ПРОГРАММИРОВАНИЕ, 2024, № 6, с. 48–58

АНАЛИЗ ДАННЫХ

УДК: 004.93

ОПТИМИЗАЦИЯ БЫСТРОДЕЙСТВИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ РЕАЛИЗАЦИИ АЛГОРИТМОВ КЛАССИФИКАЦИИ И ПРИВЯЗКИ ДЕЛОВЫХ ДОКУМЕНТОВ

© 2024 г. О. А. Славин[a, b, *]

[a] *Федеральный исследовательский центр "Информатика и управление" РАН*
119333 Москва, ул. Вавилова, 44/2, Россия
[b] *ООО "Смарт Энджинс Сервис"*
117312 Москва, проспект 60-летия Октября, 9, Россия
** E-mail: oslavin@isa.ru*

Поступила в редакцию 13.07.2024 г.
После доработки 15.07.2024 г.
Принята к публикации 15.07.2024 г.

В работе рассматриваются технологии оптимизации быстродействия программного обеспечения. Методы оптимизации подразделяются на высокоуровневые и низкоуровневые, а также на распараллеливание. Описываемые методы оптимизации применяются к программам и программным системам, реализующим разнообразную обработку информации, в которых неэффективность использования аппаратных ресурсов может присутствовать в большом числе горячих точек. Как пример приведен алгоритм классификации и привязки полей в распознанном образе делового документа. Перечисляются особенности реализации задач классификации и привязки, состоящие в применении созвездий особых текстовых точек и модифицированного расстояния Левенштейна, В качестве OCR была использована система SDK Smart Document Engine и Tesseract. Описано несколько способов оптимизации быстродействия функций классификации и привязки содержимого документа. Также описана оптимизация быстродействия системы сортировки потока изображений деловых документов. Предлагаемые методы оптимизации быстродействия программного обеспечения пригодны не только для реализации алгоритмов обработки изображений, но и для вычислительных алгоритмов, в которых проводится циклическая обработка информации большого объема.

Ключевые слова: анализ текста, распознавание документа, классификация документа, текстовая особая точка, ускорение

DOI: 10.31857/S0132347424060057, **EDN:** DYKMMM

1. ВВЕДЕНИЕ

Необходимость оптимизации быстродействия программ объясняется различными потребностями. Требования к быстродействию может сформулировать Заказчик информационной системы в техническом задании. Требования к быстродействию указываются для конкретных видов компьютеров и конкретных операционных систем. Аналогично возникают требования к быстродействию для приложений, предназначенных для мобильных устройств. Ограничения по времени выполнения мобильных приложений связаны не только с ограничением времени реакции, но и с ограничениями энергопотребления и нагрева мобильного устройства. Существует корреляция между быстродействием приложения и энергопотреблением.

Оптимизация ПО преследует одну или несколько целей:

- уменьшение среднего времени исполнения программного приложения на некотором тестовом наборе;
- уменьшение среднего времени исполнения реализованной функции.

2. ПРИНЦИПЫ ОПТИМИЗАЦИИ БЫСТРОДЕЙСТВИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

При разработке программного обеспечения (далее – ПО) возможна оптимизация нескольких типов:

- высокоуровневая оптимизация (оптимизация алгоритмов);
- низкоуровневая оптимизация с использованием особенностей вычислительной техники;
- параллельное программирование.

Высокоуровневая оптимизация базируется на выборе метода решения задачи, которая должна войти в состав разрабатываемой программы. Алгоритм может существенно зависеть от области определения, например, уменьшение объема данных очевидным образом уменьшает время перебора. Выбор параметров алгоритма также может существенно влиять на сложность. Высокоуровневая оптимизация проводится для упрощенной архитектуры компьютера, такой как архитектура фон Неймана или Гарвардская архитектура. Под архитектурой понимается совокупность пользовательских характеристик, к которым относят основные устройства и блоки упрощенного компьютера, а также структуру связей между ними. С точки зрения разработчика программного обеспечения архитектура компьютера – набор описаний используемых данных, операций (инструкций) и их характеристик.

