Нахождение похожих подпоследовательностей временного ряда с помощью многоядерного сопроцессора Intel Xeon Phi*

А.В. Мовчан, М.Л. Цымблер

Южно-Уральский государственный университет (НИУ)

Задача поиска похожих подпоследовательностей временного ряда возникает в широком спектре предметных областей, таких как медицина, моделирование климата, финансы и др. В работе предлагается параллельный алгоритм решения указанной задачи, использующий как центральный процессор, так и многоядерный сопроцессор Intel Xeon Phi. Реализация основана на технологии параллельного программирования OpenMP и режиме выполнения приложения, при котором часть кода и данных выгружается на сопроцессор. Алгоритм предполагает использование на стороне процессора очереди подпоследовательностей, которые выгружаются на сопроцессор для вычисления расстояния между подпоследовательностями и запросом, что обеспечивает высокую интенсивность вычислений, выполняемых на сопроцессоре. Результаты экспериментов показывают превосходство разработанного алгоритма над аналогами для GPU и FPGA.

1. Введение

Временной ряд представляет собой совокупность вещественных значений, каждое из которых ассоциировано с последовательными отметками времени. Задача поиска похожих подпоследовательностей предполагает нахождение участков временного ряда, которые являются похожими на заданный ряд меньшей длины. Данная задача возникает в широком спектре предметных областей: мониторинг показателей функциональной диагностики организма человека, моделирование климата, финансовое прогнозирование и др.

В качестве меры схожести временных рядов могут использоваться различные метрики на основе Евклидова расстояния, однако на сегодня динамическая трансформация шкалы времени (Dynamic Time Warping, DTW) [3] является наиболее популярной мерой во многих приложениях [4], однако по сравнению с Евклидовым расстоянием DTW вычислительно более сложна. На сегодня предложено большое количество подходов для решения данной задачи: отбрасывание заведомо непохожих подпоследовательностей на основе оценки нижней границы расстояния [4], повторное использование вычислений [19], индексирование [11] и др. Тем не менее, вычисление DTW по-прежнему занимает существенную часть времени работы алгоритмов поиска похожих подпоследовательностей. В силу этого актуальными являются исследования, посвященные использованию параллельных вычислений для решения данной задачи на кластерных системах [21], многоядерных процессорах [20], FPGA и GPU [19].

В данной статье предлагается параллельный алгоритм поиска похожих подпоследовательностей временного ряда для процессора, оснащенного многоядерным сопроцесором Intel Xeon Phi [5]. Статья организована следующим образом. Раздел 2 содержит формальное определение задачи и краткий обзор архитектуры и модели программирования многоядерного сопроцессора Intel Xeon Phi. В разделе 3 описан предложенный алгоритм. Результаты экспериментов представлены в разделе 4. В заключении суммируются полученные результаты и указываются направления будущих исследований.

*Работа выполнена при финансовой поддержке Минобрнауки России в рамках ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014–2020 годы» (Госконтракт № 14.574.21.0035).

2. Контекст исследования

2.1. Формальная постановка задачи

Временной ряд (time series) T — упорядоченная последовательность $t_1, t_2, ..., t_N$ (где N — длина последовательности) вещественных значений, каждое из которых ассоциировано с отметкой времени.

Подпоследовательность (subsequence) $T_{i,k}$ временного ряда T представляет собой непрерывное подмножество T длины k, начинающееся с позиции i.

Запрос (query) Q — последовательность, длина которой меньше N.

Задача *поиска похожих подпоследовательностей (subsequence matching)* предполагает нахождение всех подпоследовательностей $T_{i,j}$, для которых расстояние $D(T_{i,j}, Q)$ минимально.

Динамическая трансформация шкалы времени (Dynamic Time Warping, DTW) представляет собой меру схожести двух временных рядов. Расстояние на основе DTW между двумя временными рядами X и Y, где $X = x_1, x_2, ..., x_N$ и $Y = y_1, y_2, ..., y_N$, обозначается как $D(X, Y)$ и определяется следующим образом.

$$D(X, Y) = d(N, N),$$

$$d(i, j) = |x_i - y_j| + min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1), \end{cases}$$

$$d(0, 0) = 0; d(i, 0) = d(0, j) = \infty; i = 1, 2, \ldots, N; j = 1, 2, \ldots, N.$$

2.2. Архитектура и модель программирования сопроцессора Intel Xeon Phi

Многоядерный сопроцессор Intel Xeon Phi состоит из 61 ядра на базе архитектуры x86, соединенных высокоскоростной двунаправленной шиной, где каждое ядро поддерживает 4× гипертрединг и содержит 512-битный векторный процессор. Каждое ядро имеет собственный кэш 1 и 2 уровня, при этом обеспечивается когерентность кэшей всех ядер. Сопроцессор соединяется с хост-компьютером посредством интерфейса PCI Express. Поскольку сопроцессор Intel Xeon Phi основан на архитектуре Intel x86, он поддерживает те же программные инструменты и модели программирования, что и ординарный процессор Intel Xeon.

Сопроцессор поддерживает следующие режимы запуска приложений: *native*, *offload* и *symmetric*. В режиме *native* приложение выполняется независимо исключительно на сопроцессоре. В режиме *offload* приложение запускается на процессоре и выгружает вычислительно интенсивную часть работы (код и данные) на сопроцессор. Режим *symmetric* позволяет сопроцессору и процессору взаимодействовать в рамках модели обмена сообщениями (Message Passing Interface).

