# Параллельный алгоритм поиска лейтмотивов временного ряда для графического процессора*

М.Л. Цымблер, Я.А. Краева

Южно-Уральский государственный университет

Лейтмотив представляет собой пару подпоследовательностей временного ряда, наиболее похожих друг на друга. Задача поиска лейтмотивов встречается в широком спектре предметных областей: медицина, биология, предсказание погоды и др. В работе предложен новый параллельный алгоритм поиска лейтмотива во временном ряде на платформе графического процессора для случая, когда входные данные могут быть размещены в оперативной памяти. Распараллеливание выполнено с помощью технологии программирования OpenACC. Представлены результаты вычислительных экспериментов, подтверждающих масштабируемость разработанного алгоритма.

*Ключевые слова:* временной ряд, поиск лейтмотивов, параллельный алгоритм, NVIDIA GPU, OpenACC.

## 1. Введение

*Лейтмотив временного ряда* представляет собой пару подпоследовательностей этого ряда заданной длины, наиболее похожих друг на друга [18]. В настоящее время поиск лейтмотивов является актуальной задачей в широком спектре приложений обработки временных рядов: биоинформатика [2], обработка речи [1], прогнозирование природных катаклизмов [10], неврология [13] и др.

Поиск лейтмотива методом полного перебора, очевидно, имеет временную сложность $O(n^2)$, где $n$ — длина временного ряда. В силу этого в работах [4, 11, 12, 20, 21] был предложен ряд алгоритмов поиска *приближенного лейтмотива*, имеющих меньшую временную сложность $O(n)$ и $O(n \log n)$. Однако для ряда приложений, например, в сейсмологии [23], недопустима потеря точности результирующего лейтмотива, даже за счет выигрыша во времени поиска. Алгоритм MK [13] является одним из самых быстрых последовательных алгоритмов поиска *точного лейтмотива* и сокращает время поиска до трех раз по сравнению с другими алгоритмами, однако его производительность значительно снижается на временных рядах, имеющих длину от сотен тысяч элементов [13].

Одной из тенденций развития современной процессорной техники является увеличение количества вычислительных ядер вместо тактовой частоты [6]. В настоящее время ускорители архитектур Intel MIC (Many Integrated Core) [5] и NVIDIA GPU [16] обеспечивают от сотен до тысяч процессорных ядер и значительно опережают традиционные процессоры по производительности. В соответствии с этим перспективным направлением исследований является разработка параллельных алгоритмов поиска лейтмотивов временного ряда на ускорителях архитектур Intel MIC и NVIDIA GPU.

В настоящей статье предлагается новый параллельный алгоритм поиска лейтмотивов временного ряда на графическом процессоре (GPU) для случая, когда входные данные могут быть размещены в оперативной памяти. Данная статья продолжает исследование авторов, начатое в работе [24], где представлен параллельный алгоритм поиска лейтмотивов на ускорителях Intel MIC с помощью технологии параллельного программирования OpenMP [9]. В настоящем исследовании используются схожие базовые идеи, однако иная

*Работа выполнена при финансовой поддержке Министерства науки и высшего образования Российской Федерации в рамках федеральной целевой программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014–2020 годы», уникальный идентификатор проекта RFMEFI57818X0265 (контракт № 075-15-2019-1339 (14.578.21.0265)).

архитектура графического ускорителя и используемая в реализации соответствующая технология параллельного программирования OpenACC [7] требуют существенной переработки предложенных ранее технологических решений.

Статья организована следующим образом. В разделе 2 приводится формальная постановка задачи и кратко описан последовательный алгоритм МК. Раздел 3 содержит обзор работ по тематике исследования. В разделе 4 дано описание предлагаемого параллельного алгоритма. В разделе 5 представлены результаты вычислительных экспериментов по исследованию эффективности предложенного алгоритма. Заключение резюмирует результаты, полученные в рамках исследования.

## 2. Постановка задачи

### 2.1. Формальные определения и нотация

В данном разделе приводятся обозначения и определения используемых терминов в соответствии с работой [13].

*Временной ряд* представляет собой хронологически упорядоченную последовательность числовых значений: $T = (t_1, \ldots, t_n)$, $t_i \in \mathbb{R}$. Число $m$ обозначается $|T|$ и называется длиной временного ряда.

*Подпоследовательность* $T_{i,m}$ временного ряда $T$ представляет собой непрерывное подмножество $T$, состоящее из $m$ элементов и начинающееся с позиции $i$: $T_{i,m} = (t_i, \ldots, t_{i+m-1})$, $1 \leqslant i \leqslant n - m + 1,\ m \ll n$.

Множество всех подпоследовательностей ряда $T$, имеющих длину $m$, обозначается $S_T^m$. Мощность указанного множества обозначим как $N$, $N = |S_T^m| = n - m + 1$.

В качестве *функции расстояния* между подпоследовательностями ряда возьмем неотрицательную симметричную функцию $\mathrm{Dist} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$.

Пара подпоследовательностей $\{T_{i,m},\, T_{j,m}\}$ ряда $T$ называется *лейтмотивом (motif)*, если

$$\begin{gathered} \forall a, b, i, j\ \ \mathrm{Dist}(T_{i,m},\, T_{j,m}) \leqslant \mathrm{Dist}(T_{a,m},\, T_{b,m}), \\ |i - j| \geqslant w,\ |a - b| \geqslant w,\ w > 0, \end{gathered} \tag{1}$$

где $w$ — параметр, определяющий минимальный промежуток, на который должны отстоять друг от друга подпоследовательности в лейтмотиве. Данный параметр позволяет отбрасывать лейтмотивы, которые состоят из взаимопересекающихся подпоследовательностей и потому не имеют практической ценности [18].

