

IMPLEMENTATION OF SEARCHING FOR THE MOST FREQUENT DNA SEQUENCES USING THE KOKKOS LIBRARY

M. Kozlov, E. Panova, I. Meyerov

Lobachevsky State University of Nizhny Novgorod,
603950, Nizhny Novgorod, Russia

DOI: 10.24412/2073-0667-2024-2-58-71

EDN: TGQKBV

Nowadays, the wide variety of existing architectures raises the problem of developing universal approaches to programming. Various frameworks enable single-source code creation for multiple devices, for example, CPU, GPU, FPGA. Such frameworks include OpenCL, OpenACC, Kokkos, Alpaka and others. However, the problem of efficiency and performance portability remains relevant. It is not always possible to create one code that works efficiently on different devices because of their specific architectures. This article discusses performance aspects in relation to the Kokkos library, a widely used framework for creating cross-platform code.

As a benchmark, we consider a bioinformatics problem to find the most frequent DNA sequences of certain length. It is assumed that important genetic information can be encrypted in such sequences. DNA sequence can be represented as a string consisting of four characters “A”, “C”, “G”, “T”, which denote corresponding nucleobases. Therefore, the problem reduces to counting fixed-length patterns in DNA and can be solved using existing string matching algorithms. Faro and Lecroq (2013) reviewed and classified exact string matching algorithms and experimentally evaluated them on different kinds of texts. Hakak et al. (2019) showed the latest advancements in the field of string matching algorithms and designated modern trends and challenges. They analyzed various classes of algorithms and drew conclusions about the limitations and effectiveness of different string matching algorithms for various applications. In this article, we have chosen two algorithms for consideration: the well-known Rabin-Karp algorithm and the Hash3 algorithm from the Hash q family [Lecroq 2007]. The Hash3 algorithm is one of the most effective algorithms for short-length patterns of approximately 8 to 128 characters long. Both these algorithms are based on hashing and are well applicable for genome analysis. For verification and comparison, we also consider a simple naive algorithm based on sequential pattern matching.

The naive algorithm consists of character-by-character comparison of all fixed-length patterns. This algorithm is not effective enough, but has great potential for parallelization. We received an acceleration of up to 35 times when ported the parallel naive algorithm from CPU to GPU. The Rabin-Karp algorithm allows us to eliminate character-by-character comparisons effectively using hashing and shows better efficiency compared to the naive algorithm on both CPU and GPU. Our cross-platform parallel implementation of the Rabin-Karp algorithm is approximately 1.25 times faster than the naive algorithm on CPU and 2 times faster on GPU. The Hash3 algorithm cuts off character-by-character comparisons extremely efficiently. Because of this, the Hash3 algorithm is an order of magnitude faster than the naive algorithm. Due to the almost absence of character-by-character comparisons,

This work was funded by the Ministry of Science and Higher Education of the Russian Federation, project No. FSWR-2023-0034.

the algorithm is memory bound and has less potential for parallelization. The Hash3 algorithm was accelerated by 7 times on GPU relative to CPU.

We implement these algorithms for CPU and GPU using OpenMP, Cuda and Kokkos technologies. We demonstrate that when using Kokkos with a naive algorithm, the performance loss does not exceed 10 % relative to the OpenMP version. Losses are caused by the compiler making more efficient use of SIMD calculations in the OpenMP implementation when matching patterns. There is no performance loss for the Rabin-Karp and Hash3 algorithms when porting the OpenMP version to Kokkos. Speedup of all algorithms is about 14 times on 16 physical cores. It is worth noting that the Hash3 algorithm showed a noticeable improvement on the CPU when using hyper-threading, unlike other algorithms under consideration. This can be explained by more efficient memory management. Speedup on 32 threads and 16 physical cores for the naive algorithm and the Rabin-Karp algorithm is 16–17 times, while for the Hash3 algorithm it is 25 times.

Next, we run the developed code on the GPU and show that the Kokkos version of the Rabin-Karp algorithm loses to the Cuda version on the GPU by no more than 10 %. At the same time, the Kokkos versions of the naive and Rabin-Karp algorithms outperform our Cuda baseline version by 10–20 %. The authors did not set themselves the goal of optimizing the Cuda code. We believe that it is possible to optimize the Cuda code to match the performance of the Kokkos version. However, it is noteworthy that sometimes the baseline Kokkos version runs faster than the baseline Cuda version.

Overall, we demonstrate that in many cases the Kokkos version works as well as native OpenMP or Cuda code. In the worst case, the performance loss was no more than 10 %. We believe that paying this price is reasonable in order to run a single code on different devices.

Key words: Kokkos, single-source programming, cross-platform software, heterogeneous computing, program performance, string matching algorithms, bioinformatics.