3. Ускорение поиска с помощью сопроцессора Intel Xeon Phi

3.1. План исследования

Разработка параллельного алгоритма поиска похожих подпоследовательностей на сопроцессоре Intel Xeon Phi включала в себя следующие шаги, детально описанные в следующем разделе.

На первом шаге нами была разработана параллельная версия алгоритма [18]. Используя технологию параллельного программирования OpenMP, мы получили параллельный

алгоритм для процессора, предполагая запустить полученное приложение на сопроцессоре Intel Xeon Phi в режиме *native*. Эксперименты, однако, показали, что несмотря на полученое ускорение по сравнению с последовательным алгоритмом, параллельное приложение на сопроцессоре в режиме *native* работает *медленнее*, чем на процессоре. Данная ситуация является следствием низкой интенсивности вычислений, вынесенных на сопроцессор, т.е. недостаточного количества операций с плавающей точкой на байт данных, выполняемых на сопроцессоре.

На втором шаге мы модифицировали алгоритм, объединив процессор и сопроцессор для обработки временного ряда. В данной версии алгоритма (получившей название «наивной») процессор и сопроцессор Intel Xeon Phi выполняют параллельную версию алгоритма UCR-DTW, разработанную на первом шаге. Для перемещения кода и данных на сопроцессор использовался режим *offload*. Эксперименты, в которых мы варьировали размер части данных, выгружаемых на сопроцессор, показали результаты, схожие с полученными на предыдущем шаге, — по той же причине.

На третьем шаге нами была разработана улучшенная версия алгоритма. Основной идеей данной версии является поддержка очереди подпоследовательностей на стороне процессора для их выгрузки на сопроцессор и вычисления DTW. Это позволило существенно повысить интенсивность вычислений, выносимых на сопроцессор, и, как показали эксперименты, получить приемлемую производительность алгоритма.

3.2. Параллельный алгоритм для процессора

Алгоритм UCR-DTW [18], предложенный учеными Калифорнийского университета в Риверсайде, является на сегодня, по-видимому, наиболее быстрым последовательным алгоритмом поиска похожих подпоследовательностей. Диаграмма деятельности данного алгоритма представлена на рис. 1. Идея алгоритма заключается в применении каскада предварительных оценок, позволяющих отбросить непохожую подпоследовательность до вычисления DTW.

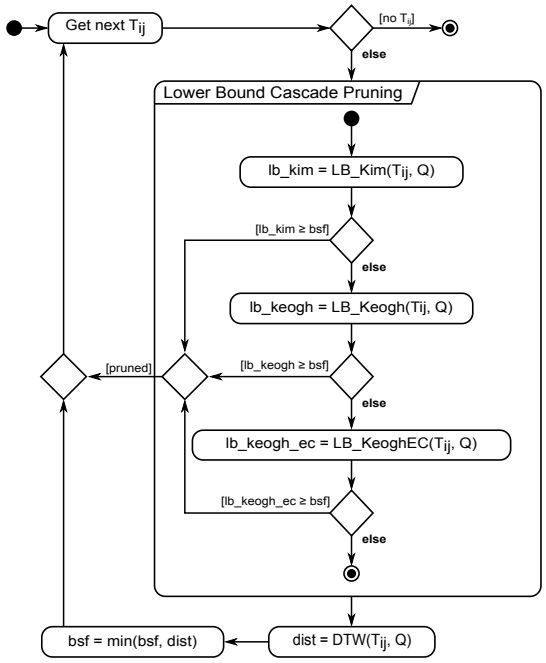

Рис. 1. Последовательный алгоритм

Предложенная нами параллельная версия оригинального алгоритма UCR-DTW[1] пред-

[1] `www.cs.ucr.edu/~eamonn/UCRsuite.html`

ставлена на рис. 2. Используя технологию программирования OpenMP, мы разбиваем временной ряд на равные промежутки, каждый из которых обрабатывается отдельной нитью. Здесь `UCR-DTW` представляет собой подпрограмму, реализующую оригинальный последовательный алгоритм. Этот алгоритм использует разделяемую переменную `bsf` (best-so-far, текущая лучшая оценка), в которой хранится расстояние до ближайшей подпоследовательности. Это позволяет каждой нити отбросить заведомо непохожую подпоследовательность, если расстояние до нее больше значения указанной переменной. Главная нить считывает данные из файла одновременно с обработкой уже прочитанных данных остальными нитями.

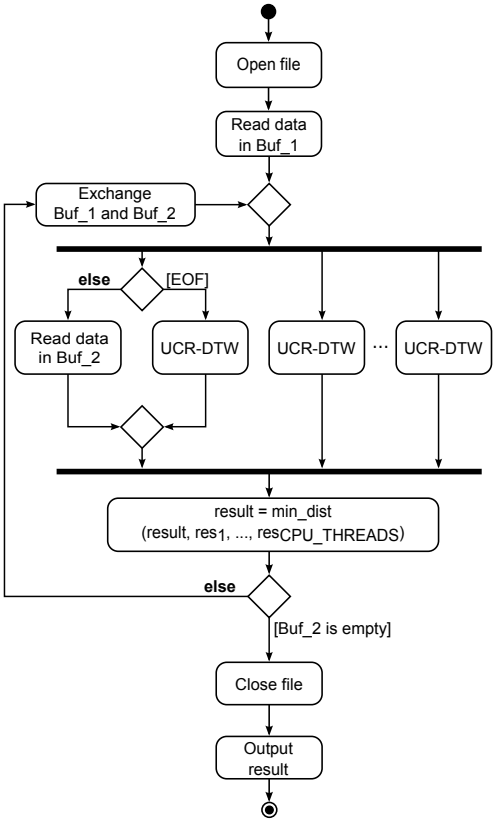

Рис. 2. Параллельный алгоритм для процессора

Мы исследовали производительность полученного алгоритма на синтетических данных (см. рис. 7). Параллельный алгоритм ожидаемо превосходит оригинальный последовательный алгоритм, однако на сопроцессоре в режиме *native* он работает *медленнее*, чем на процессоре. Это является следствием низкой сложности вычислений (малого количества арифметических операций на байт данных), вынесенных на сопроцессор.