В качестве функции расстояния между двумя подпоследовательностями $X$ и $Y$ нами будет использоваться *евклидово расстояние*, определяемое следующим образом:

$$\mathrm{ED}(X,\ Y) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}. \tag{2}$$

Для корректного определения схожести подпоследовательностей временного ряда, имеющих разные амплитуды, в дальнейшем изложении предполагается, что перед вычислением расстояния каждая подпоследовательность подвергается z-нормализации, обеспечивающей среднее арифметическое и стандартное отклонение элементов подпоследовательности, равные нулю и единице, соответственно. *Z-нормализация* подпоследовательности $C \in S_T^m$ представляет собой подпоследовательность $\widehat{C} = (\widehat{c}_1, \ldots, \widehat{c}_m)$, элементы которой вычисляются следующим образом:

$$\widehat{c}_i = \frac{c_i - \mu}{\sigma} :\ \mu = \frac{1}{m} \sum_{i=1}^{m} c_i,\ \sigma = \sqrt{\frac{1}{m} \sum_{i=1}^{m} c_i^2 - \mu^2}. \tag{3}$$

## 2.2. Последовательный алгоритм

Алгоритм MK [13] представляет собой один из самых быстрых последовательных алгоритмов для нахождения точного лейтмотива во временном ряде. Алгоритм сокращает пространство поиска лейтмотивов на основе введения т.н. опорных подпоследовательностей.

*Опорная подпоследовательность* представляет собой случайно выбранную подпоследовательность исходного ряда. Алгоритм вычисляет расстояние от каждой подпоследовательности ряда до опорной и упорядочивает все подпоследовательности в порядке возрастания вычисленных расстояний. Полученный порядок называется *линейным*. Затем используется следующее свойство: если два объекта конечномерного метрического пространства близки друг другу, то они также должны быть близки в линейном порядке (обратное утверждение неверно) [13]. В соответствии с неравенством треугольника расстояние между подпоследовательностями лейтмотива в линейном порядке является нижней границей расстояния между этими подпоследовательностями в пространстве $\mathbb{R}^m$:

$$\mathrm{ED}(ref,\, T_{i,\,m}) - \mathrm{ED}(ref,\, T_{j,\,m}) \leqslant \mathrm{ED}(T_{i,\,m},\, T_{j,\,m}),\ |i-j| \geqslant w,\ w > 0, \tag{4}$$

где $ref$ — опорная подпоследовательность, $\{T_{i,\,m},\, T_{j,\,m}\}$ — пара подпоследовательностей, отличных от $ref$. Далее для того, чтобы различать два вышеупомянутых вида расстояний, расстояние между подпоследовательностями в пространстве $\mathbb{R}^m$ мы будем называть *истинным расстоянием*.

Переменная алгоритма $bsf$ *(best-so-far)* представляет собой текущее минимальное истинное расстояние между подпоследовательностями лейтмотива, и обновляется алгоритмом, как только найдена пара подпоследовательностей, истинное расстояние между которыми меньше, чем $bsf$.

Просматривая подпоследовательности ряда в соответствии с их линейным порядком, алгоритм вычисляет нижние границы. Если нижняя граница превышает $bsf$, то истинное расстояние также превысит этот порог. Поэтому пара соответствующих подпоследовательностей заведомо не является лейтмотивом и может быть отброшена без вычисления истинного расстояния. Если пара подпоследовательностей не была отброшена, выполняется вычисление истинного расстояния. Если полученное значение меньше $bsf$, пороговое значение заменяется вычисленным.

Действуя таким образом, алгоритм просматривает все возможные пары подпоследовательностей, которые отстоят друг от друга в линейном порядке на величину $offsef$ $(1 \leqslant offset \leqslant N-1)$. Просмотр продолжается до тех пор, пока не будет достигнуто такое значение $offset$, для которого не существует пар подпоследовательностей, нижние границы которых больше, чем $bsf$, после чего алгоритм завершается.

Для получения более узких нижних границ, позволяющих отбрасывать больше бесперспективных пар подпоследовательностей, алгоритм использует более одной опорной подпоследовательности. Опорная подпоследовательность с наибольшим стандартным отклонением используется для сортировки по возрастанию расстояний между этой опорной подпоследовательностью и всеми прочими подпоследовательностями исходного ряда. Если хотя бы одна из нижних границ больше, чем $bsf$, то соответствующая пара подпоследовательностей отбрасывается. Алгоритм завершается, если все нижние границы всех пар подпоследовательностей, отстоящих друг от друга на величину $offset$, больше, чем $bsf$.

Число опорных подпоследовательностей берется существенно меньшим, чем количество подпоследовательностей в исходном ряде. Как показывают эксперименты [13], количество опорных подпоследовательностей в диапазоне от 5 до 60 обеспечивает стабильное сокращение времени поиска по сравнению с другими алгоритмами в 2–3 раза независимо от типа данных и длины временного ряда, а также длины искомого лейтмотива.

## 3. Обзор работ

Поскольку алгоритм поиска лейтмотива во временном ряде методом полного перебора имеет квадратичную временную сложность относительно длины временного ряда, в работах [4, 11, 12, 20, 21] были предложены алгоритмы поиска приближенного лейтмотива во временном ряде, которые имеют линейную или логарифмическую сложность.

В работе [22] Вилсон и др. представили алгоритм FLAME для нахождения точного лейтмотива в ДНК. Однако алгоритм FLAME является приближенным для дискретных временных рядов, значения которых представляют собой вещественные числа.

В работе [13] Муином, Кеогом и др. предложен алгоритм MK, который находит точный лейтмотив во временном ряде. Алгоритм MK основан на следующей идее. Рассмотрим подпоследовательности исходного ряда как конечное множество точек конечномерного метрического пространства. Выберем случайным образом некую точку данного множества и назовем ее опорной. Упорядочим точки исходного множества по возрастанию их расстояния до опорной точки и назовем такой порядок линейным. Для любой пары точек, отличных от опорной, верно следующее утверждение: если точки находятся близко друг к другу в исходном пространстве, то они также будут располагаться близко в линейном порядке (обратное утверждение неверно). Данное свойство вкупе с неравенством треугольника позволяет отбрасывать пары подпоследовательностей, заведомо не являющихся лейтмотивом, без вычисления расстояния. Для большего сокращения пространства поиска алгоритм использует несколько опорных точек. Эксперименты показывают, что алгоритм MK позволяет в разы ускорить поиск лейтмотива полным перебором [13].