В высокоуровневой оптимизации возможно применение следующих способов:

• анализ алгоритма (реализация, выбор, область определения, параметризация, правило остановки);

• использование промежуточных данных (мемоизация [1]);

• представление исходных данных;

• уменьшение сложности алгоритма (lookup table [2], интерполяция);

• оптимизация циклов (вынос вычислений из тела цикла, слияние циклов, loop unrolling).

Низкоуровневая оптимизация, ориентированная на аппаратную платформу, – это совокупность технических средств, определяющих среду функционирования конкретных программ. Основой аппаратной платформы является совокупность системной (материнской) платы, центрального процессора (далее – ЦП) и запоминающих устройств. Выполняемая на компьютере программа состоит из команд конкретного процессора.

Низкоуровневая оптимизация проводится для конкретных процессорных микроархитектур [3], таких как:

• CISC (различная длина машинной инструкции);

• RISC (одинаковая длина машинной инструкции);

• VLIW (параллельное выполнение нескольких операций в одной инструкции);

• суперскалярная архитектура, в которой решение о параллельном исполнении двух или более команд между несколькими устройствами исполнения принимается аппаратурой процессора на этапе исполнения.

Вообще говоря, низкоуровневая оптимизация для различных процессорных микроархитектур будет различной. Например, различаются множества команд SIMD для ЦП ARM и ЦП Intel. Директивы программиста для улучшения параллельного исполнения инструкций для ЦП архитектуры VLIW невозможны для ЦП Intel.

Использование параллельного исполнения (параллелизации) является эффективным способом повышения быстродействия ПО. Параллельному исполнению способствует такая организация вычислительного процесса, при которой на одном процессоре попеременно выполняются сразу несколько программ, совместно использующих один или несколько процессоров и другие ресурсы компьютера. Такой способ называется многозадачностью. Многозадачность призвана повысить эффективность использования вычислительной системы, при этом могут использоваться различные критерии эффективности вычислительных систем, например:

• пропускная способность – количество задач, выполняемых вычислительной системой в единицу времени в операционных системах (далее – ОС) пакетной обработки;

• удобство работы пользователей, заключающееся, в частности, в том, что они имеют возможность интерактивно работать одновременно с несколькими приложениями на одной машине в ОС разделения времени;

• реактивность системы – способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата в ОС реального времени.

Параллелизацию имеет смысл проводить после завершения высокоуровневой и низкоуровневой оптимизации последовательной программы или алгоритма. Основой параллелизации является выполнение частей программы на нескольких исполнительных устройствах, в качестве которых мы будем рассматривать несколько центральных процессоров. Формально различают параллельное выполнение задач приложения на одном компьютере и распределенное выполнение нескольких приложений на нескольких компьютерах в локальной сети. Основными подходами к разработке параллельных программ являются:

• последовательное программирование с дальнейшим автоматическим распараллеливанием;

• непосредственное формирование потоков параллельного управления с учетом особенностей архитектур параллельных вычислительных систем или операционных систем;

• описание параллелизма без использования явного управления.

3. МЕТОДЫ ОПТИМИЗАЦИИ БЫСТРОДЕЙСТВИЯ ПО

Описываемые далее методы оптимизации применяются, прежде всего, к программам и программным системам, реализующим разнообразную обработку информации. То есть мы рассматриваем не программы, в которых реализуется один или несколько математических алгоритмов, таких как, например, функции из библиотек Eigen или MKL. Иначе класс рассматриваемых программ можно охарактеризовать как программы, в которых неэффективность использования аппаратных ресурсов может присутствовать в большом числе горячих точек.

3.1. Особенности высокоуровневой оптимизации быстродействия для реализации проектов и разработки продуктов

Описанные в разделе 2 принципы оптимизации ПО применяются в различных условиях по-разному.

Так, формирование требований к быстродействию существенно различается при разработке продуктов (API, библиотек, приложений) и реализации проектов (в том числе систем, подсистем, функциональных модулей). Оба вида разработки имеются сходство, для них возможны:

• использование готовых программных модулей;

• высокоуровневая оптимизация;

• применение экспертных оценок ускорения.