3.3. Наивный параллельный алгоритм для сопроцессора

На рис. 3 представлена модифицированнная версия алгоритма, которая была названа «наивной». По сравнению с предыдущим шагом в данной версии работа лишь распределяется между процессором и сопроцессором. Здесь параметр `ALPHA` определяет долю данных, перемещаемых на сопроцессор. Перемещение данных организуется с помощью режима *offload*. Здесь подпрограмма `min_dist` выбирает подпоследовательности с минимальным значением DTW.

Как и на предыдущем шаге, мы провели эксперименты с разработанным алгоритмом на синтетических данных (см. рис. 7). Независимо от значения параметра `ALPHA` наивный алгоритм показывает худшую производительность, чем параллельный алгоритм, не использующий сопроцессор.

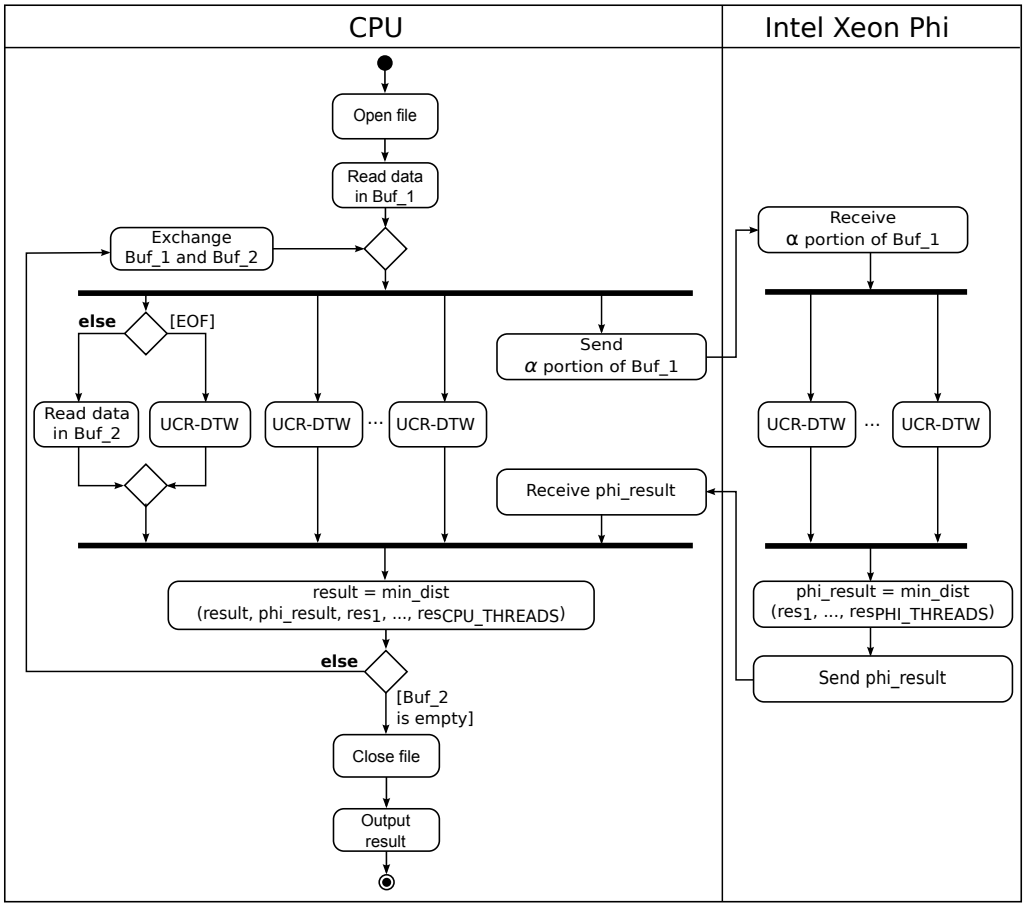

Рис. 3. Наивный параллельный алгоритм для процессора и сопроцессора Intel Xeon Phi

Причина заключается в том, что мы по-прежнему не увеличили интенсивность вычислений, выполняемых на сопроцессоре. Кроме того, разделяемая переменная `bsf` не может быть синхронизирована процессором и сопроцессором при выполнении выгрузки (в действующей модели программирования такая синхронизация выполняется автоматически лишь в начале и по завершении секции *offload*), что обеспечивает большее количество заведомо не похожих, но не отвергнутых подпоследовательностей, для которых необходимо вычислять DTW.

3.4. Улучшенный параллельный алгоритм

Улучшенная версия алгоритма, разработанного на предыдущем шаге, представлена на рис. 4.