В работе [15] Наранг и др. предложили параллельный алгоритм Par-MK для точного обнаружения лейтмотива на SMP системах для случая, когда данные полностью помещаются в оперативную память. Для распараллеливания Par-MK использует нити стандарта POSIX [17]. Каждая нить обрабатывает часть отсортированного массива, содержащего расстояния до опорной подпоследовательности. Алгоритм также распараллеливает цикл просмотра пар подпоследовательностей, отстоящих друг от друга в линейном порядке, выполняемый по величине этого отступа. Нити, которые обрабатывают разные значения отступа, но одну и ту же часть массива расстояний, выполняются на ядрах, которые разделяют кэш второго уровня.

Для устранения дисбаланса загрузки нитей из-за непредсказуемого и неравномерного количества подпоследовательностей, отбрасываемых различными нитями, авторы предложили две версии своего алгоритма: Par-MK-SLB и Par-MK-DLB (соответственно статическая и динамическая балансировка нагрузки). В вычислительных экспериментах на реальном временном ряде длины $|T| = 1.8 \cdot 10^5$ при длине лейтмотива $m = 200$, лучшая версия алгоритма продемонстрировала сверхлинейное ускорение на 32 ядрах благодаря динамической балансировке нагрузки в сочетании с превосходной производительностью кэша второго уровня. Однако на синтетическом временном ряде длины $|T| = 5 \cdot 10^4$ при длине лейтмотива $m = 1024$ ускорение лучшей версии уменьшилось более чем в 8.5 раза из-за снижения производительности кэша за счет большей длины лейтмотива. В своей дальнейшей работе авторы планировали модернизировать алгоритм для многоядерных ускорителей, но это исследование не было проведено.

В работе [24] авторами настоящей статьи представлен параллельный алгоритм поиска лейтмотивов на ускорителях Intel MIC для случая, когда входные данные могут быть размещены в оперативной памяти. Алгоритм использует набор матрично-векторных структур данных для хранения и индексации временного ряда и лейтмотивов, обеспечивающих эффективную векторизацию вычислений на платформе Intel MIC. Реализация выполнена на основе технологии параллельного программирования OpenMP [9]. Эксперименты показали высокую масштабируемость параллельного алгоритма и его более высокую производительность при исполнении на многоядерном ускорителе, чем на узле, состоящем из двух многоядерных процессоров Intel. Механический перенос данной разработки на графический уско-

ритель, однако, невозможен, как в силу существенных различий как между архитектурами Intel MIC и GPU, так и ввиду отсутствия реализации технологии параллельного программирования OpenMP для GPU. Технология параллельного программирования OpenACC [7], хотя и является идеологическим аналогом OpenMP для GPU, помимо разницы в синтаксисе, имеет ряд существенных семантических отличий, которые обуславливают переработку имеющегося решения.

# 4. Параллельный алгоритм поиска лейтмотивов для GPU

В данном разделе описан подход к распараллеливанию описанного выше алгоритма МК [13] для графического процессора. Ниже в разделе 4.1 изложены особенности данной аппаратной платформы и применяемой технологии параллельного программирования. В разделах 4.2 и 4.3 приводится описание структур данных и реализации параллельного алгоритма соответственно.

## 4.1. Аппаратно-программная платформа

Графический процессор (GPU) компании NVIDIA [16] представляет собой один из наиболее популярных в настоящее время многоядерных ускорителей. GPU имеет иерархическую архитектуру и состоит из симметричных потоковых мультипроцессоров (Streaming Multiprocessor, SM), каждый из которых, в свою очередь, состоит из симметричных CUDA-ядер. Современные GPU насчитывают тысячи CUDA-ядер, способных опередить по производительности центральные процессоры на задачах, допускающих массивно-параллельные вычисления в сочетании с векторной обработкой данных.

Параллельное приложение запускается на GPU как набор нитей, где каждая нить исполняется отдельным CUDA-ядром и предусмотрена следующая иерархия нитей. Верхним уровнем иерархии, соответствующим всем нитям, является *сетка нитей (grid)*, которая состоит из одномерного или двумерного массива симметричных блоков нитей. *Блок нитей (thread block)* представляет собой $d$-мерный ($1 \leqslant d \leqslant 3$) массив нитей. Внутри блока нити логически разделяются на группы по 32 нити — *варпы (warp)*. Нити варпа исполняются в режиме *SIMT (Single Instruction Multiple Threads)*, когда каждая нить выполняет одну и ту же инструкцию над собственной порцией общих данных.

При запуске приложения блоки нитей распределяются для исполнения между потоковыми мультипроцессорами и выполняются далее параллельно без возможности их синхронизации. Нити в пределах блока допускают синхронизацию и имеют доступ к разделяемой памяти, отведенной для данного блока. Для передачи данных между потоками, принадлежащих разным блокам, используется глобальная память ускорителя.

В настоящее время для графических процессоров разработаны технологии параллельного программирования CUDA [3], OpenCL [14] и OpenACC [7], последняя из которых применена в данном исследовании. *OpenACC* представляет собой открытый стандарт параллельного программирования для создания гетерогенных параллельных программ, задействующих как центральный, так и графический процессоры. Стандарт включает библиотеку функций, переменные окружения и директивы компилятора для определения участков исходного кода программы, которые необходимо выполнить на графическом процессоре.

Модель исполнения OpenACC-приложения предусматривает иерархию нитей и соответствующие уровни параллелизма. Одна или более нитей составляют *бригаду (gang)*, исполняемую на одном потоковом мультипроцессоре. В рамках бригады определяется одна или более симметричных групп нитей — *рабочих (worker)*. Внутри рабочего нити исполняются в режиме SIMT, обеспечивая еще один, *векторный (vector)* уровень параллелизма.