References

1. Gaster B., Howes L., Kaeli D. R., Mistry P., and Schaa D. Heterogeneous computing with openCL: revised openCL. Newnes, 2012.
2. Farber R. Parallel programming with OpenACC. Newnes, 2016.
3. Kokkos 3: Programming model extensions for the exascale era / Trott C. R., Damien LG, Arndt D., Ciesko J., Dang V., Ellingwood N., Gayatri R., Harvey E., Hollman D. S., Ibanez D., et al. // IEEE Transactions on Parallel and Distributed Systems. 2021. V. 33, N 4. P. 805–817.
4. Alpaka — an abstraction library for parallel kernel acceleration / Zenker E., Worpitz B., Widera R., Huebl A., Juckeland G., Knupfer A., Nagel W. E., and Bussmann M. // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) / IEEE. 2016. P. 631–640.
5. Reinders J. et al. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Springer Nature, 2021. P. 548.
6. The Kokkos library. [Electron. Res.]: <https://github.com/kokkos/kokkos>. Date of access: 10.01.2024.
7. Subirana J. A., Messeguer X. The most frequent short sequences in non-coding DNA // Nucleic acids research. 2010. V. 38, N 4. P. 1172–1181.
8. Faro S., Lecroq T. The exact online string matching problem: A review of the most recent results // ACM Computing Surveys(CSUR). 2013. V. 45, N 2. P. 1–42.
9. Exact string matching algorithms: survey, issues, and future research directions / Hakak S. I., Kamsin A., Shivakumara P., Gilkar G. A., Khan W. Z., and Imran M. // IEEE access. 2019. V. 7. P. 69614–69637.
10. Stephen G. A. String searching algorithms. World Scientific, 1994.
11. Al-Khamaiseh K., Alshagarin S. A survey of string matching algorithms // Int. J. Eng. Res. Appl. 2014. V. 4, N 7. P. 144–156.

12. Karp R. M., Rabin M. O. Efficient randomized patternmatching algorithms // IBM Journal of Research and Development. 1987. V. 31, N 2. P. 249–260.
13. Lecroq T. Fast exact string matching algorithms // Information Processing Letters. 2007. V. 102, N 6. P. 229–235.
14. Galil Z. A constant-time optimal parallel string-matching algorithm // Journal of the ACM (JACM). 1995. V. 42, N 4. P. 908–918.
15. Park J. H., George K. M. Efficient parallel hardware algorithms for string matching // Microprocessors and Microsystems. 1999. V. 23, N 3. P. 155–168.
16. Accelerating string matching using multi-threaded algorithm on GPU / Lin C. H., Tsai S. Y., Liu C. H., Chang S. C., and Shyu J. M. // 2010 IEEE Global Telecommunications Conference GLOBECOM 2010 / IEEE. 2010. P. 1–5.
17. Kouzinopoulos C. S., Michailidis P. D., Margaritis K. G. Multiple string matching on a GPU using cuda // Scalable Computing: Practice and Experience. 2015. V. 16, N 2. P. 121–138.
18. Kozlov M. A., Panova E. A., Meerov I. B. Implementation of searching for the most frequent DNA sequences using the Kokkos library // Mathematical modeling and supercomputer technologies. Proceedings of the XXIII International Conference (N. Novgorod, November 13–16, 2023) / Ed. prof. D. V. Balandina. Nizhny Novgorod: Publishing House of Nizhny Novgorod State University, 2023. ISBN 978-5-91326-834-1. 2023. P. 73–78.
19. Benchmark source code. [Electron. Res.]: https://github.com/Mishaizlesa/most_common_string_kokkos. Date of access: 10.01.2024.
20. DNA Bank (National library of medicine). [Electron. Res.]: <https://www.ncbi.nlm.nih.gov/genbank>. Date of access: 10.01.2024.

РЕАЛИЗАЦИЯ ПОИСКА НАИБОЛЕЕ ЧАСТО ВСТРЕЧАЮЩИХСЯ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ДНК С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ KOKKOS

М. А. Козлов, Е. А. Панова, И. Б. Мееров

Национальный исследовательский Нижегородский государственный университет
им. Н. И. Лобачевского,
603950, Нижний Новгород, Россия

УДК 004.424, 575.112

DOI: 10.24412/2073-0667-2024-2-58-71

EDN: TGQKBV

Существующее на текущий день большое разнообразие архитектур ставит вопрос разработки универсального программного обеспечения. В связи с этим появляются и развиваются различные программные средства, позволяющие создавать единый кроссплатформенный код для запуска на CPU, GPU, FPGA и других архитектурах. Тем не менее, остается вопрос эффективности и переносимости производительности разработанного кода. В данной работе мы исследуем этот и другие аспекты применительно к библиотеке Kokkos, которая на сегодняшний день является одним из наиболее популярных средств для создания кроссплатформенного кода. В качестве бенчмарка мы рассматриваем задачу из области биоинформатики по поиску наиболее часто встречающихся последовательностей ДНК, которая решается с использованием строковых алгоритмов. Мы приводим несколько алгоритмов решения задачи, реализуем их с использованием технологий OpenMP, Cuda и Kokkos и демонстрируем, что потери производительности при использовании Kokkos не превышают 10 %, в то время как код может быть запущен как на CPU, так и на GPU.

Ключевые слова: Kokkos, кроссплатформенное ПО, гетерогенные вычисления, оптимизация программ, строковые алгоритмы, биоинформатика.

Введение. На сегодняшний день существует большое число аппаратных архитектур, требующих уникального подхода к разработке параллельного программного обеспечения. Как следствие, появляется множество реализаций одного и того же кода под различные архитектуры, заниматься одновременной поддержкой которых требует достаточно большого количества ресурсов. Требуются средства разработки универсального программного обеспечения, и такие средства существуют и активно развиваются в последнее десятилетие. Среди них можно отметить OpenCL [1], OpenACC [2], Kokkos [3], Alpa [4], oneAPI [5] и др. Однако вопрос эффективности и переносимости производительности разработанного с помощью этих средств кода остается открытым.