Однако имеются и существенные различия. Например, план выпуска продукта допускает увеличение времени на разработку, связанную с созданием новых алгоритмов, которые априори должны обладать меньшей сложностью, нежели уже реализованные алгоритмы. При разработке продуктов необходимо предусмотреть возможность постоянного профилирования и другого анализа быстродействия. В разработке продуктов необходим анализ быстродействия конкурирующих продуктов, на которые следует ориентироваться. В реализации проектов требования к быстродействию могут быть сформированы заказчиком проекта, но срок реализации и ресурсы проекта могут ограничить время, затрачиваемое на оптимизацию. В обоих случаях предусматривается оптимизация покупных и собственных модулей. Оптимизация покупных модулей в части замены или модификации имплементированных алгоритмов чаще всего невозможна, но возможно управление быстродействием посредством представления данных и выбора параметров вызываемых методов.

3.2. Профилирование ПО

Во всех случаях необходимо *профилирование*, т. е. измерение быстродействия пользовательской программы или частей программы. Целью профилирования является исследование быстродействия приложения как в целом, так и составляющих его объектах (точках): функциях, циклах, строках, инструкциях.

Замеры позволяют анализировать:

• общее время исполнения приложения;

• набор горячих точек (hot spots);

• удельное время каждой точки;

• количество вызовов точки;

• степень покрытия программы (доля кода, которая была использована при исполнении);

• загрузку CPU и захват памяти;

• низкоуровневую статистику о загрузке CPU и шины доступа к памяти, кэш-промахах и кэш-попадания;

• степень параллелизма приложения.

Профилированию может подвергаться как отладочная версия приложения, так и оптимизированная отладочная версия или релизная версия с отладочной информацией. Существенным условием профилирования является проведение замеров на одном и том же тестовом наборе (датасете), который может быть как опубликованным в открытом доступе, так и собственным. Профилирование необходимо проводить в одних и тех же условиях на выбранных платформах, включающих аппаратную часть и системное ПО.

Известны несколько методов инструментального профилирования:

• ручной (проведение замеров времени с помощью вручную добавленных в код приложения вызовов системных функций, таких как std:: clock, или использования методов библиотеки std:: chrono);

• семплирование (time-based profiling) – сбор статистики о работе приложения во время профилирования;

• инструментирование (instrumentation) – сбор детализированной информации о времени работы каждой вызванной функции во время профилирования.

Достоинством ручного подхода является простота замеров небольшого числа заранее известных точек, недостатки состоят в следующем:

• необходимость вставки дополнительного кода в программу, что может приводить

к появлению не только ошибок, но и наведенных эффектов;

• возможность анализа результата в текстовом виде;

• отсутствие дерева вызовов функций, средств автоматического анализа.

В профайлере, основанном на семплировании, периодически собирается информация о состоянии программы, например, значения счетчиков производительности процессора, значение счетчика команд. На основании этих замеров проводится подсчет производительности. Метод семплирования менее всего влияет на работу анализируемого ПО и полученная информация обладает малой погрешностью. Основными недостатками таких профайлеров являются:

• получение неполной информации о коде;

• длительное время для сбора реальной статистики.

Метод инструментирования модифицирует исполняемый файл ПО для сбора информации о выполнении каждой функции, исключая время, потраченное на вызываемые функции и обращения к операционной системе. Метод инструментирования предоставляет больше информации, чем метод семплирования, но может существенно замедлить работу профилируемого ПО.

Известно несколько универсальных профайлеров, к ним относятся следующие программы:

• CodeAnalyst;

• Valgrind;

• Performance Profiler из среды Visual Studio;

• Intel VTune.

Все перечисленные профайлеры позволяют определять горячие точки программы, локализовать участки кода, в которых неэффективно используются аппаратные ресурсы, неэффективность использования процессора, выявлять объекты синхронизации, которые негативно влияют на производительность программы. Продукт Intel VTune предоставляет возможность оценки большого числа аппаратных счетчиков и метрик для определения критических объектов. Intel VTune позволяет в режиме эмуляции моделировать работу определенного процессора, включая кэш-память различного уровня с механизмами замещения, декодеры и буферы инструкций, конвейеры инструкций и другие компоненты ЦП и его взаимодействия с памятью.