Одной из основных директив компилятора в технологии OpenACC является `#pragma acc parallel loop`, которая распараллеливает цикл с фиксированным количеством повторений, равномерно распределяя итерации цикла между нитями бригад для исполнения (при

отсутствии между итерациями цикла зависимостей по данным). Указанная директива может быть дополнена одним или несколькими ключевыми словами `gang`, `worker`, `vector`, которые обяжут компилятор применить в данном цикле соответствующие уровни параллелизма.

### 4.2. Реализация структур данных

Для хранения данных в оперативной памяти графического процессора предлагаемый алгоритм использует матрицы и массивы, что обеспечивает возможность векторизации распараллеливаемых вычислений.

Пусть варп графического процессора состоит из $size_{warp}$ нитей. Если длина искомого лейтмотива $m$ не кратна $size_{warp}$, то подпоследовательность выравнивается посредством дополнения фиктивными нулевыми элементами. Выравнивание данных позволяет избежать простаивания нитей в варпе, снижающего производительность вычислений. Обозначим количество фиктивных элементов через $pad = size_{warp} - (m \bmod size_{warp})$, тогда выровненная подпоследовательность $\tilde{T}_{i,\,m}$ определяется следующим образом:

$$\tilde{T}_{i,\,m} = \begin{cases} t_i, t_{i+1}, \ldots, t_{i+m-1}, \underbrace{0, \ldots, 0}_{pad}, & if\ m \bmod size_{warp} > 0 \\ t_i, t_{i+1}, \ldots, t_{i+m-1}, & otherwise. \end{cases} \tag{5}$$

Все выровненные подпоследовательности ряда хранятся в *матрице подпоследовательностей* $S_T^m \in \mathbb{R}^{N \times (m+pad)}$, которая определяется следующим образом:

$$S_T^m(i,j) = \tilde{t}_{i+j-1}. \tag{6}$$

Обозначим количество опорных подпоследовательностей за $r$ $(0 < r \ll N)$. Тогда *матрица опорных подпоследовательностей* $Ref \in \mathbb{R}^{r \times (m+pad)}$ определяется следующим образом:

$$Ref(i,\cdot) = T_{i_k,\,m} :\ T_{i_k,\,m} \in S_T^m,\ 1 \leqslant k \leqslant r,\ i_k = random(1..N),\ \forall p \neq q\ i_p \neq i_q. \tag{7}$$

*Индекс опорных подпоследовательностей* $I_{Ref} \in \mathbb{N}^r$ для каждой опорной подпоследовательности хранит позицию ее начала во временном ряде.

*Матрица расстояний* $D \in \mathbb{R}^{r \times N}$ предназначена для хранения истинного расстояния между каждой опорной подпоследовательностью и каждой подпоследовательностью ряда и определяется следующим образом:

$$D(i,j) = \mathrm{ED}(T_{I_{Ref}(i),\,m}, T_{j,\,m}). \tag{8}$$

*Вектор стандартных отклонений* представляет собой массив $SD \in \mathbb{R}^r$, который для каждой опорной подпоследовательности содержит стандартное отклонение ее истинного расстояния до каждой подпоследовательности ряда.

*Индекс стандартных отклонений* представляет собой массив $I_{SD} \in \mathbb{N}^r$, содержащий уникальные числа в диапазоне от 1 до $r$, где числа соответствуют номерам опорных подпоследовательностей в $Ref$, упорядоченных по убыванию их стандартного отклонения.

*Индекс подпоследовательностей* $I_S \in \mathbb{N}^N$ представляет собой массив, который содержит позиции подпоследовательностей в матрице $S_T^m$, упорядоченные по возрастанию их истинных расстояний до опорной подпоследовательности, имеющей наибольшее стандартное отклонение.

*Индекс лейтмотива* $I_M \in \mathbb{N}^{N \times 2}$ предназначен для хранения позиций двух подпоследовательностей в $S_T^m$, которые являются потенциальным лейтмотивом и в индексе подпоследовательностей $I_S$ отстоят друг от друга на величину $offset$.

*Матрица нижних границ* $LB \in \mathbb{R}^{r \times N}$ содержит значения нижних границ между каждой опорной подпоследовательностью и каждым возможным лейтмотивом и в соответствии с (4) вычисляется следующим образом:

$$LB(i,j) = |D\big(I_{SD}(i), I_M(j,1)\big) - D\big(I_{SD}(i), I_M(j,2)\big)|. \tag{9}$$

*Битовая карта* представляет собой массив $B \in \mathbb{B}^N$, в котором для каждого потенциального лейтмотива хранится результат конъюнкции проверок превышения текущим значением порога $bsf$ каждой из нижних границ. Если элемент битовой карты равен `FALSE`, то соответствующий лейтмотив отбрасывается без вычисления истинного расстояния между парой подпоследовательностей. Битовая карта определяется следующим образом:

$$B(i) = \wedge_{j=1}^{r}(LB(i,j) < bsf). \tag{10}$$

### 4.3. Реализация алгоритма

**Алг. 1** MotifDiscovery(in $T$, $m$, $w$, $r$; out $motif$)

▷ **Инициализация**

1: $Ref \leftarrow r$ случайно выбранных подпоследовательностей из $S_T^m$
2: $I_{Ref} \leftarrow r$ индексов элементов в $Ref$
3: #pragma acc data create($LB$, $SD$, $B$, $I_M$) copyin($S_T^m$, $D$, $I_{Ref}$, $I_S$, $N$, $m$, $w$, $r$) copyout($motif$)
4: {