В рамках данной работы рассматривается программный пакет Kokkos [3, 6], являющийся удобным C++ фреймворком для создания переносимого кода, который может быть

Работа выполнена при поддержке Министерства науки и высшего образования России, проект № FSWR-2023-0034.

запущен на различных аппаратных платформах (CPU, GPU). Мы исследуем некоторые аспекты написания переносимых производительных программ, демонстрируем особенности организации кода с использованием синтаксиса Kokkos и показываем, что Kokkos позволяет разрабатывать универсальное программное обеспечение для различных устройств без существенных потерь в производительности.

В качестве демонстрационного примера мы рассматриваем задачу из области биоинформатики по поиску наиболее часто встречающихся паттернов фиксированной длины во фрагменте ДНК [7]. Предполагается, что такие паттерны потенциально могут нести некоторую генетическую информацию. ДНК представляется в виде последовательности четырех азотистых оснований, традиционно обозначаемых символами «А», «С», «G», «Т». Таким образом, задача сводится к поиску наиболее часто встречающихся подстрок небольшой фиксированной длины m ($4 \lesssim m \lesssim 100$) в последовательности ДНК достаточно большой длины n ($10^3 \lesssim n \lesssim 10^6$), состоящей из символов четырехбуквенного алфавита. В такой постановке задача решается с использованием строковых алгоритмов. Стоит отметить, что в биоинформатике имеет смысл учитывать генетические мутации и выполнять поиск последовательностей с не более чем k несоответствиями (« k -mismatches problem»), однако в данной статье рассматриваются только точные совпадения.

Задача поиска подстроки встречается во многих областях, таких как обработка естественных языков, речи, обработка изображений, компьютерное зрение, распознавание образов, биоинформатика. На сегодняшний день существует множество алгоритмов, основанных на посимвольном сравнении, хешировании, побитовых операциях, конечных автоматах и т. д. [8–11]. Многие алгоритмы показывают очень хорошие результаты на данных, характерных для определенных областей приложений. Фаро и Лекрок [8] представляют классификацию существующих алгоритмов и делают сравнительный анализ их эффективности для текстов из различных областей приложений. Хакак и др. [9] демонстрируют последние достижения в области алгоритмов на строках, обозначают современные проблемы и тенденции и делают выводы касательно применимости тех или иных алгоритмов в различных приложениях.

Задачи биоинформатики имеют свою специфику, связанную с небольшим размером алфавита и необходимостью выполнять поиск множества различных паттернов в одном и том же тексте. Для таких задач хорошие результаты демонстрируют алгоритмы, основанные на хешировании, т. к. они дают возможность предобработки текста и легко расширяются на случай поиска множества паттернов. В данной работе мы рассматриваем два разных алгоритма из этого класса: известный алгоритм Рабина-Карпа [12] и алгоритм Hash3 из семейства Hash q [13], который является одним из наиболее эффективных алгоритмов для решения данной задачи [8]. Для сравнения в качестве базового алгоритма рассматривается наивный алгоритм, основанный на последовательном посимвольном сравнении подстрок. Стоит отметить, что наивный алгоритм и алгоритм Рабина-Карпа требуют намного больше посимвольных сравнений, чем алгоритм Hash3, что накладывает свои условия при оптимизации кода.

Одним из важных вопросов при реализации строковых алгоритмов является возможность их адаптации под современные параллельные системы [14–15], в том числе под графические ускорители [16–17]. В данной работе мы предлагаем параллельные реализации перечисленных выше алгоритмов применительно к рассматриваемой задаче, исследуем их эффективность на CPU и GPU. В качестве базовой технологии распараллеливания мы используем OpenMP для CPU и Cuda для GPU. Также мы рассматриваем возможность ис-

пользования библиотеки Kokkos для написания кроссплатформенного кода, изучаем некоторые аспекты производительности и делаем вывод о возможности использования Kokkos для подобного рода задач. Мы демонстрируем, что потери производительности при использовании Kokkos не превышают 10 %, в то время как один и тот же код может быть запущен как на CPU, так и на GPU.

Работа является расширенным вариантом материалов доклада Козлова М. А., Пановой Е. А., Меерова И. Б. «Реализация поиска наиболее часто встречающихся последовательностей ДНК с использованием библиотеки Kokkos» [18] в рамках конференции «Математическое моделирование и суперкомпьютерные технологии 2023», рекомендована к публикации программным комитетом конференции «Математическое моделирование и суперкомпьютерные технологии».

1. Постановка задачи. Рассматривается задача поиска наиболее часто встречающихся паттернов в ДНК. Фрагмент ДНК представляется в виде последовательности символов четырехбуквенного алфавита «А», «С», «Т», «G», где каждая буква соответствует одному из четырех азотистых оснований: аденин, цитозин, тимин, гуанин. С биологической точки зрения имеет смысл искать достаточно короткие паттерны, при этом точных границ не существует, рассматриваются как совсем короткие паттерны (от 1 до 6 символов), так и паттерны длиной 20–30 символов и более [7]. Как было ранее отмечено, выполняется поиск точных совпадений паттернов без учета мутаций. Таким образом, задача поиска наиболее часто встречающихся паттернов в ДНК может быть сведена к задаче вычисления числа вхождений каждой подстроки длины m ($1 \leq m \leq 1000$) в текст четырехбуквенного алфавита длины n ($10^3 \leq n \leq 10^6$). Общий псевдокод решения данной задачи демонстрирует листинг 1. Функция `patternCount` возвращает число вхождений паттерна в текст и представляет собой реализацию одного из рассматриваемых алгоритмов поиска подстроки. Подробнее данные алгоритмы описаны в разделе 2.

