Санкт-Петербургский государственный университет

Кафедра системного программирования

Смирнов Кирилл Вадимович

Распараллеливание на GPU алгоритма SSV

Курсовая работа

Научный руководитель:
ст. преп. Сартасов С. Ю.

Санкт-Петербург
2018

Оглавление

Введение

Бурное развитие биологии и информатики за последние 40-50 лет привело к появлению новой дисциплины - биоинформатики. Среди задач, которые стоят перед специалистами в этой области, можно назвать анализ генетических последовательностей для определения эвалюционного родства между видами.

Белки могут быть представлены как последовательность аминокислот, которые в свою очередь могут быть представлены в виде символов (A, G, C, T). Для определения похожести двух последовательностей были введены специальные метрики. Высокая степень похожести может свидетельствовать об эволюционной связи между последовательностями. Процесс называется выравниванием (alignment) и часто применяется на практике. Например, определение наличия или отсутствия ретровируса у двух видов может помочь определить их эволюционную близость.

Размеры входных данных могут достигать нескольких миллиардов аминокислот, что исключает возможность обрабатывать их без вычислительной техники. Большая востребованность привела к появлению целого ряда инструментов для решения задачи выравнивания, например, BLAST [1] (1990), HMMER [2] (1996). Развитие компьютерной техники и популяризация многопоточного программирования предопределили вектор развития современных решений.

Как правило на практике входными данными являются файл с интересующими нас последовательностями и образец (например, белок), который нужно найти.

1. Постановка задачи

Цель данной работы заключается в повышении производительности решений, основывающихся на архитектуре HMMER, за счет оптимизации алгоритма SSV. Для ее достижения были сформулированы следующие задачи:

1. Провести анализ существующих реализаций SSV
2. Разработать архитектуру нового решения
3. Реализовать разработанную архитектуру
4. Провести апробацию новой реализации и оценить ее качество

2. CUDA и архитектура GPU

В последние годы разработчики CPU (центральных процессоров) столкнулись с рядом на данный момент непреодолимых сложностей, которые не позволяют применять старые подходы для дальнейшего повышения производительности. В то же время все большую популярность набирает практика использования видеокарт. Особенности архитектуры видеокарт позволяют применять новые подходы для решения задач. [3]

В отличие от CPU, в которых количество арифметико-логических устройств ограничивается несколькими десятками, GPU (видеокарта) может содержать несколько сотен, то есть GPU новую модель к вычислениям - SIMT (Single Instruction, Multiple Threads). Арифметико-логические устройства сгруппированы в блоки, называемые мультипроцессорами. Таким образом, GPU представляет множество мультипроцессоров, каждый из которых обладает множеством арифметико-логических устройств.

Для удобства разработчиков компания Nvidia предложила кроссплатформенную систему компиляции и исполнения программ CUDA (Compute Unified Device Architecture). С помощью CUDA у программистов появилась возможность разрабатывать ПО, которое может работать на CPU и GPU, без привязки к графическим интерфейсам. Важной особенностью CUDA является использование языков высокого уровня (Например, C++ и Fortran).

CUDA поддерживает иерархию потоков и памяти. Множество потоков (thread) в CUDA подчинены следующим правилам:

1. Потоки объединяются в блоки (block)
2. Количество потоков внутри одного блока не превосходит 1024
3. Каждому блоку сопоставляется определенный мультипроцессор, на котором этот блок будет исполняться
4. Каждому мультипроцессору может быть сопоставлено несколько блоков

5. Потоки внутри блока организуются в группы по 32 потока (warp). Потоки внутри одной группы исполняются параллельно. Синхронизация между группами потоков отсутсвует

CUDA поддерживает иерархию памяти, доступную на GPU:

1. Каждый поток получает в свое распоряжение некоторое количество регистров, которые доступны только внутри потока

2. Каждый блок получает определенное количество разделяемой памяти (shared memory), которая доступна всем потокам внутри блока

3. Глобальная память доступна всем потокам

4. Время доступа к регистрам и разделяемой памяти сопоставимы и намного меньше, чем время доступа к глобальной памяти

3. Обзор решений

3.1. BLAST (Basic Local Alignment Search Tool)

BLAST был впервые предложен в 1990 году и использовал PAM-120 матрицу. PAM-120 - это матрица аминокислотной замены, то есть матрица, которая кодирует ожидаемое эволюционное изменение на уровне аминокислот.