Основной идеей улучшенной версии является поддержка очереди подпоследовательностей на стороне процессора. Подпоследовательности, поступающие в очередь, являются «кандидатами» для выгрузки на сопроцессор и последующего вычисления DTW. Одна из нитей процессора объявляется *мастером*, остальные — *рабочими*. Как только очередь заполняется, мастер выгружает ее на сопроцессор.

Поведение рабочего заключается в следующем. Рабочий вычисляет каскадные оценки для подпоследовательности. Если подпоследовательность не похожа на запрос, рабочий отбрасывает ее, в противном случае рабочий добавляет подпоследовательность в очередь. Если очередь заполнена, и данные, загруженные на сопроцессор на предыдущем шаге, еще не обработаны, рабочий осуществляет вычисление DTW самостоятельно.

Здесь параметр `BETA` определяет размер очереди, а подпрограмма `UCR-DTW*` (см. рис. 5)

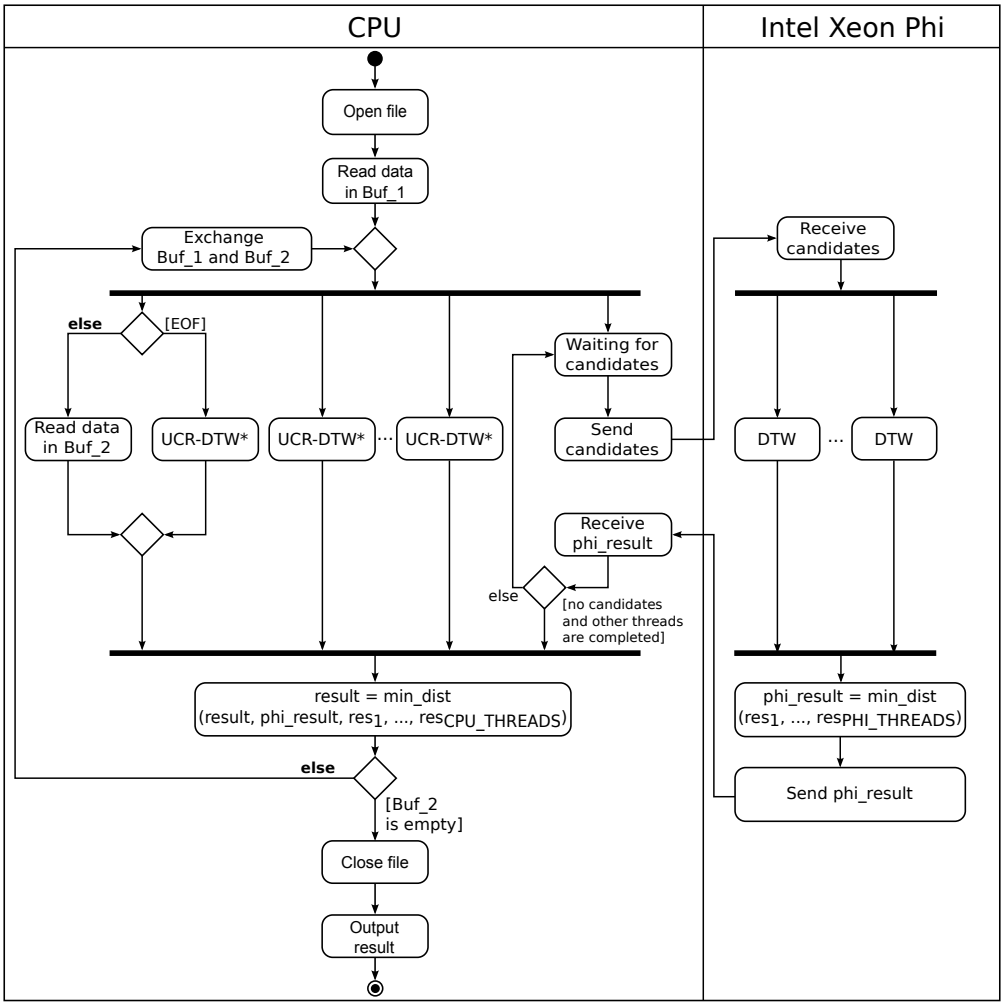

Рис. 4. Улучшенный параллельный алгоритм для процессора и сопроцессора Intel Xeon Phi

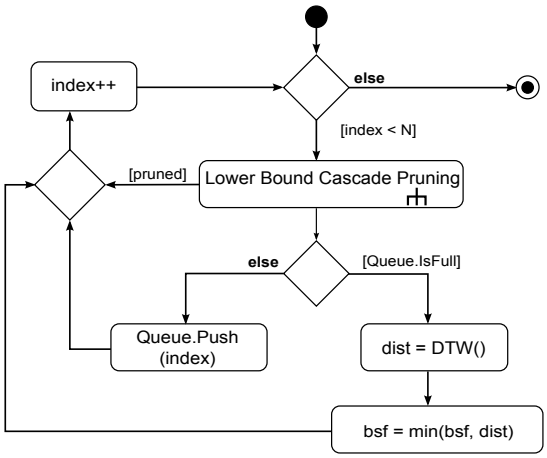

Рис. 5. Вспомогательный алгоритм UCR-DTW*

реализует поведение рабочего, как описано выше.

В конце выгружаемой секции информация о наиболее похожих подпоследовательностях, найденных на сопроцессоре, передается на процессор. Финальный результат вычисляется среди похожих подпоследовательностей, найденных как на процессоре, так и на сопроцессоре.