Листинг 1. Общий псевдокод программы. Входные данные: текст длины n и длина паттерна m . Выходные данные: массив частот для каждого паттерна.

```
int [] mostFrequentPatterns(text : string , m : int) :
    int n = text.length
    frequencies : array [0..n - m + 1] of int
    for i from 0 to n - m + 1: # parallel loop
        pattern = text[i..i + m - 1]
        frequencies[i] = patternCount(text , pattern , n , m)
    return frequencies
```

При решении задачи стоит учитывать, что один паттерн может встречаться в тексте много раз, и следует вычислять частоту вхождения паттерна однократно, исключая его из последующего рассмотрения. Однако множественные проверки не влияют на асимптотическую сложность и сравнительный анализ алгоритмов, поэтому мы не учитываем этот факт и рассматриваем псевдокод 1 как некоторый бенчмарк, основанный на сравнении строковых паттернов.

Поскольку в задаче требуется выполнять поиск многих паттернов, то имеет смысл выстраивать параллелизм именно по паттернам. Таким образом, параллельная схема не зависит от используемого алгоритма поиска. Специфика исходных данных позволяет утверждать, что параллельные подзадачи являются достаточно сбалансированными и будут выполняться за примерно одинаковое время.

2. Алгоритмы. 2.1. *Наивный алгоритм.* Представленный в данном разделе алгоритм, далее называемый «наивным», реализует очевидную идею поэлементного последовательного сравнения паттернов в цикле (листинг 2). Асимптотическая сложность данного алгоритма $O(mn)$, где n — длина исходного текста, а m — длина искомого паттерна. Поскольку для решения поставленной задачи наивный алгоритм применяется к каждому паттерну длины m исходного ДНК (листинг 1), то общая сложность всего алгоритма `mostFrequentPatterns` в худшем случае равна $O(mn^2)$. Данный алгоритм можно оптимизировать, например, обратить внимание на то, что в текущем варианте сравнение двух паттернов текста проводится минимум дважды, однако это не изменит асимптотику алгоритма. В данной работе наивный алгоритм используется для проверки корректности работы более сложных алгоритмов и в качестве точки отсчета при проведении анализа эффективности.

Листинг 2. Псевдокод наивного алгоритма.

```
int naive(text : string , pattern : string , n : int , m : int) :
    int answer = 0
    for i = 0 to n - m + 1:
        bool flag = true
        for j = 0 to m - 1:
            if text[i + j] != pattern[j]:
                flag = false
                break
        if flag == true:
            answer = answer + 1
    return answer
```

2.2. *Алгоритм Рабина-Карпа.*

Алгоритм Рабина-Карпа [12] является одним из самых известных алгоритмов поиска подстроки в тексте. Алгоритм основан на идее хеширования подстрок. Основная идея алгоритма заключается в том, чтобы сравнивать не сами паттерны, а их значения хеш-функции. Псевдокод алгоритма Рабина-Карпа представлен листингом 3. Алгоритм Рабина-Карпа имеет ту же сложность в худшем случае, что и наивный алгоритм, но на практике обычно работает быстрее.

Листинг 3. Псевдокод алгоритма Рабина-Карпа.

```
int rabinKarp(text : string , pattern : string , n : int , m : int) :
    int answer = 0
    int hashT = hash(text[0..m - 1])
    int hashP = hash(pattern)
    for i = 0 to n - m + 1:
        if hashT == hashP:
            answer = answer + 1
            hashT = hash(text[i..i + m - 1])
    return answer
```

Для того чтобы алгоритм Рабина-Карпа работал быстрее наивного алгоритма, требуется, чтобы хеш-функция подстроки вычислялась достаточно быстро. Один из вариантов — предварительно вычислить хеш-функции всех подстрок текста. Другая, более оптималь-

ная опция — воспользоваться тем фактом, что две соседние подстроки длины m в тексте имеют общую подпоследовательность длиной $m - 1$. Использование кольцевого хеша позволяет учесть эту особенность и вычислять значение хеш-функции за время, не зависящее от длины паттерна. В данной работе в качестве кольцевого хеша использовался полиномиальный хеш:

$$h(s_0s_1\dots s_{m-1}) = (s_0p^{m-1} + s_1p^{m-2} + \dots + s_{m-1}p^0) \bmod q, \quad (1)$$

где $s_0s_1\dots s_{m-1}$ — подстрока длины m , p и q — параметры, \bmod — операция взятия остатка. Таким образом:

$$h(s_{i+1}s_{i+2}\dots s_{i+m}) = (h(s_i s_{i+1}\dots s_{i+m-1})p - s_i p^m + s_{i+m}) \bmod q. \quad (2)$$

Для того чтобы операции умножения и взятия остатка можно было заменить на побитовые операции и получить преимущество в скорости, в качестве параметров были выбраны значения $p = 2$ и $q = 2^{31} - 1$.