3.2. HMMER

Первая версия HMMER [2] появилась в 1996 году и в отличие от BLAST'а, использующего попарное сопоставление аминокислот, в HMMER использовалась скрытая марковская модель (HMM). Скрытая марковская модель - это модель процесса, процесс в которой считается Марковским (новое событие зависит только от предыдущего), причем неизвестно, в каком состоянии находится система, но каждое состояние S может с некоторой вероятностью P произвести событие O, которое можно наблюдать. Скрытая марковская модель позволяла строить более точную математическую модель. Одно из основных преимуществ скрытой марковской модели против матриц аминокислотной замены заключается в возможности учитывать "вставку" и "удаление" аминокислот из последовательности вследствие мутации.

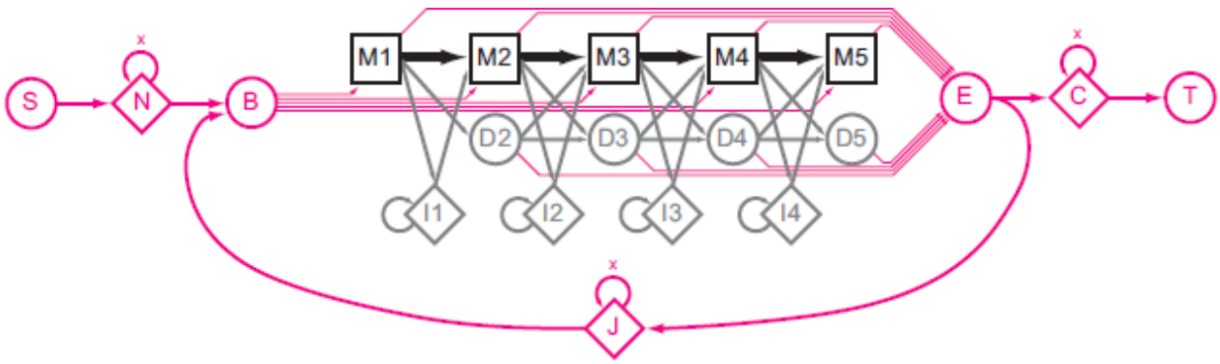

Рис. 1: Архитектура HMM, используемая в HMMER [6]

Модель (рис. 1) начинается с состояния S (Start) и следует по стрелкам до состояний T(Terminate). В каждом состоянии действием является либо испускание остатка(аминокислоты), либо выбор стрелки для перехода. Действия регулируются набором вероятностей. Данная модель (рис. 1) имеет 5 состояний совпадения M(Match), 4 состояния удаления D(Delete) и 4 состояния вставки I(Insert). Дополнительные состояния N, C и J испускают ноль и более остатков негомологичных областей, которые предшествуют, следуют или присоединяются между гомологичными областями. Состояния S, B, E, T являются неиспускающими.

С использованием HHM появилась возможность учитывать позицию аминокислоты в последовательности при подсчете степени схожести, что повысило качество результата. Недостатком данного решения долгое время оставалась низкая производительность в сравнении с BLAST'ом. Однако, с 3 версией HMMER'а был предложен конвейер, который привел к значительному ускорению. Его развитие продолжилось и в новой версии HMMER 3.1

Важным свойством используемых HHM является возможность их вычисления с использованием динамического программирования. Для подсчета результата используется двумерная матрица, столбцы которой соответствуют состояниям HMM, строки - аминокислотам последовательности (рис. 2).

3.3. Конвейер HMMAR 3.x

В HMMER 3.x был представлен pipeline [4, 5] (рис. 3), состоящий из MSV/SSV-Filter и P7Viterbi. Используемые эвристики позволяют отбросить последовательности, которые имеют заведомо низкую метрику похожести.