▷ **Фаза предварительной обработки данных**

5: $S_T^m \leftarrow$ Znormalize($S_T^m$)
6: $D \leftarrow$ CalculateDistances($S_T^m$, $I_{Ref}$)
7: $SD \leftarrow$ CalculateStdDev($D$)
8: $I_{SD} \leftarrow$ Sort($SD$); $I_S \leftarrow$ Sort($D(I_{SD}(1), \cdot)$)
9: $bsf \leftarrow$ InitBSF($D$, $I_{Ref}$, $bsf$)
10: $I_M(\cdot, 1) \leftarrow$ GeneratePairs($I_S$, $offset$)

▷ **Фаза поиска лейтмотива**

11: **for all** $offset \in 1..N$ **do**
12: $abandon \leftarrow$ `FALSE`
13: $I_M(\cdot, 2) \leftarrow$ GeneratePairs($I_S$, $offset$)
14: $LB \leftarrow$ CalculateLowerBounds($D$, $I_{Ref}$, $I_M$, $bsf$)
15: $abandon \leftarrow$ Verify($LB$, $I_M$, $bsf$, $B$, $abandon$)
16: **if** $abandon$ **then**
17: **break**
18: **else**
19: $bsf \leftarrow$ UpdateBSF($S_T^m$, $B$, $I_M$, $bsf$, $motif$)
20: }
21: **return** $motif$

Предлагаемая параллельная реализация поиска лейтмотива временного ряда представлена в алг. 1. Входными данными алгоритма являются временной ряд $T$, длина искомого лейтмотива $m$, минимальный промежуток между подпоследовательностями лейтмотива $w$ и количество опорных подпоследовательностей $r$. Алгоритм возвращает найденный лейтмотив в виде кортежа, состоящего из двух подпоследовательностей и величины расстояния между ними.

Алгоритм выполняется следующим образом. Сначала на центральном процессоре формируются основные структуры данных: матрицы выровненных подпоследовательностей $S_T^m$ и опорных подпоследовательностей $Ref$, а также их индексы $I_S$ и $I_{Ref}$ соответственно, —

а затем с помощью директивы OpenACC `#pragma acc data` передает их на графический процессор. Затем выполняются фазы предварительной обработки данных и нахождения лейтмотива, рассматриваемые ниже в разделах 4.3.1 и 4.3.2 соответственно.

### *4.3.1. Предварительная обработка данных*

Фаза предварительной обработки данных предполагает z-нормализацию строк матрицы подпоследовательностей $S_T^m$ и вычисление на ее основе структур данных, описанных выше в разделе 4.2: матрица расстояний $D$, вектор стандартных отклонений $SD$ и индекс стандартных отклонений $I_{SD}$, а также вычисление начального значения порога $bsf$ и генерация индексов левых частей предполагаемых лейтмотивов $I_M(\cdot, 1)$.

**Алг. 2** ZNORMALIZE(IN OUT $S_T^m$)

```
1:  #pragma acc parallel loop gang
2:  for all i ∈ 1..N do
3:      mean ← 0; std ← 0
4:      #pragma acc loop vector reduction(+: mean, std)
5:      for all j ∈ 1..m do
6:          mean ← mean + S_T^m(i, j)
7:          std ← std + S_T^m(i, j)^2
8:      mean ← mean/m; std ← std/m; std ← sqrt(std − mean^2)
9:      #pragma acc loop vector
10:     for all j ∈ 1..m do
11:         S_T^m(i, j) ← (S_T^m(i, j) − mean)/std
```

Z-нормализация (см. алг. 2) реализуется с помощью двух вложенных циклов. Внешний цикл по строкам матрицы подпоследовательностей распараллеливается на уровне бригады с помощью директивы `#pragma acc parallel loop gang`. Внутренние циклы по элементам подпоследовательностей, выполняющие вычисление среднего арифметического и стандартного отклонения в соответствии с (3) и обновление элементов подпоследовательностей нормализованными значениями, распараллеливаются на уровне вектора с помощью директивы `#pragma acc parallel loop vector`.

**Алг. 3** CALCULATEDISTANCES(IN $S_T^m$, $I_{Ref}$; OUT $D$)

```
1:  #pragma acc parallel loop gang collapse(2)
2:  for all i ∈ 1..r do
3:      for all j ∈ 1..N do
4:          d ← 0
5:          #pragma acc loop vector reduction(+: d)
6:          for all k ∈ 1..m do
7:              d ← d + (S_T^m(j, k) − S_T^m(I_Ref(i), k))^2
8:          D(i, j) ← sqrt(d)
```

Вычисление матрицы расстояний (см. алг. 3) предполагает два вложенных цикла, внешний из которых выполняется параллельно бригадами нитей, а внутренний распараллеливается на векторном уровне. Поскольку циклы независимы и количество итераций во внутреннем цикле больше, чем во внешнем, используется стандартный прием, обеспечивающий бóльшую степень параллелизма: свертка указанных циклов в один, выполняемая добавлением в директиву распараллеливания внешнего цикла атрибута `collapse(2)`.

Вычисление вектора стандартных отклонений (см. алг. 4) организуется как два вложенных цикла: внешний — по опорным подпоследовательностям, внутренний — по элементам

**Алг. 4** CalculateStdDev(in $D$; out $SD$)

1: **for all** $i \in 1..r$ **do**
2: $mean \leftarrow 0$; $std \leftarrow 0$
3: #pragma acc parallel loop gang vector reduction(+: $mean$, $std$)
4: **for all** $j \in 1..N$ **do**
5: $mean \leftarrow mean + D(i,j)$
6: $std \leftarrow std + D(i,j)^2$
7: $mean \leftarrow \frac{mean}{N-1}$; $std \leftarrow \frac{std}{N-1}$
8: $SD(i) \leftarrow \sqrt{(std - mean^2)}$

матрицы расстояний. Поскольку количество опорных подпоследовательностей существенно меньше, чем количество нитей, запущенных на графическом процессоре (см. раздел 2.2), распараллеливанию подвергается только внутренний цикл, на бригадном и векторном уровнях.