2.3. Алгоритм Hash3. Алгоритм Hash3 [13] продолжает идею алгоритма Рабина-Карпа по отсечению как можно большего количества посимвольных сравнений строк. Для искомого паттерна предварительно вычисляется расстояние от каждой его трехбуквенной подстроки до конца паттерна. Это расстояние записывается в хеш-таблицу `shift`, ключом в которой является значение хеш-функции трехбуквенной подстроки. Для вычисления хеш-функции трехбуквенной подстроки используется полиномиальный хеш (формула 1). Алгоритм Hash3 на каждой итерации вычисляет хеш-функцию текущей трехбуквенной подстроки текста и производит сдвиг на значение, сохраненное в хеш-таблице. Таким образом, алгоритм перемещается по памяти не последовательно, постоянно пытаясь перейти к концу предполагаемого вхождения паттерна в исходный текст. Это значительно уменьшает требуемое количество посимвольных сравнений.

Алгоритм в исходном виде, представленный в работе [13], был оптимизирован, исходя из специфики рассматриваемой задачи (листинг 4). Поскольку алфавит состоит всего из 4 символов, и всевозможных комбинаций из трех букв существует только 64, то вместо хеш-таблицы `shift` используется массив с индексами от 0 до 63. Для оптимизации работы на GPU данный массив выделяется в локальной памяти ядра.

Листинг 4. Псевдокод алгоритма Hash3.

```
int Hash3(text: string, pattern : string, n : int, m : int):
    int shift_size = 64
    shift : array [0 .. shift_size - 1] of int
    for ind = 0 to shift_size - 1:
        shift[ind] = m
    for j = 2 to (m - 1):
        shift[hash(pattern[j - 2 : j])] = m - 1 - j
    int answer = 0
    int j = m - 1
    while j < n:
        int temp_shift = 1
        while temp_shift != 0 and j < size:
            temp_shift = shift[hash(text[j - 2 : j])]
            j += temp_shift
```

```

    if j < n ans text[j - m + 1 : j] == pattern:
        answer += 1
        j += 1
return answer

```

С точки зрения производительности данный алгоритм имеет некоторые особенности по сравнению с рассмотренными ранее наивным алгоритмом и алгоритмом Рабина-Карпа. Во-первых, большое количество ветвлений может ухудшать работу на GPU. Это связано с тем, что графические ускорители основаны на параллельной архитектуре, где большое количество ядер одновременно выполняет одну и ту же инструкцию на разных данных. В случае ветвления кода те ядра, для которых не срабатывает условие, простаивают, что снижает эффективность параллельного выполнения. Во-вторых, непоследовательное перемещение по памяти приводит к тому, что производительность кода сильно ограничена пропускной способностью памяти как на CPU, так и на GPU.

3. Программная реализация. Для каждого из рассмотренных в разделе 2 алгоритмов было подготовлено три программных реализации. Одна из них является нативной для CPU и использует технологию OpenMP для организации параллелизма. Вторая предназначена для запусков только на GPU и написана на языке Cuda. Третья реализация, разработанная с использованием Kokkos, является универсальной и может быть запущена как на CPU, так и на GPU.

Параллелизм в версии OpenMP для CPU выполняется тривиальным образом: задачи распределяются по итерациям внешнего цикла (листинг 1) согласно статическому расписанию, т. е. используется стандартная OpenMP директива `#pragma omp parallel for`. Ядро в версии Cuda реализует один из алгоритмов поиска количества вхождений паттерна в текст (функция `patternCount` листинга 1). В качестве аргументов ядро принимает массив частот, исходный текст, длину текста и искомого паттерна. Алгоритмы реализованы тривиальным образом, специальных оптимизаций под GPU произведено не было.

Реализация алгоритмов на Kokkos универсальна и может работать как на CPU, так и на GPU. Также Kokkos выполняет ряд оптимизаций под архитектуру, на которой производится запуск. Для CPU выполняется автоматическая векторизация, выбирается оптимальное число потоков. На GPU автоматически определяется число блоков Cuda и потоков в блоке. Кроме того, Kokkos может определять оптимальную структуру хранения многомерного массива в зависимости от архитектуры. Например, в связи с особенностями устройства памяти матрицы на CPU обычно выгодно хранить по строкам, а на GPU по столбцам, и Kokkos это учитывает. Для инкапсуляции подобного рода деталей Kokkos предоставляет для хранения многомерных статических или динамических массивов специальную структуру данных `View`, которая при объявлении принимает в качестве аргумента шаблона информацию об аппаратном окружении.

Для написания версии Kokkos применительно к рассматриваемой задаче требовалось представить массив частот и исходный текст как два объекта типа `View`. Для организации параллелизма используется синтаксис `Kokkos parallel_for` (листинг 5). Ядро параллельной программы на Kokkos практически идентично ядру в версии Cuda, за исключением некоторых незначительных синтаксических особенностей.

Листинг 5. Фрагмент программы на Kokkos.

```

Kokkos::View<int*, Kokkos::SharedSpace> freq("frequency", size);
Kokkos::View<const char*, Kokkos::SharedSpace> data("txt", size);

```

```

Kokkos::parallel_for("pattern_search", size-len+1,
  KOKKOS_LAMBDA (int i) { // algorithm kernel }
);

```

Программная реализация находится в открытом доступе [19].