Кроме того, используемые эвристики основываются на упрощенных в сравнении с исходной HMM(рис. 4), что позволяет быстрее их обрабатывать. В MSV(Multiple Segment Viterbi) были удалены состояния, отвечающие за вставку (I) и удаление (D). В SSV(Single Segment Viterbi)

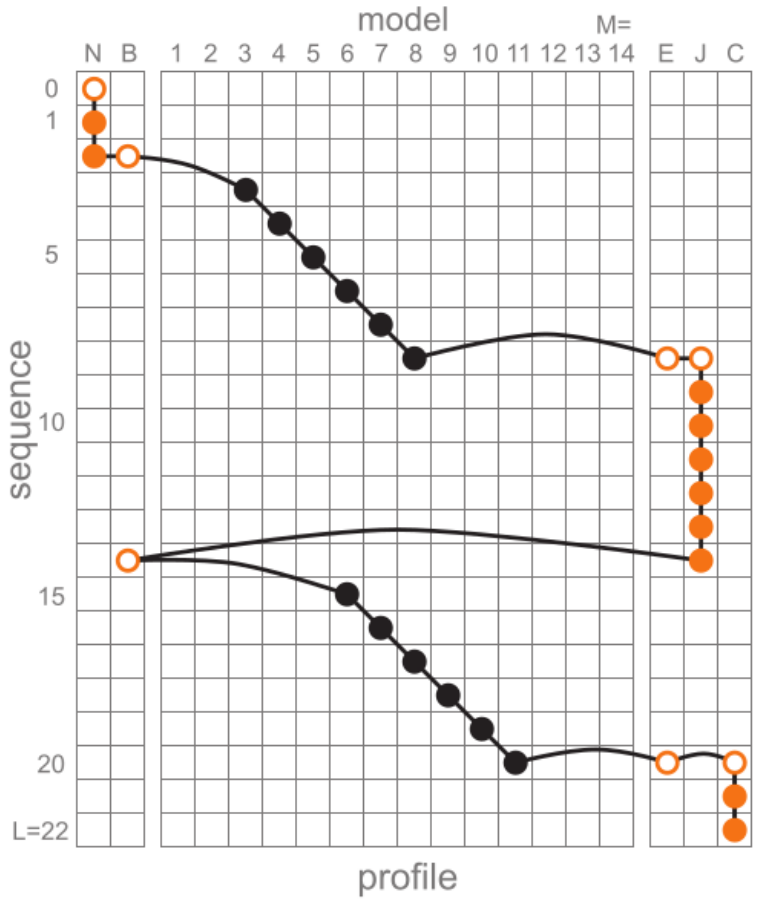

Рис. 2: Пример матрицы динамического программирования, используемой в HHMER [4]

исчезло состояние, отвечающие за конкатенацию двух частей последовательности (J). Корректность таких изменений HHM достигается за счет изменения значения функции похожести необходимого для перехода на следующую ступень конвейера

3.4. CUDAMPF

CUDAMPF [5] является многоступенчатым фреймворком для ускорения конвейера HMMER 3.x с использованием GPU. Авторы предложили четырехступенчатую архитектуру(рис. 5), которая позволяет ускорить каждую ступень конвейера.

Первая ступень предполагает распределение блоков с потоками по различным мультипроцессорам.

Вторая ступень предполагает, что каждая последовательность бу-

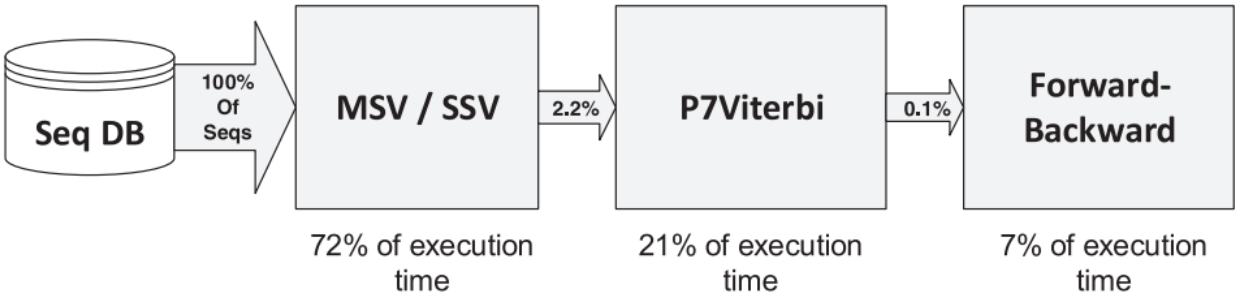

Рис. 3: Конвейер HMMER 3.1 [5]