**Алг. 5** InitBSF(in $D$, $I_{Ref}$; out $bsf$)

1: $bsf \leftarrow +\infty$
2: #pragma acc parallel loop gang vector collapse(2) reduction(min: $bsf$)
3: **for all** $i \in 1..r$ **do**
4: **for all** $j \in 1..N$ **do**
5: **if** $|j - I_{Ref}(i)| \geqslant w$ **then**
6: $bsf \leftarrow \min(bsf,\ D(i,j))$

В инициализации порога $bsf$ минимумом матрицы расстояний $D$ (см. алг. 5) применяется двухуровневое распараллеливание вычислений (бригадное и векторное), а также прием свертки циклов, рассмотренный выше.

### *4.3.2. Поиск лейтмотива*

Фаза поиска лейтмотива представляет собой просмотр пар подпоследовательностей, отстоящих друг от друга на $offset$ позиций и реализуется с помощью цикла по указанной переменной. На каждой итерации цикла индекс лейтмотива $I_M(\cdot, 2)$ заполняется индексами подпоследовательностей из правой части возможного лейтмотива, которые смещены на $offset$ позиций относительно индекса $I_M(\cdot, 1)$ в индексе $I_S$.

**Алг. 6** CalculateLowerBounds(in $D$, $I_{Ref}$, $I_M$; out $LB$)

1: #pragma acc parallel loop gang vector collapse(2)
2: **for all** $i \in 1..r$ **do**
3: **for all** $j \in 1..N - offset - 1$ **do**
4: $LB(i,j) \leftarrow |D\big(I_{Ref}(i), I_M(j,1)\big) - D\big(I_{Ref}(i), I_M(j,2)\big)|$

Затем в соответствии с неравенством треугольника осуществляется параллельное вычисление нижних границ потенциальных лейтмотивов и заполнение матрицы нижних границ $LB$ (см. алг. 6), выполняемые с помощью двухуровневое распараллеливания и свертки циклов.

Далее осуществляется проверка потенциальных лейтмотивов путем вычисления битовой карты и нахождение условия останова поиска лейтмотива *abandon* (см. алг. 7). Если дизъюнкция всех элементов битовой карты дает в результате `TRUE` (все нижние границы для всех пар подпоследовательностей, отстоящих друг от друга на $offset$ позиций, больше, чем текущее значение порога $bsf$), то результирующий лейтмотив найден, а остальные канди-

**Алг. 7** VERIFY(IN $LB$, $I_M$, $bsf$; IN OUT $B$, $abandon$)

```
1: #pragma acc parallel loop gang vector reduction(OR: abandon)
2: for all i ∈ 1..N − offset − 1 do
3:     B(i) ← TRUE
4:     if |I_M(i,1) − I_M(i,2)| ⩾ w then
5:         #pragma acc loop seq
6:         for all j ∈ 1..r do
7:             B(i) ← B(i) and (LB(j,i) < bsf)
8:         abandon ← abandon or B(i)
9: abandon ← not abandon
```

даты могут быть отброшены. Проверка реализуется посредством двух вложенных циклов: внешний — по потенциальным лейтмотивам и внутренний — по элементам матрицы нижних границ. Внешний цикл распараллеливается на уровнях бригад и вектора, а с помощью конструкции `reduction` обеспечивается нахождение условия останова поиска лейтмотива $abandon$. Поскольку каждая порождаемая внешним циклом нить вычисляет несколько элементов битовой карты, для корректного результата внутренний цикл должен исполняться последовательно, что обеспечивается директивой компилятора `#pragma acc parallel seq`.

**Алг. 8** UPDATEBSF(IN $S_T^m$, $B$, $I_M$; IN OUT $bsf$; OUT $motif$)

```
1: #pragma acc parallel loop gang worker reduction(min: bsf)
2: for all i ∈ 1..N − offset − 1 do
3:     if B(i) and |I_M(i,1) − I_M(i,2)| ⩾ w then
4:         d ← 0
5:         #pragma acc loop vector reduction(+: d)
6:         for all j ∈ 1..m do
7:             d ← d + (S_T^m(I_M(i,1), j) − S_T^m(I_M(i,2), j))^2
8:         if bsf > √d then
9:             bsf ← √d
10:            motif ← {I_M(i,1); I_M(i,2); bsf}
```

Если описанная выше процедура проверки не выявила лейтмотив, то выполняется обновление порога $bsf$ следующим образом (см. алг. 8). Вычисляется истинное расстояние между подпоследовательностями в каждом потенциальном лейтмотиве, для которого соответствующий элемент битовой карты равен `TRUE`. Порог обновляется вычисленным значением истинного расстояния, если оно меньше текущего значения $bsf$. Цикл, выполняющий вычисление истинного расстояния, распараллеливается на уровнях бригады и рабочего и дополнен конструкцией `reduction` для свертки операции нахождения минимума $bsf$ среди значений, вычисленных запущенными нитями.

## 5. Вычислительные эксперименты

Для исследования эффективности разработанного алгоритма нами были проведены вычислительные эксперименты на графическом процессоре NVIDIA GeForce RTX 2080 Ti[1] со следующими характеристиками: количество ядер — 4362 (68 мультипроцессоров), частота ядра — 1.35 ГГц, память — 11 Гб, пиковая производительность — 11 TFLOPS.