4. Эксперименты. 4.1. *Тестовая конфигурация.* Все запуски проводились на кластере ННГУ «Лобачевский». Запуски на CPU производились на следующей конфигурации:

- 2 процессора Intel Sandy Bridge E5-2660 2.2 GHz (8 ядер);
- 64 ГБ оперативной памяти;
- Компилятор Intel DPC++ Compiler 2023.0.0;
- OpenMP 5.0;
- Kokkos 4.0.1.

Запуски на GPU производились на видеокарте NVidia A100:

- 40 ГБ видеопамати (пропускная способность 1555 ГБ/с);
- Частота ГП 1410 МГц;
- Версия Cuda 11.3;
- Kokkos 4.0.1.

На вход алгоритмам подается представленный в виде строки фрагмент незакодированной ДНК длиной $n \sim 5 \cdot 10^5$, а также длина искомого паттерна m ($4 \lesssim m \lesssim 1024$). В качестве тестовых данных выступают реальные фрагменты ДНК, взятые из открытых источников [20].

4.2. *Результаты экспериментов.* В процессе работы было разработано несколько версий кода: реализация рассматриваемых алгоритмов для CPU на C++ с использованием OpenMP и векторных вычислений; реализация для GPU на языке Cuda; кроссплатформенная реализация с использованием библиотеки Kokkos. На рис. 1, 2 представлены результаты замеров времени для каждой версии кода. Можно видеть, что производительность версии Kokkos отличается от производительности версии OpenMP на CPU или версии Cuda на GPU не более чем на 5–10 %.

На рис. 1 представлен анализ производительности на CPU. Можно заметить, что алгоритм Hash3 работает гораздо быстрее алгоритмов Рабина-Карпа и наивного алгоритма. Алгоритм Рабина-Карпа и Hash3 не понесли потери производительности при переходе на Kokkos. Наивный алгоритм при переходе на Kokkos стал работать медленнее примерно на 10 %. Это связано с тем, что в реализации OpenMP компилятор более эффективно задействовал векторные вычисления при посимвольном сравнении паттернов.

Все алгоритмы ускорились примерно в 14 раз при запуске на 16 потоках, что является достаточно хорошим результатом. Однако можно заметить, что алгоритм Hash3, в отличие от алгоритма Рабина-Карпа и наивного алгоритма, получил также существенный прирост производительности при задействовании гипертрединга. Наивный алгоритм и алгоритм Рабина-Карпа на 16 физических ядрах и 32 потоках демонстрируют ускорение 16–17 раз, в то время как ускорение алгоритма Hash3 составляет примерно 25 раз. Это связано с тем, что выделение большего количества потоков позволяет создавать большее число запросов к памяти, в результате чего шина ОЗУ начинает использоваться активнее.

Поскольку в алгоритме Hash3 практически не выполняются посимвольные сравнения и все основное время занимает подгрузка данных из памяти, то, как и ожидалось, при переходе на GPU алгоритм Рабина-Карпа и наивный алгоритм получили значительно большее ускорение (в 51 и 34 раза соответственно), чем Hash3 (примерно в 7 раз). Мож-