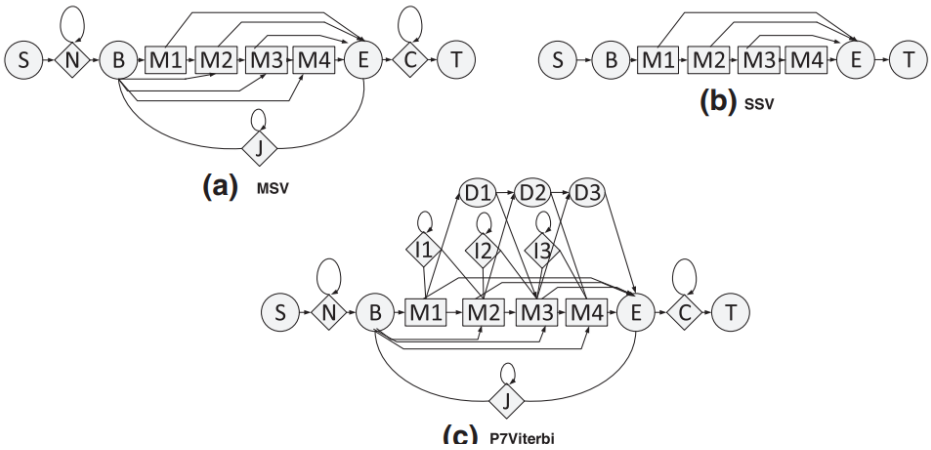

Рис. 4: HMM для различных ступеней конвейера HMMER 3.1 [5]

дет выполняться потоками одной группы. Это позволит избежать синхронизации между группами потоков.

Третья ступень предполагает, что все потоки внутри одной группы будут одновременно подсчитывать значения ячеек матрицы на одной строке, то есть для одного элемента последовательности и разных состояний HMM.

Четвертая ступень - это использование SIMD инструкций каждым потоком, что позволяет обрабатывать до 4 ячеек матрицы ДП (динамического программирования) за раз. Такой подход вынуждает использовать специфическое представление данных (layout) и производить обмен значениями между смежными потоками, что означает дополнительное межпоточное взаимодействие для каждого элемента последовательности.