3.3. Методы низкоуровневой оптимизации быстродействия ПО

Основным способом низкоуровневой оптимизации при разработке ПО является выбор параметров компилятора. Например, для среды Microsoft Visual Studio и других компиляторов важнейшим параметром является вид оптимизации: без оптимизации или с оптимизацией быстродействия, или с оптимизацией объема программы. Другим способом оптимизации быстродействия является явное указание архитектуры, позволяющее компилятору применять при трансляции инструкции выбранного набора команд.

Использование математических библиотек также является эффективным средством ускорения вычислений. Компилятор Intel Compiler Enable Matrix содержит параметр Multiply Library Call, который включает или отключает вызов библиотеки Matrix Multiply. Широко известна библиотека Intel MKL, содержащая многочисленные реализованные и оптимизированные для современных ЦП вычислительные методы.

Разумеется, остается востребованным и программирование на ассемблере или применение интринсиков, что, собственно, и позволяет создавать библиотеки для быстрых вычислений.

3.4. Подходы к распараллеливанию программ

Основными подходами к разработке параллельных программ являются:

• последовательное программирование с дальнейшим автоматическим распараллеливанием;

• непосредственное формирование потоков параллельного управления с учетом особенностей архитектур параллельных вычислительных систем или операционных систем;

• описание параллелизма без использования явного управления.

Последовательное программирование с дальнейшим автоматическим распараллеливанием упрощает разработку, позволяя полностью абстрагироваться от возможностей параллелизации и возложить распараллеливание на инструментальные средства. Очевидный недостаток этого подхода – невозможность достичь максимального ускорения.

Перечислим некоторые средства автоматического распараллеливания с помощью кроссплатформенных многопоточных библиотек для языка C++:

• Qt4 Threads;

• Intel Threading Building Blocks (TBB).

Достоинством подхода, состоящего в непосредственном формировании потоков параллельного управления, является возможность получения значительно большего ускорения для конкретной вычислительной системы по

сравнению с предыдущим подходом. Недостатки подхода состоят в следующем:

• усложнение разработки из-за ручного управления собственными процессами и потоками, в том числе, анализ возможных конфликтов;

• зависимость от конкретной архитектуры многопроцессорного комплекса, что затрудняет переносимость на другие платформы.

Упрощение разработки возможно при распараллеливании с указанием директив, которые указывают компилятору на необходимость распараллеливании фрагмента исходного кода с предполагаемой возможной параллельной реализацией.

При распараллеливании также необходимо проводить профилирование для оценки потерь времени из-за конфликтов потоков и неоптимальности использования данных несколькими потоками.

4. ЗАДАЧИ РАСПОЗНАВАНИЯ И КЛАССИФИКАЦИИ ДЕЛОВЫХ ДОКУМЕНТОВ

Распознавание изображений документов с известным описанием включает в себя несколько задач, таких как:

• поиск границ документа;

• нормализация размера и границ документа;

• извлечение графических примитивов;

• распознавание символов и слов, анализ структуры документа;

• поиск границ и распознавание полей документа;

• постобработка результатов распознавания.

Важнейшими задачами являются классификация типа документа (фрагмента документа) и привязки полей (поиск областей документа для извлечения заполнения). При анализе распознанных изображений необходимо учитывать ошибки OCR, появляющиеся в зашумленных, осветленных или искаженных образах документов. В данной работе рассматриваются деловые документы, предназначенные для обмена данными с организациями и физическими личностями [4]. Деловые документы характеризуются относительно простой структурой и ограниченным словарем статических текстов.

Мы будем определять документ как совокупность полей и статической информации. Структура текста делового документа может быть описана с помощью трех объектов: слово, строка текста и фрагмент текста. Для классификации распознанного документа и привязки полей могут быть применены текстовые особые точки и созвездия текстовых точек, определенные в [5, 6]. Текстовые особые точки являются аналогами геометрических особых точек [7, 8]. Слово модели представляется последовательностью символов. Распознанное слово представляется матрицей альтернатив соответствия знакомест символов символам алфавита распознавания и рамкой слова.