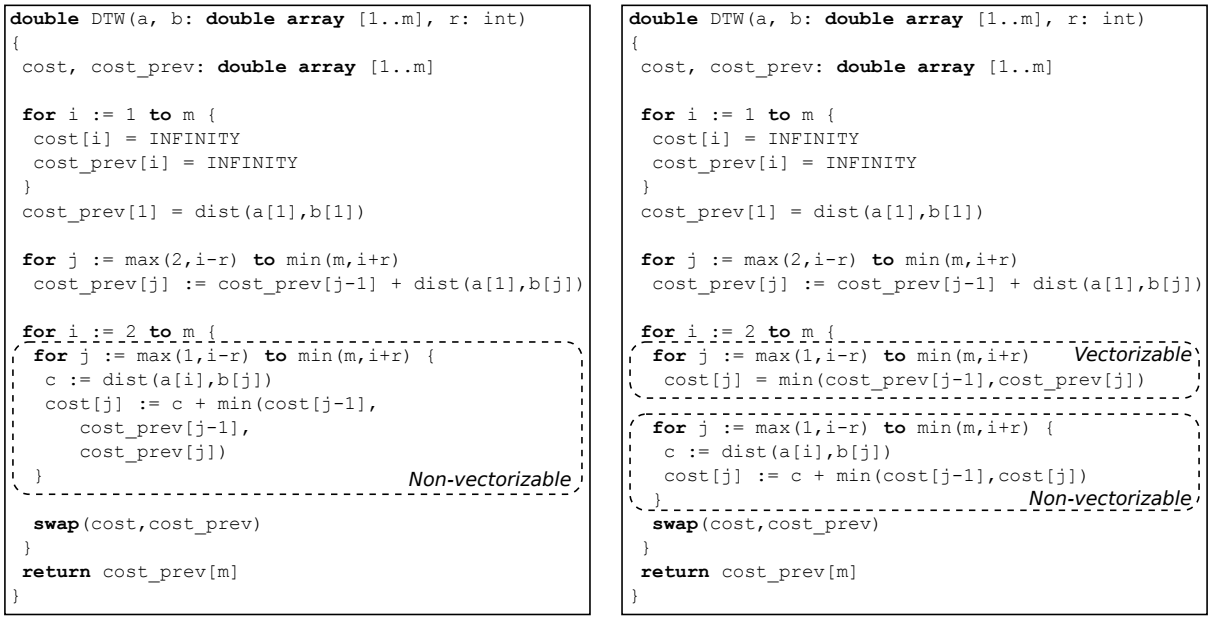

Рис. 6. Изменения в исходном коде, обеспечивающие векторизацию вычисления DTW

В дополнение к этому мы выполнили небольшую модификацию исходного кода, вычисляющего DTW, как показано на рис. 6, чтобы обеспечить векторизацию операций внутри цикла `for`.

4. Эксперименты

Для оценки разработанного алгоритма мы выполнили эксперименты на узле суперкомпьютера «Торнадо ЮУрГУ»[1], спецификации которого представлены в табл. 1.

Таблица 1. Спецификация узла суперкомпьютера «Торнадо ЮУрГУ»

Спецификации	Процессор	Сопроцессор
Модель	Intel Xeon X5680	Intel Xeon Phi SE10X
Количество ядер	6	61
Тактовая частота, ГГц	3.33	1.1
Количество нитей на ядро	2	4
Пиковая производительность, TFLOPS	0.371	1.076

В качестве временного ряда фигурировали как синтетические, так и реальные данные. В экспериментах измерялось время поиска похожих подпоследовательностей для запросов различной длины. Мы также исследовали утилизацию сопроцессора и влияние на производительность векторизации и размера очереди, и сравнили наш алгоритм с аналогами для GPU и FPGA.

[1] `supercomputer.susu.ru/en/computers/tornado/`

4.1. Производительность на синтетических и реальных данных

В первой серии экспериментов мы использовали синтетический временной ряд, состоящий из 100 млн. точек, сгенерированный на основе модели случайных блужданий [17].

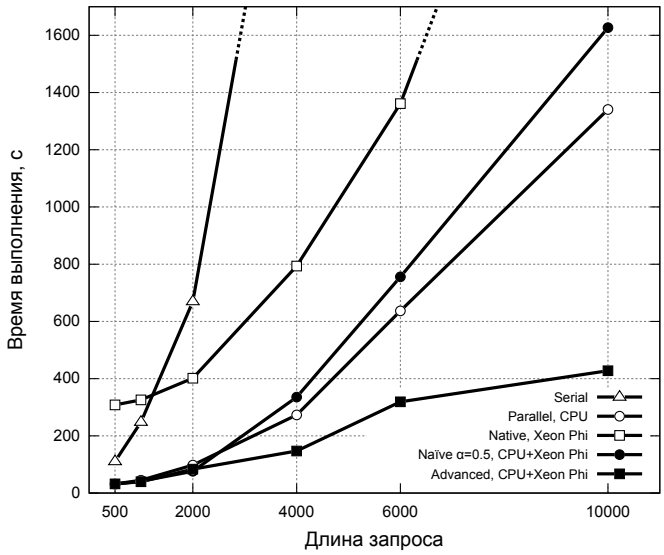

Рис. 7. Производительность на синтетических данных

Результаты экспериментов на синтетических данных, представленные на рис. 7, показывают, что разработанный алгоритм более эффективен для запросов большей длины. В случае, когда запросы имеют меньшую длину, наш алгоритм показывает производительность, сходную с параллельным алгоритмом для процессора (не использующим сопроцессор).