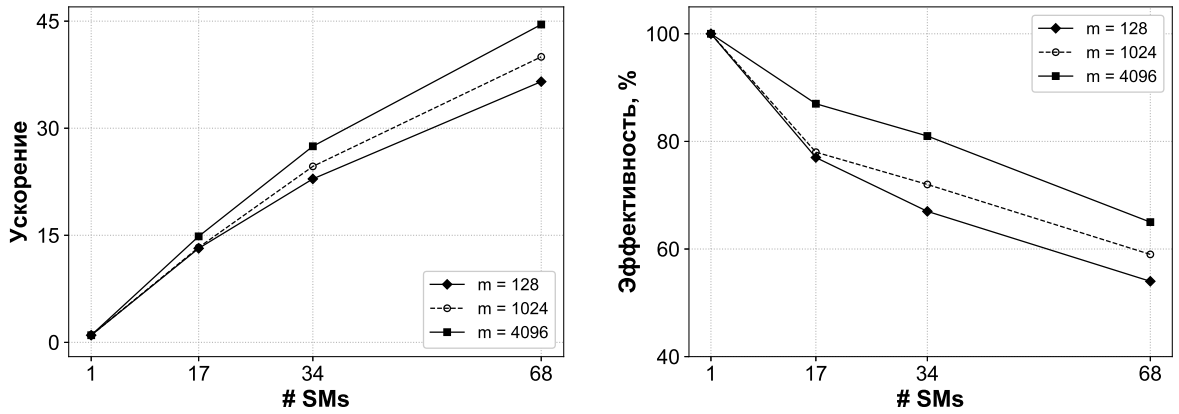
В экспериментах исследовались ускорение и параллельная эффективность алгоритма, понимаемые как его способность адекватно адаптироваться к увеличению параллельно ра-

[1] Спецификации видеокарты GeForce RTX 2080 Ti

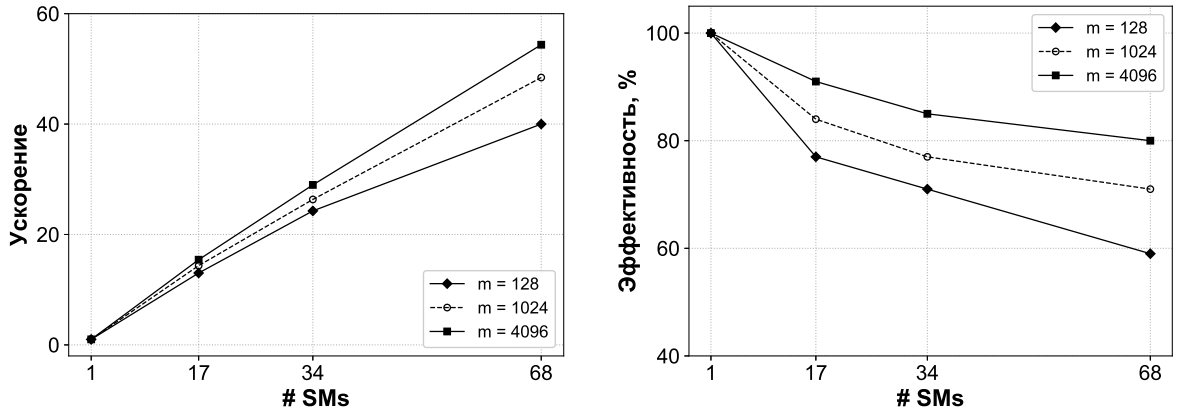
ботающих блоков мультипроцессоров графического ускорителя при варьируемой длине искомого лейтмотива.

Ускорение и параллельная эффективность параллельного алгоритма, запускаемого на $k$ блоках мультипроцессоров, вычисляются как $s(k) = \frac{t_1}{t_k}$ и $e(k) = \frac{s(k)}{k}$ соответственно, где $t_1$ и $t_k$ — время работы алгоритма на одном и на $k$ мультипроцессорах соответственно. При измерении времени работы не учитывалось время, затрачиваемое алгоритмом на загрузку данных в память графического процессора и на выдачу результата.

В экспериментах использовались синтетический и реальный временные ряды, состоящие из $10^5$ элементов. Генерация синтетического временного ряда осуществлена на основе модели случайных блужданий (Random Walk) [19]. Реальный временной ряд взят из работы [8] и представляет собой сигналы ЭКГ, снятые с дискретизацией 128 Гц. Количество опорных подпоследовательностей в экспериментах взято $r = 10$.




(a) синтетический ряд Random Walk ($n = 10^5$)




(b) реальный ряд ЭКГ ($n = 10^5$)

**Рис. 1.** Ускорение и параллельная эффективность алгоритма

Результаты экспериментов представлены на рис. 1. Можно видеть, что разработанный параллельный алгоритм демонстрирует ускорение, близкое к линейному, и параллельную эффективность от 50 до 100 процентов (в зависимости от длины искомого лейтмотива). При этом лучшие показатели ожидаемо наблюдаются при бóльших значениях длины искомого лейтмотива, обеспечивающих алгоритму бóльшую вычислительную нагрузку.

## 6. Заключение

В настоящей статье рассмотрена задача ускорения поиска лейтмотива временного ряда на современном графическом процессоре. Лейтмотив представляет собой пару подпоследовательностей временного ряда, наиболее похожих друг на друга. Задача поиска лейтмотивов встречается в широком спектре предметных областей: биоинформатика, обработка речи, прогнозирование природных катаклизмов, неврология и др.

Разработан новый параллельный алгоритм поиска лейтмотива временного ряда на графическом процессоре для случая, когда временной ряд может быть размещен в оперативной памяти. Предложенный алгоритм является параллельной версией последовательного алгоритма MK [13]. Распараллеливание выполнено с помощью технологий программирования OpenACC. Разработаны матричные структуры данных, позволяющие эффективно распараллелить и векторизовать вычисления на графическом процессоре. Представлены результаты вычислительных экспериментов, показывающие хорошую масштабируемость алгоритма.

Проведенное исследование может быть продолжено в следующих направлениях: разработка версии с алгоритма использованием технологии программирования CUDA, а также распределенной версии алгоритма для высокопроизводительных кластеров с вычислительными узлами на базе графических процессоров.

*Авторы благодарят Ксению Юрьевну Никольскую за помощь в проведении вычислительных экспериментов.*

## Литература

1. Balasubramanian A., Wang J., Prabhakaran B. Discovering Multidimensional Motifs in Physiological Signals for Personalized Healthcare // J. Sel. Topics Signal Processing. 2016. Vol. 10, No. 5. P. 832–841. DOI: 10.1109/JSTSP.2016.2543679.