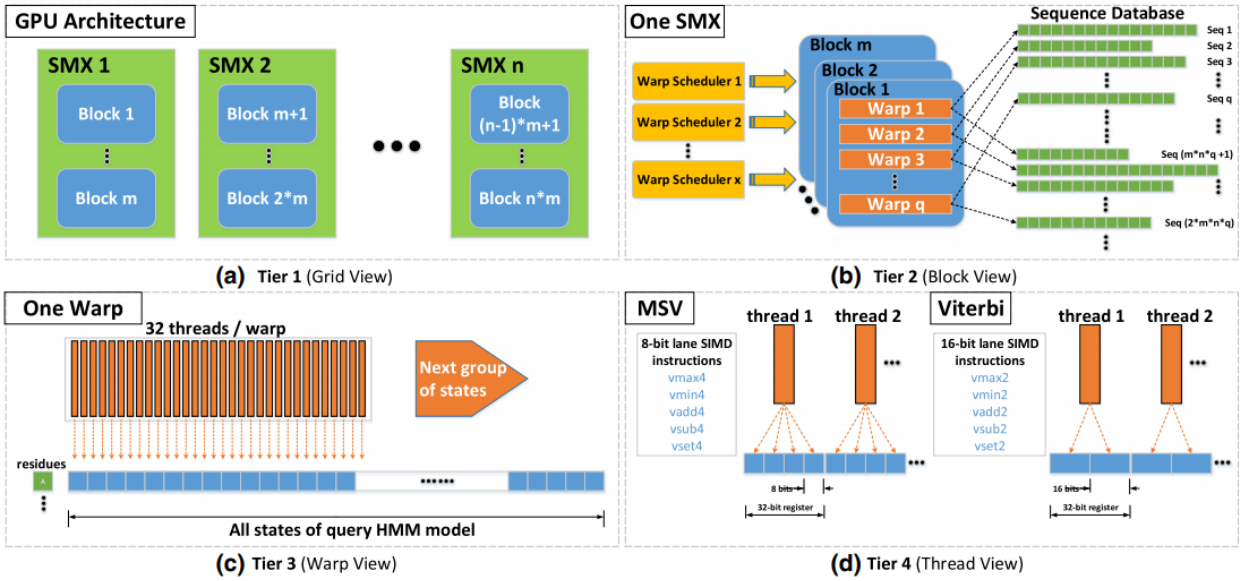

Рис. 5: Конвейер в CUDAMPF

4. Решение

В CUDAMPF матрицу ДП редуцировали до одной строчки, соответствующей предыдущему элементу последовательности (аминокислоте), которая хранится в разделяемой памяти. Таблица, соответствующая скрытой марковской модели, и последовательность аминокислот хранятся в глобальной памяти GPU. Архитектура HMM, которая используется для SSV, предполагает, что существует только диагональная зависимость между ячейками матрицы ДП. Подсчет ячеек матрицы, при котором для каждой диагонали выбирается поток, который посчитает все ячейки ей принадлежащие, позволяет избежать межпоточного взаимодействия вплоть до конца подсчета матрицы (рис. 6).

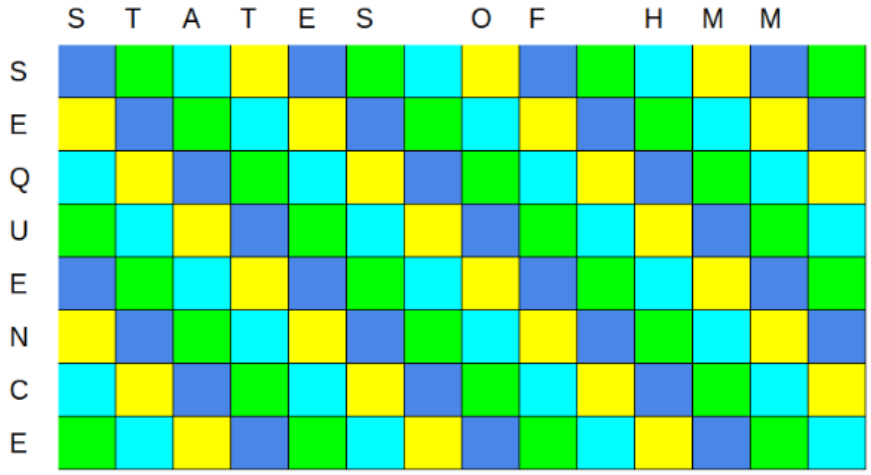

Рис. 6: Матрица ДП. Ячейки одного цвета обрабаывает один поток

Данный подход позволяет не хранить предыдущую строчку ДП, следовательно, увеличить доступное пространство в разделяемой памяти. Доступную разделяемую память разумно использовать для хранения скрытой марковской модели, потому что она меньше последовательности и является общей для всех потоков. Таким образом удалось уменьшить количество обращений к глобальной памяти для чтения скрытой марковской модели, что дало значительный прирост производительности в сравнении с архитектурой, в которой скрытая марковская модель находится в глобальной памяти (рис. 7).

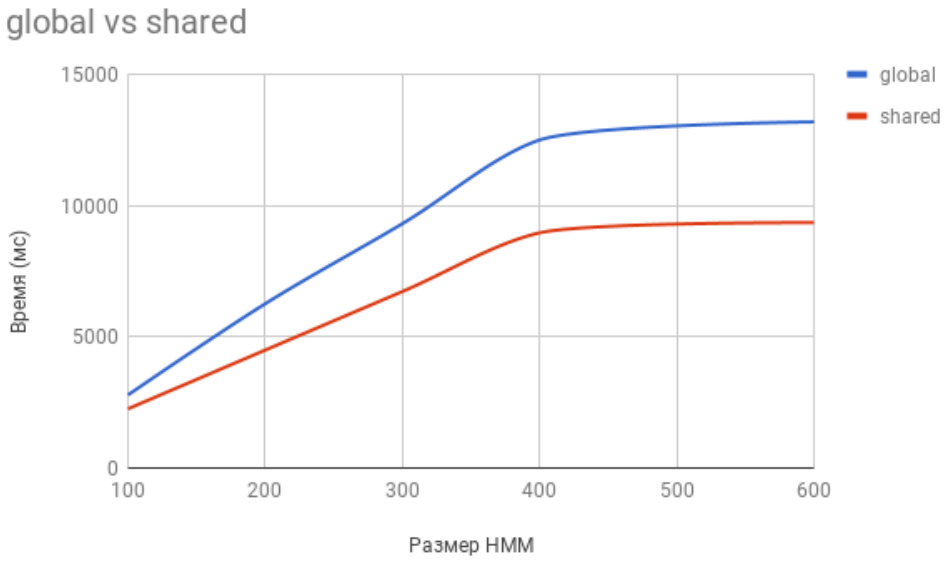

Рис. 7:

Однако, элементы последовательности по-прежнему хранятся в глобальной памяти. Последовательность хранится как беззнаковый целочисленный массив. Каждый элемент массива - это 4 последовательные аминокислоты. Данное решение позволяет повысить скорость чтения, так как за один раз считывается 4 элемента. Были разработаны следующие подходы чтения элементов последовательности:

1. Каждый поток независимо считывает один элемент последовательности из глобальной памяти

2. Каждый поток независимо считывает 4 элемента последовательности из глобальной памяти и последовательно обрабатывает их

3. Потоки совершают объединенное чтение 128 элементов последовательности (каждый поток считывает 4 элемента, которые объединенны в одину целочисленную переменную), которые записываются в разделяемую память. В разделяемой памяти участок последовательности хранится в циклическом массиве символьных переменных, то есть для определения позиции элемента последовательности в массиве в разделяемой памяти берется номер позиции элемента в массиве в глобальной памяти по модулю равному размеру массива в разделяемой памяти

4. Гибридный подход, при котором чтение происходит по аналогии со вторым подходом, но массив в разделяемой памяти является целочисленным. Каждый поток независимо считывает из него 4 элемента последовательности и последовательно обрабатывает их.

Для определения наилучшего алгоритма чтения последовательности были поставлены эксперименты (рис. 8). Для измерений использовалась видеокарта Nvidia GeForce 920M. В качестве исходных последовательностей были взяты последовательности участков ДНК дрожжей и комара, размер скрытой марковской модели во всех экспериментах был равен 300 состояниям.

	подход №1	подход №2	подход №3	подход №4
дрожжи	6728	3018	3207	3016
комар	3025	1295	2151	1194

Рис. 8: Сводная таблица экспериментов для алгоритмов чтения последовательности (время указано в мс).

Для дальнейшей оптимизации была предпринята попытка использовать SIMD-инструкции. Разрядность типов данных, используемых в матрице ДП, позволяет использовать SIMD-инструкции с фактором векторизации равным 4. Было проанализировано два подхода:

1. Диагональный (рис. 9). При данном подходе 4 ячейки матрицы ДП, которые обрабатываются с помощью SIMD-инструкций, могут соответствовать любым 4 последовательным состояниям марковской модели и любому набору из 4 аминокислот (число различных аминокислот может достигать 20). Таким образом, использование данного подхода сопряжено с высокими расходами памяти в сравнении с решением без SIMD-инструкций (Размер памяти для хранения марковской модели увеличится на несколько порядков).

2. Горизонтальный (рис. 10). При таком подходе 4 ячейки матрицы ДП, которые обрабатываются с помощью SIMD-инструкций, могут соответствовать любым 4 последовательным состояниям марковской модели, но только одной аминокислоте. Это влечет увеличение размеров используемой памяти для марковской модели в 4 раза, что позволяет использовать данных подход при маленьких размерах модели (не более 200 состояний).

Рис. 9: Модель использования SIMD-инструкций при диагональном подходе. Разные цвета ячеек соответствуют разным потокам, которые будут подсчитывать их значения. Номера в ячейках соответствуют порядку, в котором они будут обработаны.

Рис. 10: Модель использования SIMD-инструкций при горизонтальном подходе. Четыре ячейки матрицы ДП объединены в одну целочисленную переменную. Были добавлены фиктивные ячейки, которые не соответсвуют состояниям марковской модели и которые необходимы для обработки крайних (начальных и конечных) состояний модели. Для того, чтобы избежать влияния фиктивных ячеек на результат, им заранее присваивается определенное значения (-inf = -128).