Вторая серия экспериментов исследует производительность разработанного алгоритма на реальных данных ЭКГ, состоящих из 20 млн. точек (около 22 час. ЭКГ, снятой с дискретизацией 250 Гц).

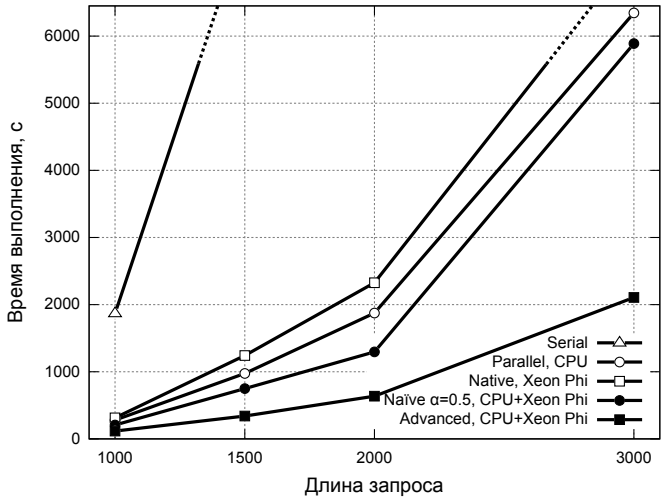

Рис. 8. Производительность на реальных данных

Результаты экспериментов на реальных данных показаны на рис. 8. Разработанный алгоритм показывает в три раза бóльшую производительность, чем параллельный алгоритм, не использующий сопроцессор.

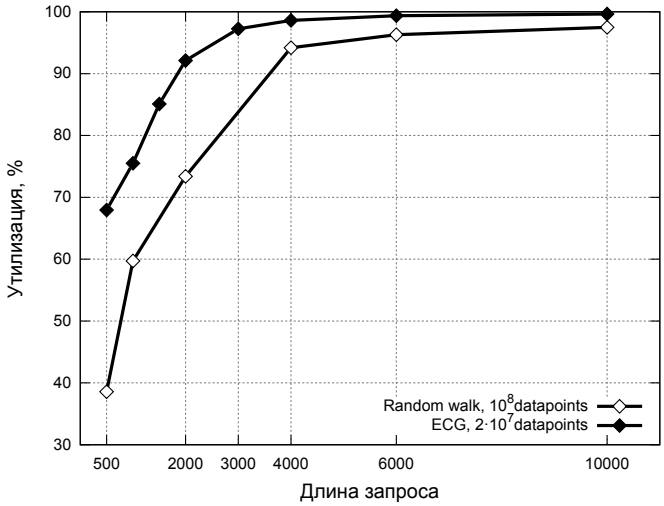

Рис. 9. Утилизация сопроцессора

Мы также исследовали утилизацию сопроцессора во время вычислений. Рис. 9 показывает, что при длине запроса более 4000 мы имеем практически 100% утилизацию сопроцессора как для синтетических, так и для реальных данных.

4.2. Влияние векторизации и размера очереди

Влияние векторизации вычислений DTW на производительность показано на рис. 10; оно тем больше, чем больше длина запроса.

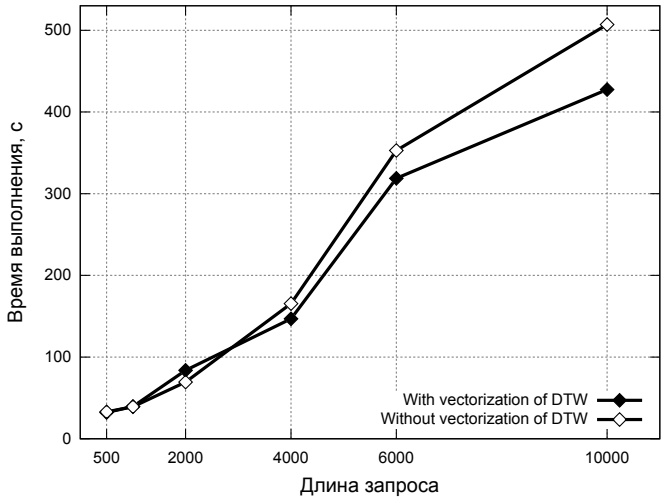

Рис. 10. Влияние векторизации на производительность

Результаты экспериментов, представленные выше, были получены при значении параметра `BETA` (размер очереди) 2400.

Данное значение получено эмпирическим путем в результате предварительных экспериментов, представленных на рис. 11.

4.3. Сравнение с аналогами

Мы сравнили производительность разработанного алгоритма с аналогами для GPU и FPGA, разработанными в работе [19], повторив эксперименты, приведенные в данной ста-

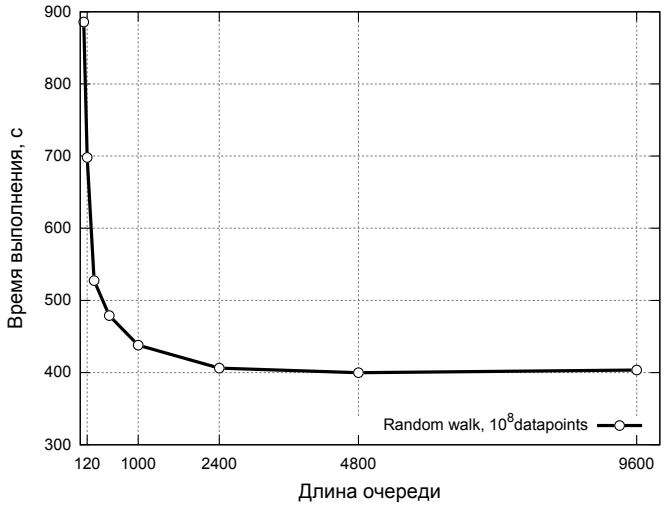

Рис. 11. Влияние размера очереди на производительность

тье. Результаты экспериментов представлены на рис. 12.