Для пары текстовых особых точек (ω_1, ω_2) могут быть заданы следующие отношения:

• $\omega_1 \in S$, $\omega_2 \in S$ ($\omega_1 \in F$, $\omega_2 \in F$) – обе текстовые точки принадлежат одной строке S или одному фрагменту текста F;

• $\omega_1 \in S_1$, $\omega_2 \in S_1$ ($\omega_1 \in F_1$, $\omega_2 \in F_2$) – обе точки ω_1 и ω_2 принадлежат различным строкам S_1 и S_2 или различным фрагментам текста F_1 и F_2;

• $\omega_1 < \omega_2$ – точка ω_1 размещена “перед” точкой ω_2;

• $\omega_1 \vee \omega_2$ – точка ω_1 размещена “выше” точки ω_2.

Строка текста является созвездием нескольких близких друг к другу текстовых особых точек. Строки текста могут быть найдены с помощью алгоритмов кластеризации рамок распознанных слов. Строка описывается множеством упорядоченных текстовых особых точек. Для двух строк (S_1, S_2) могут быть заданы следующие отношения:

• $S_1 \in F$, $S_2 \in F$ – обе строки текста принадлежат одному фрагменту текста F;

• $S_1 \in F_1$, $S_2 \in F_2$ – обе строки текста принадлежат различным фрагментам текста F_1 и F_2;

• $S_1 \vee S_2$ – строка S_1 размещена “выше” строки S_2.

Под привязкой строки мы понимаем установление соответствия слов распознанной строки с одной из описанных возможных строк. Некоторые точки являются обязательными для привязки. При привязке с каждой обязательной точкой должно быть отождествлено некоторое распознанное слово. Также в описании строки могут присутствовать запрещенные текстовые точки. При привязке строки ни одна из запрещенных точек не может быть отождествлена с некоторым распознанным словом. Остальные текстовые точки могут быть отождествлены с распознанными словами при привязке строки, или не быть отождествленными.

В описании строки для пары текстовых точек могут быть заданы ограничения с помощью следующих метрик:

• количество точек в промежутке между двумя точками ω_1 и ω_2;

• сумма ширин текстовых точек, размещенных между точками ω_1 и ω_2;

• количество строк в промежутке между строками, содержащими точки ω_1 и ω_2;

• евклидово расстояние между двумя проекциями рамок точек ω_1 и ω_2.

Фрагмент текста является совокупностью нескольких текстовых строк. В нашей модели предполагается, что в одном фрагменте строки группируются только в одну колонку. Для двух фрагментов (F_1, F_2) могут быть заданы следующие отношения:

• $F_1 \in F$, $F_2 \in F$ – два фрагмента принадлежат текстовому фрагменту F;

• $F_1 < F_2$ – фрагмент F_1 размещен "перед" фрагментом F_2;

• $F_1 \vee F_2$ – фрагмент F_1 размещен "выше" фрагмента F_2.

Фрагменты текста могут быть созданы с помощью алгоритмов анализа структуры текста. Разбиение документа на части осуществляется на основе его графического строения (разделяющих прямых, колонок, абзацев и т. п.) под управлением некоторого описания (шаблона) документа [9, 10]. Для разбиения на фрагменты могут быть использованы как отрезки, разделяющие фрагменты, так и промежутки между фрагментами. Под привязкой фрагмента мы понимаем установление соответствия слов и строк фрагмента с одним из описанных возможных фрагментов. Аналогично описаниям строки в описании фрагмента содержатся обязательные, запрещенные и обычные строки и слова.

Созвездие задается в виде последовательности упорядоченных точек $\omega_1, \ldots, \omega_n$. Простым случаем созвездия является последовательность точек, принадлежащих одной текстовой строке, в самом простом случае это – шингл (последовательность слов заголовка документа). Другим случаем созвездия являются цепи – последовательность точек, пары которых упорядочены отношениями $\omega_1 < \omega_2$ (простая цепь) или $\omega_1 \vee \omega_2$ (вертикальная цепь). Использование цепей и созвездий позволяет не только находить тип документа, но и детектировать фрагменты и строки текста. Последнее позволяет сократить объем текста, используемого в анализе текстового объекта, например, в привязке поля.