В силу очевидного преимущества второго подхода по количеству используемой памяти было принято решение о его реализации. Таким образом, итоговое решение представляет из себя систему, в которой набор оптимизационных техник выбирается в соответствии с входными данными:

Размер HMM				
	< 230	< 600	< 1500	
SIMD	да	нет	нет	нет
Способ чтения	гибридный	гибридный	считывание 4 элементов (2 подход)	считывание 4 элементов (2 подход)
Размещение HMM	разделяемая память	разделяемая память	разделяемая память	глобальная память

Рис. 11: Финальная архитектура решения

Эксперименты показали, что применение всех доступных оптимизаций позволяет превзойти CUDAMPF и HMMER. Ограничения, вызванные размерами разделяемой памяти, вынуждают отказываться от различных способов оптимизации с ростом размеров входных данных, что влечет ухудшение производительности (рис. 12).

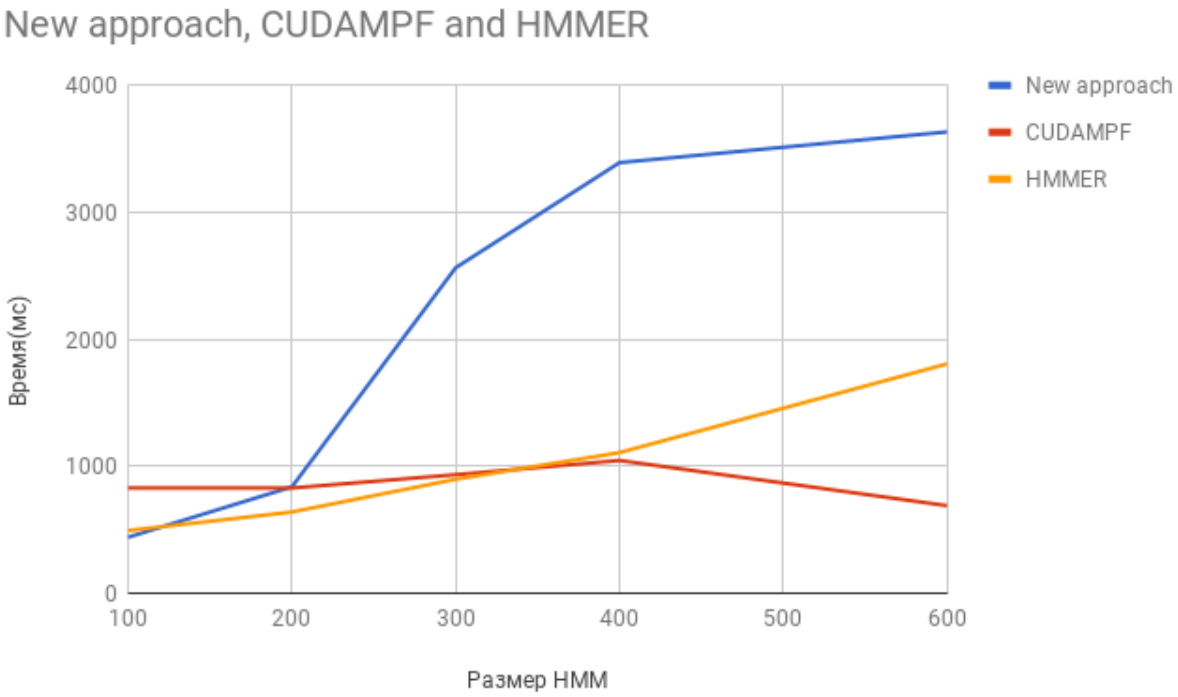

Рис. 12: В качестве исходной последовательности были взяты последовательность участков ДНК дрожжей. Были взята марковские модели размерами 100, 200, 300, 400 и 600

Список литературы

[1] Stephen F. AltschuP, Warren Gish, Webb Miller: Basic Local Alignment Search Tool, 1990

[2] Sean R Eddy: Hidden Markov models, 1996

[3] Боресков А. и др. - Параллельные вычисления на GPU: Архитектура и программная модель CUDA, 2012

[4] Eddy S.: Accelerated Profile HMM Searches, 2011

[5] Hanyu Jiang and Narayan Ganesan: CUDAMPF: a multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU, 2016

[6] Lin Cheng: Implementing and Accelerating HMMER3 Protein Sequence Search on CUDA-Enabled GPU