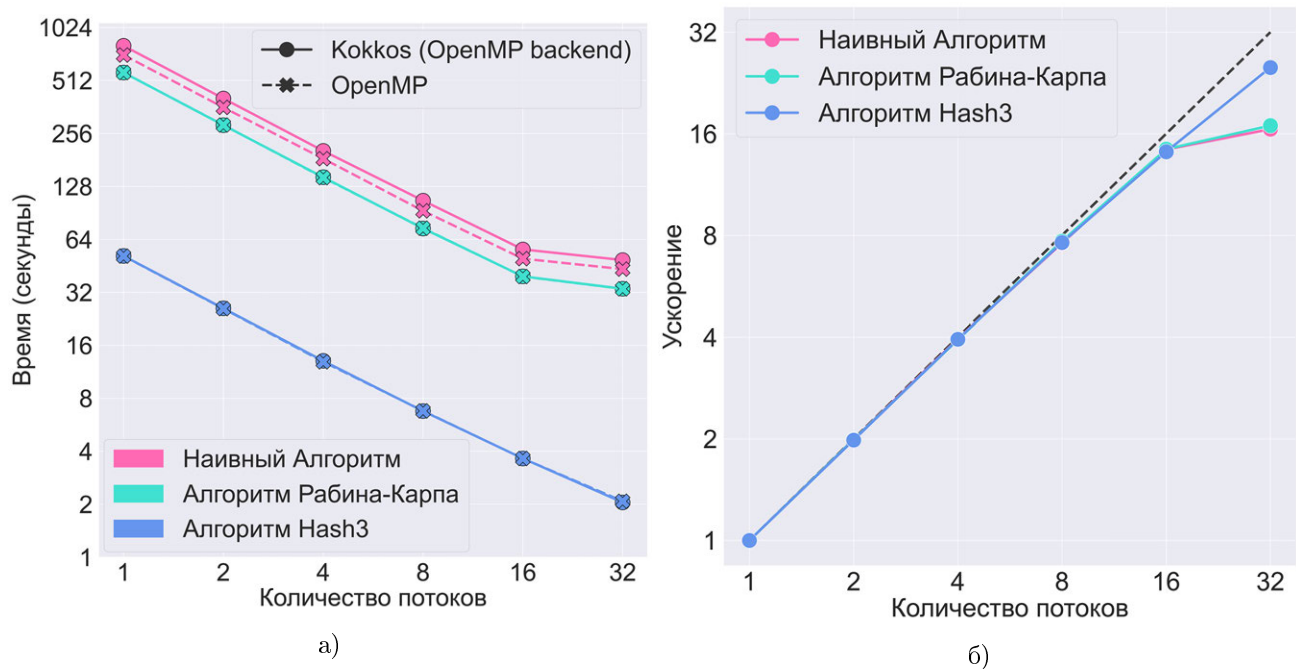


Рис. 1. Результаты тестирования алгоритмов на CPU ($m=64$). (а) Время работы в зависимости от числа потоков версий OpenMP и Kokkos с бэкэндом OpenMP. (б) Ускорение для версии Kokkos. На числе потоков до 16 включительно один поток соответствует одному физическому ядру; 32 потока соответствуют 16 физическим ядрам с включенным гипертредингом

но заметить, что производительность алгоритма Hash3 зависит от длины искомой строки (рис. 2). На коротких паттернах не удастся достичь отсечения большого количества элементарных сравнений строк, поэтому на малых длинах паттернов алгоритм показывает меньшую эффективность, чем на больших длинах паттернов. В то же время производительность наивного алгоритма и алгоритма Рабина-Карпа не зависит от длины паттерна, поэтому они показывают более хорошую производительность на коротких паттернах, чем Hash3.

Результаты тестирования на GPU также показали, что реализации рассматриваемых алгоритмов, написанные с помощью Kokkos, не теряют в производительности относительно нативных решений. Более того, версия программы с использованием Kokkos иногда работает на GPU быстрее, чем базовая реализация того же кода на Cuda (рис. 2). Вероятно, Kokkos распределяет задачи между ядрами GPU более эффективно, чем это было сделано авторами вручную при написании кода на Cuda. Авторы уверены, что код на CUDA можно оптимизировать так, чтобы версия Cuda работала не медленнее, чем версия Kokkos, однако это требует отдельного исследования, и авторы не ставили себе подобной задачи. Тем не менее, заслуживает внимания тот факт, что Kokkos выполняет ряд оптимизаций для GPU самостоятельно без участия программиста.

Заключение. На основе задачи из биоинформатики по поиску наиболее часто встречающихся паттернов небольшой длины во фрагменте ДНК был разработан бенчмарк, демонстрирующий работу рассматриваемых в работе алгоритмов на CPU и GPU с использованием технологий OpenMP и Cuda, а также библиотеки Kokkos. Было продемонстрировано, что во многих случаях версия Kokkos работает не хуже, чем версия

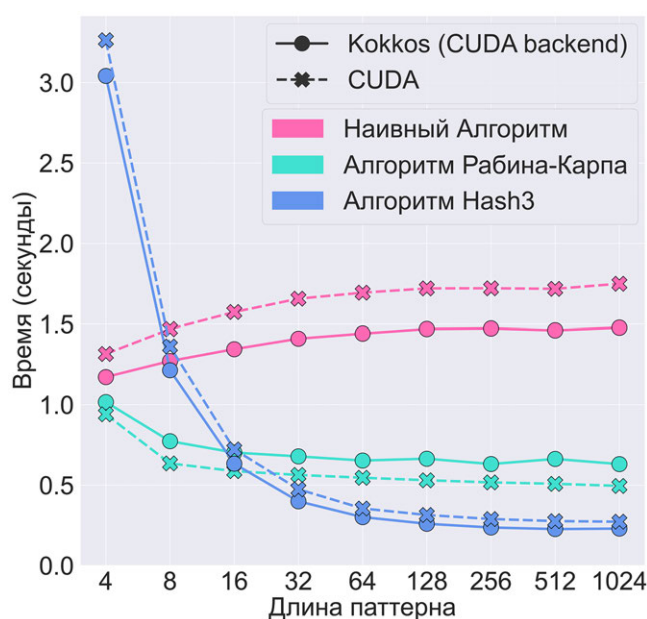


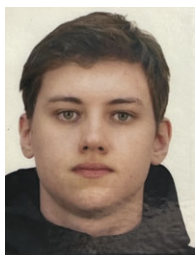
Рис. 2. Результаты тестирования алгоритмов на GPU. Зависимость времени работы алгоритмов от длины паттерна m для версий CUDA и Kokkos с бэкендом CUDA

кода при использовании OpenMP или Cuda, при этом версия с использованием Kokkos может быть запущена на устройствах различной архитектуры. В худшем случае потери производительности составили не более 10 %. Код бенчмарка находится в открытом доступе на платформе GitHub [19].