2. Brown A., Yemini E., Grundy L., Jucikas T., Schafer W. A Dictionary of Behavioral Motifs Reveals Clusters of Genes Affecting Caenorhabditis Elegans Locomotion // Proceedings of the National Academy of Sciences of the United States of America. 2012. Vol. 110, No. 2. P. 791–796. DOI: 10.1073/pnas.1211447110.

3. Cheng J., Grossman M., McKercher T. Professional CUDA C Programming. 1st Edition. Wrox, 2014. 528 p.

4. Chiu B.Y., Keogh E.J., Lonardi S. Probabilistic Discovery of Time Series Motifs // Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03, August 24–27, 2003, Washington, D.C., USA. ACM, 2003. P. 493–498. DOI: 10.1145/956750.956808.

5. Duran A., Klemm M. The Intel Many Integrated Core Architecture // Proceedings of the 2012 International Conference on High Performance Computing and Simulation, HPCS 2012, July 2–6, 2012, Madrid, Spain. 2012. P. 365–366. DOI: 10.1109/HPCSim.2012.6266938.

6. Fang J., Varbanescu A.L., Sips H.J. Sesame: A User-Transparent Optimizing Framework for Many-Core Processors // Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, May 13–16, 2013, Delft, Netherlands. 2013. P. 70–73. URL: 10.1109/CCGrid.2013.79.

7. Farber R. Parallel Programming with OpenACC. 1st Edition. Morgan Kaufmann, 2016. 326 p.

8. Goldberger A., Amaral L., Glass L., Hausdorff J., Ivanov P., Mark R., Mietus J., Moody G., Peng C., Stanley H. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals // Circulation. 2000. Vol. 101, No. 23. P. 215–220. DOI: 10.1161/01.CIR.101.23.e215.

9. Mattson T. S08 - Introduction to OpenMP // Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11–17, 2006, Tampa, FL, USA. 2006. P. 209. DOI: 10.1145/1188455.1188673.

10. McGovern A., Rosendahl D.H., Brown R.A., Droegemeier K. Identifying Predictive Multi-dimensional Time Series Motifs: an Application to Severe Weather Prediction // Data Min. Knowl. Discov. 2011. Vol. 22, No. 1–2. P. 232–258. DOI: 10.1007/s10618-010-0193-7.

11. Meng J., Yuan J., Hans M., Wu Y. Mining Motifs from Human Motion // Proceedings of the Eurographics 2008 - Short Papers, April 14–18, 2008, Crete, Greece. Eurographics Association, 2008. P. 71–74. DOI: 10.2312/egs.20081024.

12. Minnen D., Isbell C.L., Essa I.A., Starner T. Discovering Multivariate Motifs using Subsequence Density Estimation and Greedy Mixture Learning // Proceedings of the 22nd AAAI Conference on Artificial Intelligence, July 22–26, 2007, Vancouver, British Columbia, Canada. AAAI Press, 2007. P. 615–620.

13. Mueen A., Keogh E.J., Zhu Q., Cash S., Westover M.B. Exact Discovery of Time Series Motifs // Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 – May 2, 2009, Sparks, Nevada, USA. SIAM, 2009. P. 473–484. DOI: 10.1137/1.9781611972795.41.

14. Munshi A., Gaster B.R., Mattson T.G., Fung J., Ginsburg D. OpenCL Programming Guide. 1st Edition. Addison-Wesley, 2011. p. 646.

15. Narang A., Bhattacherjee S. Parallel Exact Time Series Motif Discovery // Proceedings of the 16th International Euro-Par Conference, August 31 – September 3, 2010, Ischia, Italy. Part II. Lecture Notes in Computer Science. Vol. 6272. Springer, 2010. P. 304–315. DOI: 10.1007/978-3-642-15291-7_28.

16. Owens J. GPU Architecture Overview // Proceedings of the International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '07, August 5–9, 2007, San Diego, California, USA. ACM, New York, NY, USA. DOI: 10.1145/1281500.1281643.

17. Padua D.A. POSIX Threads (Pthreads) // Encyclopedia of Parallel Computing. Springer, 2011. P. 1592–1593. DOI: 10.1007/978-0-387-09766-4_447.

18. Patel P., Keogh E.J., Lin J., Lonardi S. Mining Motifs in Massive Time Series Databases // Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM 2002, December 9–12, 2002, Maebashi City, Japan. IEEE Computer Society, 2002. P. 370–377. DOI: 10.1109/ICDM.2002.1183925.

19. Pearson K. The Problem of the Random Walk // Nature. 1905. Vol. 72, No. 294. DOI: 10.1038/072342a0.

20. Shieh J., Keogh E.J. *i*SAX: Indexing and Mining Terabyte Sized Time Series // Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 24–27, 2008, Las Vegas, Nevada, USA. ACM, 2008. P. 623–631. DOI: 10.1145/1401890.1401966.

21. Tanaka Y., Iwamoto K., Uehara K. Discovery of Time-Series Motif from Multi-Dimensional Data Based on MDL Principle // Machine Learning. 2005. Vol. 58, No. 2-3. P. 269–300. DOI: 10.1007/s10994-005-5829-2.

22. Wilson D.R., Martinez T.R. Reduction Techniques for Instance-based Learning Algorithms // Machine Learning. 2000. Vol. 38, No. 3. P. 257–286. DOI: 10.1023/A:1007626913721.

23. Yoon C.E., O'Reilly O., Bergen K.J., Beroza G.C. Earthquake Detection through Computationally Efficient Similarity Search // Science Advances. 2015. Vol. 1, No. 11. P. 1–13. DOI: 10.1126/sciadv.1501057.

24. Zymbler M.L., Kraeva Ya.A. Discovery of Time Series Motifs on Intel Many-Core Systems // Lobachevskii Journal of Mathematics. 2019. Vol. 40, No. 12. P. 2124–2132. DOI: 10.1134/S199508021912014X.