В работах [5, 6] описан способ привязки полей гибкого документа. Привязка строк, параграфов и фрагментов документа проводится с помощью следующего алгоритма классификации. Задаются модели допустимых строк $M_1, M_2, \ldots, M_q$, каждая модель M определена набором текстовых особых точек:

$$M = \begin{Bmatrix} W^+_1, W^+_2, \ldots, W^+_{k+}, W_1, W_2, \ldots, \\ W_k, W^-_1, W^-_2, \ldots, W^-_{k-} \end{Bmatrix} \quad (1)$$

и параметр $d_{\mathrm{LINK}}(M)$ – пороговое значение числа привязанных точек для надежной привязки. В наборе (1) используются три мешка слов:

• запрещенные слова $W^- = \{W^-_1, W^-_2, \ldots, W_{k-}\}$;

• обязательные слова $W^+ = \{W^+_1, W^+_2, \ldots, W^+_{k+}\}$;

• необязательные слова $W = \{W_1, W_2, \ldots, W_k\}$.

Вычисляются оценки $\Delta(S, M_i)$ соответствия моделям каждой из строк S. Оценка $\Delta(S, M_i)$ равняется 0, если:

• была привязана хотя бы одна точка из множества $W^-(S)$;

• не было привязано ни одной точки из множества $W^+(S)$.

Оценка $\Delta(S, M_i)$ равняется 1, если не было привязано ни одной точки $W^-(S)$ и число привязанных точек W(S) и $W^+(S)$ превосходит $d_{\mathrm{LINK}}(M_i)$. Если $\Delta(S, M_i)$ равняется 1, то строка S считается привязанной к модели M_i. Точность привязки строк зависит от предварительной привязки окрестности допустимого размещения строк. После привязки строк проводится поиск (прогноз) границ полей для последующего извлечения информации. Для области каждого поля задаются опорные элементы, определяющие прямоугольник или многоугольник. Привязка поля проводится с помощью привязанных опорных элементов.

Описанный алгоритм классификации строк применяется для классификации документа. Классификация образа страницы документа проводится с помощью привязки точек созвездия с учетом заданных отношений между некоторыми точками.

Отождествление текстовой точки и распознанного слова проводится с помощью предложенного в [6] модифицированного расстояния Левенштейна (далее – МРЛ). Механизм отождествления слов применяется во многих задачах, основанных на сравнении слов с алфавитом в базе данных [11, 12]. Оригинальное расстояние Левенштейна [13] между двумя текстовыми строками V и W определяется как минимальное число редакционных операций для трансформации V в W и вычисляется следующим образом:

$$d_{\mathrm{LEV}}(V, W) = D_{\mathrm{LEV}}(|V|, |W|),$$

$$\forall_j D_{\mathrm{LEV}}(0, j) = 0, \forall_i D_{\mathrm{LEV}}(i, 0) = 0, \quad (2)$$

$$D_{\mathrm{LEV}}(i, j) = \min\big(D_{\mathrm{LEV}}(i, j-1) + 1, D_{\mathrm{LEV}}(i-1, j) + 1, D_{\mathrm{LEV}}(i-1, j-1) + \mathrm{substCost}(v_i, w_i)\big),$$

где $\mathrm{substCost}(v_i, w_j)$ – цена операции замены символа v_i на символ w_j, |V| и |W| – длины слов V

и W. По умолчанию цена любой из редакционных операций равняется 1. В работе алгоритм вычисления расстояния Левенштейна между двумя текстовыми строками реализован в полном соответствии с рекуррентной формулой (2). В реализации не применялись методы экономии памяти, уменьшающие производительность.

Мы будем считать тождественными слова V и W, если $d_{\mathrm{LEV}}(\mathrm{V}, \mathrm{W}) < d(\mathrm{V})$, где $d(\mathrm{V})$ – известный порог для слова модели. При распознавании программами OCR появляются неединичные ошибки распознавания. Поэтому порог $d(\mathrm{V})$ не может быть нулевым. Очевидно, что порог $d(\mathrm{V})$ должен быть различным для слов различной длины. Для учета этого обстоятельства можно использовать нормализованное расстояние Левенштейна [14]:

$$\rho_{\mathrm{LEV}}\left(\mathrm{V}, \mathrm{W}\right) = \frac{2d_{\mathrm{LEV}}\left(\mathrm{V}, \mathrm{W}\right)}{\left|\mathrm{V}\right| + \left|\mathrm{W}\right| + d_{\mathrm{LEV}}\left(\mathrm{V}, \mathrm{W}\right)}.$$