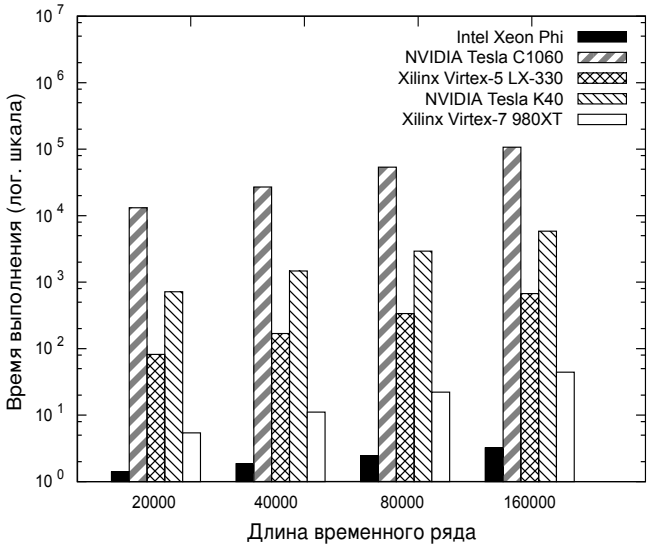

Рис. 12. Сравнение производительности

Заметим здесь, что оборудование, использованнное нами в экспериментах, имеет существенно бóльшую производительность, чем каждый из аналогов из упомянутой работы: 1.44 TFLOPS против 77.8 GFLOPS у NVIDIA Tesla C1060 и 65 GFLOPS у Xilinx Virtex-5 LX-330. В силу этого для создания «честных» условий сравнения мы добавили на график гипотетические результаты для NVIDIA Tesla K40[1] (1.43 TFLOPS) и Xilinx Virtex-7 980XT[2] (0.99 TFLOPS), умножив результаты, показанные на реальных GPU и FPGA, на коэффициенты, соответствующие увеличившейся к текущему моменту пиковой производительности оборудования. Наш алгоритм показывает существенно более высокую производительность.

Наконец, мы сравнили относительную стоимость оборудования, использованного в экспериментах (см. рис. 13).

Решение на основе продукции Intel имеет несколько бóльшую стоимость, чем на основе NVIDIA, однако в то же время дает существенно более высокую производительность.

[1] `www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Data-sheet.pdf`

[2] `www.xilinx.com/publications/prod_mktg/Virtex7-Product-Brief.pdf`

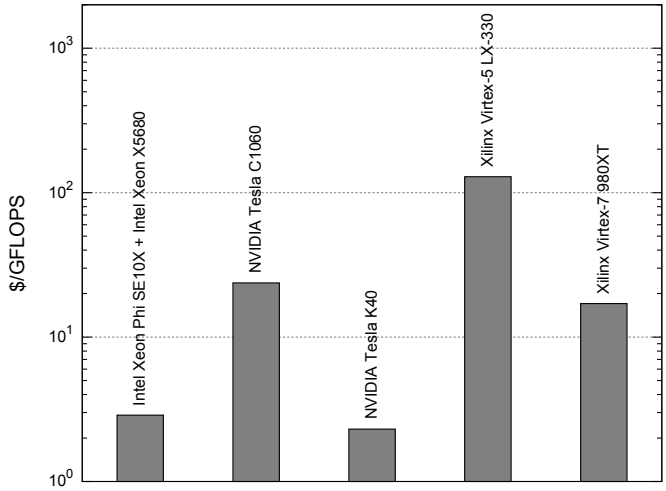

Рис. 13. Сравнение относительной стоимости

5. Заключение

В данной статье описаны проектирование и реализация параллельного алгоритма поиска похожих подпоследовательностей временного ряда, использующего как центральный процессор, так и многоядерный сопроцессор Intel Xeon Phi. Реализация использует технологию параллельного программирования OpenMP. Взаимодействие процессора и сопроцессора реализовано с помощью режима *offload*, когда часть кода и данных выгружается на сопроцессор. Высокая интенсивность вычислений, выполняемых на сопроцессоре, достигается за счет использования на стороне процессора очереди подпоследовательностей, которые выгружаются на сопроцессор для вычисления динамической трансформации шкалы времени. Эксперименты, проведенные на синтетических и реальных данных, показали превосходство разработанного алгоритма как над параллельным алгоритмом, использующим только процессор, так и над аналогичными алгоритмами поиска похожих подпоследовательностей для GPU и FPGA.

В качестве возможного направления дальнейших исследований интересными представляются две задачи: модернизация разработанного алгоритма для случая вычислительного узла с несколькими сопроцессорами Intel Xeon Phi и расширение данного алгоритма для кластерной системы, вычислительные узлы которой оснащены сопроцессорами Intel Xeon Phi.

