиций равновероятно, а всего ошибок у нас около 15%, то не следует брать *k* слишком большим. Вероятность того, что 30 нуклеотидов подряд будут прочитаны правильно довольно мала. Оптимально использовать *k*=14.
2) Параметр *h*. Определяет необходимое минимальное количество общих нуклеотидов в каждой из диагоналей, необходимое для дальнейшего выравнивания.
3) Параметр *p*. Определяет число участков по 64 символа, которые необходимо просмотреть и выровнять для двух подстрок, от крайних найденных k-меров в обе стороны.
4) Параметр τ. Определяет минимальную длину перекрытия. Если мы нашли перекрытие меньше длины, то выводиться оно не будет. Никак на производительность не влияет.
5) Параметр *ε*. Определяет допустимую долю ошибок в найденном перекрытии.
6) Параметр *s*. Определяет ширину диагонали. Оптимально брать 64 или 128.

2.8 Вывод по главе 2

В данной главе был описан предлагаемый алгоритм к поиску локальных выравниваний. Был рассмотрен каждый из этапов работы алгоритма, описаны детали реализации предлагаемого подхода.

3 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

В ходе разработки были проведены различные исследования, чтобы оценить производительность, время работы и точность предложенного метода. А так же результаты применения эвристик.

Так, замена подсчета числа общих *k*-меров, на подсчет оснований позволила сократить в среднем в пять раз количество элементов для выравнивания.

А разделение на проверяемые участки позволило в рамках каждой диагонали ровно один раз смотреть каждый участок, что так же сократило число проверяемых элементов. И более эффективно формировать ответ, давая данные сразу о больших участках перекрытий.

3.1 Исследование корректности и точности работы алгоритма

Для подтверждения корректности работы метода и оценки его точности, для известного двухмиллионного генома было сгенерировано случайным образом десятикратное покрытие длинными чтениями. В полученные чтения с некоторой вероятностью были добавлены ошибки вставки, удаления и замены. При форматировании покрытия, для каждого полученного чтения была известная его позиция в настоящем геноме. Это позволило составить точный список перекрытий между всеми парами чтений генома. В таблице 1 представлены результаты экспериментов, а так же приведено сравнение точности с аналогом DALIGNER[23].

Всего в исследуемых чтениях было 18170 перекрытий (Это было подсчитано точным методом).

k	*My*		*Daligner*	
	Количество ненайденных перекрытий	*Точность метода*	*Количество ненайденных перекрытий*	*Точность метода*
12	8	99.96%	15	99.91%
13	37	99.79%	72	99.6%
14	93	99.48%	158	99.13%
15	236	98.7%	461	97.46%
16	513	97.17%	983	94.59%
17	1170	93.5%	2104	88.4%
18	2060	88.7%	3756	79.3%
19	3216	82.3%	4963	72.68%
20	4107	77.4%	6210	66%

Таблица 1 – точность метода при различных значениях *k*.

Тестирование проводилось на следующей аппаратуре:

- Процессор: AMD Phenom II X4 955
- Видеокарта: nVidia GeForce GTX 770

По точности метод несколько превосходит свой аналог. Помимо того, что предложенный метод позволяет находить несколько больше перекрытий, по сравнению с аналогом, на найденные перекрытия он дает вдвое лучшую верхнюю оценку наличия ошибок между ними. Это происходит потому, что предложенный метод анализирует весь промежуток между найденными *k*-мерами. Особенно разница заметна для больших *k*, когда найденных общих *k*-меров в перекрытие не так много. Как показывает таблица, для наиболее точных результатов следует брать $k \leq 15$.

Для подтверждения того, что, метод правильно делает верхнюю оценку числа ошибок в найденном перекрытии, полученные перекрытия

проверялись точным методом. Результаты показали, что метод дает корректную верхнюю оценку числа ошибок в промежутке.

3.2 Оценка эффективности работы алгоритма при различных параметрах

Далее исследовалась зависимость времени работы предложенного метода, числа общих найденных *k*-меров и числа промежутков, предоставленных для выравнивания в зависимости от выбранного *k*. Эти данные помогут понять изменение точности работы метода в зависимости от выбора параметра, а так же изменение скорости работы метода.

k	Время работы	Число общих *k*-меров	Число промежутков	Элементов для выравнивания
12	14.063 сек	39 569 428	1 671 575	5 578 969
13	9.322 сек	13 503 958	1 205 632	4 016 585
14	8.032 сек	5 897 402	895 394	3 751 664
15	7.49 сек	3 309 370	666 120	3 558 390
16	7.12 сек	2 183 054	494 824	3 367 469
17	6.48 сек	1 551 058	365 279	3 148 759
18	6.27 сек	1 135 796	268 396	2 894 761

Таблица 2 – Результаты тестирования времени работы и количества найденных *k*-меров, в зависимости от *k*.

Как видно из таблицы, чем больше k , тем быстрее выполняется программа. Этот результат можно было предсказать. Так как, чем больше *k*, тем больше вероятность, что произвольно выбранный *k*-мер будет содержать ошибку. А значит, тем меньшее число общих *k*-меров мы сможем обнаружить. Второй столбец подтверждает это предположение.