При распознавании зашумленных и искаженных изображений документов возможно появление многочисленных ошибок распознавания. Некоторые ошибки OCR не являются случайными. Ошибочное распознавание образа символа “X” как символа “O” маловероятно. В то же время образ символа “Ъ” может быть ошибочно распознан как символ “Ь” из-за сходства образов “Ъ” и “Ь”. Примерами сходных образов для латинского алфавита являются пары символов “B8”, “DO”, “1I”. Другими словами, некоторые ошибки распознавания символов случаются чаще, чем другие. Для учета этого нужно построить $\mathrm{substCost}(v_i, w_j)$ так, чтобы при вычислении расстояния Левенштейна штраф за сходные символы был меньше, чем за символы несходные:

- для одинаковых символов $\mathrm{substCost}(v_i, v_i) = 0$;
- для различных несходных символов $\mathrm{substCost}(v_i, w_j) = 1$;
- для сходных же символов $\mathrm{substCost}(v_i, w_j) = 0$, либо $0 < \mathrm{substCost}(v_i, w_j) < 1$.

Описанная модификация позволяет уменьшить расстояние, вычисляемое для слов с ошибками в виде сходных символов.

Для некоторых далеких по смыслу слов, например, идентификаторов, расстояние Левенштейна между ними является небольшим. Для исключения рассмотренных случаев ложного отождествления предлагается применять шаблоны слов модели следующего вида:

$$G\left(V\right) = b_1 b_2 \ldots b_k \cdot m_1 m_2 \ldots m_p \cdot e_1 e_2 \ldots e_q.$$

В этих шаблонах заданы обязательные символы в начале, в середине или в конце слова. Если при сравнении символы распознанного слова не удовлетворяют шаблону, то расстояние Левенштейна увеличивается на заданный заранее штраф. Эта модификация позволяет увеличить расстояние между словами, различающимися незначительным числом символов, которые являются признаками для различия слов.

Штраф при отождествлении может быть назначен за несоответствие длин слов V и W:

$$G_2\left(V,\ W\right) = \left\|V\right| - \left|W\right\| > \delta\left(V\right).$$

Сходство между словом модели V и распознанным словом W устанавливают по формуле

$$\mathrm{Sim}\left(\mathrm{V}, \mathrm{W}\right) = d_{\mathrm{LEV}}\left(\mathrm{V}, \mathrm{W}\right) - f_1\left(\mathrm{G}\left(\mathrm{V}\right), \mathrm{W}\right) - f_2\left(\mathrm{G}_2\left(\mathrm{V}, \mathrm{W}\right)\right),$$

где $f_1(\mathrm{G}(\mathrm{V}), \mathrm{W})$ – штраф за несоответствие слова модели V и распознанного слова W, вычисленный с помощью шаблона $\mathrm{G}(\mathrm{V})$; $f_2(\mathrm{G}_2(\mathrm{V}, \mathrm{W}))$ – штраф за несоответствие длин слова модели V и распознанного слова W.

При этом может применяться функция $\mathrm{substCost}(v_i, w_j)$, учитывающая ошибки распознавания для сходных символов. Если штраф отсутствует ($f(\mathrm{G}(\mathrm{V}), \mathrm{W}) = 0$) и сходных символов нет ($\mathrm{substCost}(v_i, w_j) = 0$ или $\mathrm{substCost}(v_i, w_j) = 1$), то $\mathrm{Sim}(\mathrm{V}, \mathrm{W})$ совпадает с расстоянием Левенштейна $d_{\mathrm{LEV}}(\mathrm{V}, \mathrm{W})$. Сходство $\mathrm{Sim}(\mathrm{V}, \mathrm{W})$ также может быть нормализовано аналогично $\rho_{\mathrm{LEV}}(\mathrm{V}, \mathrm{W})$.

Предложенные модификации расстояния Левенштейна позволяют уменьшить число совпадений слов, которые нельзя отождествлять, и одновременно увеличить число совпадений слов, в которых имеются несущественные ошибки распознавания.

5. ОПТИМИЗАЦИЯ РЕАЛИЗОВАННЫХ АЛГОРИТМОВ КЛАССИФИКАЦИИ И ПРИВЯЗКИ ДЕЛОВЫХ ДОКУМЕНТОВ

Реализацию описанных алгоритмов классификации и привязки, основанных на отождествлении слов, мы рассмотрим в качестве объекта оптимизации быстродействия.