Список литературы

1. Gaster B., Howes L., Kaeli D. R., Mistry P., and Schaa D. Heterogeneous computing with openCL: revised openCL. Newnes, 2012.
2. Farber R. Parallel programming with OpenACC. Newnes, 2016.
3. Kokkos 3: Programming model extensions for the exascale era / Trott C. R., Damien LG, Arndt D., Ciesko J., Dang V., Ellingwood N., Gayatri R., Harvey E., Hollman D. S., Ibanez D., et al. // IEEE Transactions on Parallel and Distributed Systems. 2021. V. 33, N 4. P. 805–817.
4. Alpaka — an abstraction library for parallel kernel acceleration / Zenker E., Worpitz B., Widera R., Huebl A., Juckeland G., Knupfer A., Nagel W. E., and Bussmann M. // 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) / IEEE. 2016. P. 631–640.
5. Reinders J. et al. Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Springer Nature, 2021. P. 548.
6. Библиотека Kokkos. [Электрон. рес.]: <https://github.com/kokkos/kokkos>. Дата доступа: 10.01.2024.
7. Subirana J. A., Messeguer X. The most frequent short sequences in non-coding DNA // Nucleic acids research. 2010. V. 38, N 4. P. 1172–1181.
8. Faro S., Lecroq T. The exact online string matching problem: A review of the most recent results // ACM Computing Surveys(CSUR). 2013. V. 45, N 2. P. 1–42.
9. Exact string matching algorithms: survey, issues, and future research directions / Hakak S. I., Kamsin A., Shivakumara P., Gilkar G. A., Khan W. Z., and Imran M. // IEEE access. 2019. V. 7. P. 69614–69637.

10. Stephen G. A. String searching algorithms. World Scientific, 1994.
11. Al-Khamaiseh K., Alshagarin S. A survey of string matching algorithms // Int. J. Eng. Res. Appl. 2014. V. 4, N 7. P. 144–156.
12. Karp R. M., Rabin M. O. Efficient randomized patternmatching algorithms // IBM Journal of Research and Development. 1987. V. 31, N 2. P. 249–260.
13. Lecroq T. Fast exact string matching algorithms // Information Processing Letters. 2007. V. 102, N 6. P. 229–235.
14. Galil Z. A constant-time optimal parallel string-matching algorithm // Journal of the ACM (JACM). 1995. V. 42, N 4. P. 908–918.
15. Park J. H., George K. M. Efficient parallel hardware algorithms for string matching // Microprocessors and Microsystems. 1999. V. 23, N 3. P. 155–168.
16. Accelerating string matching using multi-threaded algorithm on GPU / Lin C. H., Tsai S. Y., Liu C. H., Chang S. C., and Shyu J. M. // 2010 IEEE Global Telecommunications Conference GLOBECOM 2010 / IEEE. 2010. P. 1–5.
17. Kouzinopoulos C. S., Michailidis P. D., Margaritis K. G. Multiple string matching on a GPU using cuda // Scalable Computing: Practice and Experience. 2015. V. 16, N 2. P. 121–138.
18. Козлов М. А., Панова Е. А., Мееров И. Б. Реализация поиска наиболее часто встречающихся последовательностей ДНК с использованием библиотеки Kokkos // Математическое моделирование и суперкомпьютерные технологии. Труды XXIII Международной конференции (Н. Новгород, 13–16 ноября 2023 г.) / Под ред. проф. Д. В. Баландина. Нижний Новгород: Изд-во Нижегородского государственного университета, 2023. ISBN 978-5-91326-834-1. 2023. С. 73–78.
19. Открытый исходный код бенчмарка. [Электрон. рес.]: https://github.com/Mishaizlesa/most_common_string_kokkos. Дата доступа: 10.01.2024.
20. DNA Bank (National library of medicine). [Электрон. рес.]: <https://www.ncbi.nlm.nih.gov/genbank>. Дата доступа: 10.01.2024.



Козлов Михаил Андреевич — студент 3 курса кафедры Высокопроизводительных вычислений и системного программирования Института информационных технологий, математики и механики ННГУ им. Н. И. Лобачевского.

E-mail: misha11ru@gmail.com.

Область научных интересов: высокопроизводительные вычисления, строковые алгоритмы.

Kozlov Mikhail — 3rd year student at the Department of HPC and System Programming at the Lobachevsky State University of Nizhny Novgorod. Research interests: HPC, string matching algorithms.



Панова Елена Анатольевна — аспирантка кафедры Высокопроизводительных вычислений и системного программирования Института ин-

формационных технологий, математики и механики ННГУ им. Н. И. Лобачевского. E-mail: elena.panova@itmm.unn.ru.

Ее исследовательские интересы включают разработку научного программного обеспечения, высокопроизводительные вычисления, анализ производительности, оптимизацию программ.

Elena Panova — Ph.D. student at the Department of HPC and System Programming at the Lobachevsky State University of Nizhny Novgorod. Her research interests include scientific software development, HPC, performance analysis, optimization.



Мееров Иосиф Борисович — канд. техн. наук, доцент, заведующий кафедрой Высокопроизводительных вычислений и системного программирования Института информационных технологий, математики и механики НН-

ГУ им. Н. И. Лобачевского. E-mail: `meerov@vmk.unn.ru`.

Научные интересы включают высокопроизводительные вычисления, анализ производительности и оптимизацию программ, вычислительные науки, разработку научного программного обеспечения.

Iosif Meyerov — Ph.D., head of Department of HPC and System programming at the Lobachevsky State University of Nizhny Novgorod. Research interests include HPC, performance analysis and optimization, computational science, scientific software development.

Дата поступления — 31.01.2024