Третий столбец таблицы содержит данные о числе совпавших участках – участков, после объединения пересекающихся одинаковых *k*-меров в чтениях, в которых предполагается наличие перекрытий. Как видно, число таких участков так же уменьшается с ростом *k*. Однако, не прямо пропорционально числу общих *k*-меров. Во-первых, не все совпавшие *k*-меры содержатся в перекрытиях. При небольших *k*, довольно высокая вероятность случайного совпадения *k*-меров в чтениях. Во-вторых, при небольших *k*, гораздо большее число *k*-меров объединяются в промежутки, нежели, чем при больших *k*.

По предоставленным двум таблицам можно сделать вывод о том, что *k* эффективно брать равным 13 или 14.

3.3 Сравнение с аналогом

Далее, было произведено сравнение времени работы разработанного алгоритма и его аналога DALIGNER. Сравнение проводилось на чтения, сгенерированных по геному заданной длинны, покрывающих его 10 раз.

Длина генома	My	Daligner
1Mb	3.09 сек	8.3 сек
2Mb	8.032 сек	26 сек
5Mb	18.387 сек	93 сек
10Mb	59.41 сек	313 сек

Таблица 3 – Сравнение времени работы.

Таким образом, можно сделать вывод, что предложенный метод работает быстрее своего аналога.

3.4 Вывод по главе 3

Были проведены исследования точности работы метода при различных параметрах. Проверена корректность работы метода. Алгоритм был протестирован при различных параметрах, что дало возможность подобрать наиболее эффективные параметры для работы алгоритма. Было проведено сравнение метода с аналогом. Качество предложенного метода отказалось выше, чем качество аналога, при меньшем времени работы.

ЗАКЛЮЧЕНИЕ

В работе был предложен алгоритм для поиска перекрытий между длинными чтениями, содержащими ошибки вставки, удаления и замены символов. Разработанный алгоритм позволил перенести наиболее трудоемкие вычисления на GPU, тем самым снизив нагрузку на CPU и сократив время работы. Разработанный алгоритм был проанализирован.

Предложенный алгоритм был реализован на языке C++ с использованием технологии CUDA для вычислений на GPU. Разработанная программа была успешно протестирована на различных данных, было исследовано качество результатов и время работы при различных параметрах.

Программа показала более высокую эффективность, по сравнению с аналогами.

СПИСОК ЛИТЕРАТУРЫ

1. http://www.pacificbiosciences.com/products/
2. Sanger F., Nicklen S., and Coulson A.R. Dna sequencing with chainterminating inhibitors. // Proc Natl Acad Sci U S A. Dec 1977. Vol. 74. No. 12. pp. 5463–5467
3. Staden R. A strategy of dna sequencing employing computer programs // Nucleic Acids Res. Jun 1979. Vol. 6. No. 7. pp. 2601–2610.
4. Quail M., al E. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers // BMC genomics. 2012. Vol. 13. No. 1. P. 341.
5. Myers E.W. The fragment assembly string graph // Bioinformatics. 2005. Vol. 21. No. supl 2. pp. ii79-ii85
6. Jennifer Commins**,** Christina Toft**,** Mario A Fares. Computational Biology Methods and Their Application to the Comparative Genomics of Endocellular Symbiotic Bacteria of Insects. Biological Procedures Online, 2009
7. de Bruijn N. G. A Combinatorial Problem // Koninklijke Nederlandse Akademie v. Wetenschappen. Vol. 49. Pp. 758–764
8. http://soap.genomics.org.cn/soapaligner.html
9. SHRiMP2: Sensitive yet Practical Short Read Mapping, Matei David, Misko Dzamba, Dan Lister, Lucian Ilie and Michael Brudno, Bioinformatics (2011) 27(7):1011—1012
10. Burrows-Wheeler Aligner, http://bio-bwa.sourceforge.net
11. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg, Genome Biology 2009, 10:R25.
12. Udi Manber, Sun Wu. "Fast text search allowing errors." Communications of the ACM, 35(10): pp. 83–91, October 1992

13. http://www.nvidia.ru/object/cuda-parallel-computing-ru.html
14. http://www.nvidia.ru
15. http://docs.nvidia.com/cuda/thrust/
16. Дональд Кнут. Искусство программирования, том 3. Сортировка и поиск. – 2-е изд. – М: «Вильямс», 2007. – С.192–201.
17. http://code.google.com/p/back40computing/
18. Gene Myers. Efficient local Alignment Discovery amongst Noisy Long Reads// WABI 2014, Pp.52-67.
19. Gene Myer, E.W. An O(ND) difference algorithm and its variations. // Algorithmica 1, 1986, Pp.251-266.
20. Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. // Genome Biology 2009.
21. Клещенко Е. Читаем ДНК: в сто раз быстрее, в тысячу раз дешевле//Химия и жизнь, 2006, №1
22. Я.М.Краснов, Н.П.Гусева, Н.А.Шарапова, А.В.Черкасов Современные методы секвенирования ДНК (ОБЗОР) ФКУЗ «Российский научно-исследовательский противочумный институт «Микроб, Саратов, Российская Федерация»
23. Сергушичев А.А. Разработка метода восстановления фрагментов генома по парным чтениям с ошибками вставки и удаления. НИУ ИТМО, 2013. Магистерская диссертация.