Литература

1. Мовчан А.В., Цымблер М.Л. Разработка параллельного алгоритма поиска похожих подпоследовательностей временного ряда для сопроцессора Intel Xeon Phi // Параллельные вычислительные технологии (ПаВТ 2014): труды международной научной конференции (1–3 апреля 2014 г., г. Ростов-на-Дону). Челябинск: Издательский центр ЮУрГУ, 2014. С. 372.

2. Мовчан А.В., Цымблер М.Л. Параллельный алгоритм поиска похожих подпоследовательностей временного ряда для сопроцессора Intel Xeon Phi // Научный сервис в сети Интернет: многообразие суперкомпьютерных миров: Труды Международной суперкомпьютерной конференции (22–27 сентября 2014 г., г. Новороссийск). М.: Изд-во МГУ, 2014. С. 245–251.

3. Berndt D.J., Clifford J. Using Dynamic Time Warping to Find Patterns in Time Series //

Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop, Seattle, Washington, July 1994. AAAI Press, 1994. P. 359–370.

4. Ding H., Trajcevski G., Scheuermann P., Wang X., Keogh E. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures // Proceedings of the VLDB Endowment, 2008. Vol. 1, No. 2. P. 1542–1552.

5. Duran A., Klemm M. The Intel Many Integrated Core Architecture // 2012 International Conference on High Performance Computing and Simulation, HPCS 2012, Madrid, Spain, July 2–6, 2012. IEEE, 2012. P. 365–366.

6. Faloutsos C., Ranganathan M., Manolopoulos Y. Fast Subsequence matching in Time-series Databases // The 1994 ACM SIGMOD International Conference on Management of Data, New York, NY, 24–27 May, 1994. ACM, 1994. P. 419–429.

7. Fu T.-C. A Review on Time Series Data Mining // Engineering Applications of Artificial Intelligence, 2011. Vol. 24, No. 1. P. 164–181.

8. Heinecke A., Klemm M., Pfluger D., Bode A., Bungartz H.-J. Extending a Highly Parallel Data Mining Algorithm to the Intel Many Integrated Core Architecture // Proceedings of the Euro-Par 2011 Workshops, Bordeaux, France, August 29 – September 2, 2011. Lecture Notes in Computer Science. 2012. Vol. 7156. Springer, 2011. P. 375–384.

9. Kim M.-S., Kim S.-W., Shin M. Optimization of Subsequence Matching under Time Warping in Time-series Databases // SAC. ACM, 2005. P. 581–586.

10. Kim S.W., Yoon J., Park S., Kim T.H. Shape-based Retrieval of Similar Subsequences in Time-series Databases // The 2002 ACM symposium on Applied computing, Madrid, Spain, 10–14 March, 2002. ACM, 2002. P. 438–445.

11. Lim S.-H., Park H., Kim S.-W. Using Multiple Indexes for Efficient Subsequence Matching in Time-series Databases // Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12–15, 2006, Proceedings. Lecture Notes in Computer Science. Vol. 3882. Springer, 2006. P. 65–79.

12. Loh W.-K., Moon Y.-S., Srivastava J. Distortion-free Predictive Streaming Time-series Matching // Information Sciences. 2010. Vol. 180, No. 1. P. 1458–1476.

13. Moon, Y.-S., Whang, K.-Y., Han, W.-S., 2002. General match: a subsequence matching method in time-series databases based on generalized windows. In: Franklin, M. J., Moon, B., Ailamaki, A. (Eds.), SIGMOD Conference. ACM, pp. 382–393.

14. Moon Y.-S., Whang K.-Y., Loh W.-K. Duality-Based Subsequence Matching in Time-Series Databases // The 17th International Conference on Data Engineering, Washington, DC, USA, 2–6 April, 2001. IEEE Computer Society, 2001. P. 263–272.

15. Park S., Kim S.W., Cho J.S., Padmanabhan S. Prefix-querying: an Approach for Effective Subsequence Matching under Time Warping in Sequence Databases // The 10th International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, 5–10 November, 2001. ACM, 2001. P. 255–262.

16. Park S., Kim S.-W., Chu W.W. Segment-based Approach for Subsequence Searches in Sequence Databases // Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), March 11–14, 2001, Las Vegas, NV, USA. ACM, 2001. P. 248–252.

17. Pearson K. The Problem of the Random Walk // Nature. 1905. Vol. 72, No. 1865. P. 294.

18. Rakthanmanon T., Campana B., Mueen A., Batista G., Westover B., Zhu Q., Zakaria J., Keogh E. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping // The 18th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Beijing, China, 12–16 August, 2012. ACM, 2012. P. 262–270.

19. Sart D., Mueen A., Najjar W., Keogh E., Niennattrakul V. Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs // The 10th IEEE International Conference on Data Mining, Sydney, NSW, Australia, 13–17 December, 2010. IEEE, 2010. P. 1001–1006.

20. Srikanthan S., Kumar A., Gupta R. Implementing the Dynamic Time Warping Algorithm in Multithreaded Environments for Real Time and Unsupervised Pattern Discovery // Computer and Communication Technology (ICCCT), Allahabad, India, 15–17 September, 2011. IEEE Computer Society, 2011. P. 394–398.

21. Takahashi N., Yoshihisa T., Sakurai Y., Kanazawa M. A Parallelized Data Stream Processing System Using Dynamic Time Warping Distance // 2009 International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2009, Fukuoka, Japan, March 16–19, 2009. IEEE Computer Society, 2009. P. 1100–1105.