Высокоуровневая оптимизация быстродействия алгоритмов классификации и привязки основана на создании описания структуры страницы документа и соответствующих фрагментам документа созвездий. При профилировании реализации алгоритмов на языке C++ вычисление расстояния между словами является “горячей точкой” и занимает 20–50% от общих затрат времени на работу алгоритма (5–15 миллисекунд на один документ различного типа). Другими словами, основное время занимает отождествление слов.

Цель оптимизации функций привязки и классификации обусловлена необходимостью применять эти функции не один раз для изображения документа, а столько раз, сколько имеется описаний типов различных возможных документов.

Очевидно, что при анализе фрагмента документа количество кандидатов на отождествление с текстовыми точками модели может быть существенно меньше, чем при анализе всех распознанных слов страницы. Это обеспечивается штрафными функциями f_1 и f_2, отношениями между точками в созвездии и порогами, применяемыми при вычислении расстояний между V и W с помощью метрик. При обучении моделей классификации (1) на документах более чем на 10 типах деловых документов объем модели составляет более 100 точек, то при задании созвездий в виде простой или вертикальной цепей требуется не более 10 точек. Затраты на работу реализации уменьшаются существенно.

Эффективной оптимизацией вычисления расстояния между словами V и W является вычисление на первом этапе штрафов $\mathrm{Pen}(V, W) = f_1(G(V), W) + f_2(G_2(V, W))$. В случае превышения Pen(V, W) порога $d(V)$. Вычисление по рекуррентной формуле (2), имеющей квадратичную сложность, проводится только в случае, когда $\mathrm{Pen}(V, W) < d(V)$.

Предложенная оптимизация является высокоуровневой и будет давать эффект независимо от архитектурной платформы, на которой исполняется реализация алгоритма. Также была предпринята низкоуровневая оптимизация. Целью этого была реальная потребность. Затраты времени на классификацию и привязку на наборе распознанных слов являются незначительными в схеме обработки, в которой классификация и привязка полей документа проводится один раз для каждого образа документа. Если же классификация и привязка проводятся многократно для нескольких типов документа и многократно применяются к одному набору распознанных слов, то затраты времени увеличиваются вместе с числом применяемых типов. Опишем две оптимизации, направленные на ускорение вычислений для реальных типов центральных процессоров.

Первая оптимизация была связана с вычислением функции substCost(s, c). В реализации алгоритма создавались глобальные (соответствующие документу в целом) и локальные (соответствующие одному слову) таблицы эквивалентности символов, $s \neq c$ для которых $\mathrm{substCost}(s, c) = 0$. При сравнении двух символов проводился поиск в таблице эквивалентности m_nEquChars, имеющий целый 32-разрядный тип, этих символов с учетом перестановки. При профилировании на компьютере Intel(R) Core(TM) i7–4790 CPU3.60 GHz, 16,0 GB, Windows 7 prof 64-bit с помощью ПО MVS Analizer и Intel VTune [15] функция substCost определилась как “горячая точка”. Исходный и ассемблерный коды представлены на рис. 1.

Функция может быть ускорена за счет использования 64-разрядного целого типа (рис. 2). При использовании типа __int64 при компиляции с помощью MSV Compiler для режима x64 количество инструкций для реализации тела цикла уменьшается с 13 до 8. Ускорение на некоторых типах документов составляет 10%.

Другая оптимизация была предназначена для платформы ARM в смартфонах iPhone. Оказалось, что в сложных сценариях многократного применения классификации и привязки реализация доступа к объектам (текстовым точкам, строкам, отношениям между точками) типа get_object(int id, void *pObject) приводят к появлению горячей точки на архитектуре RISC. Это объясняется высокой латентностью операций копирования данных из оперативной памяти. Ускорение достигается при отмене создания новой копии объекта и предоставления непосредственного доступа к объекту.

Описанные алгоритмы были внедрены в SDK Smart Document Engine [16], предназначенный для распознавания гибких деловых документов.

```
int get_substitutionCost_EnableEquChars(uint16_t c1, uint16_t c2) {
      for (int i = 0; i < m_nCountEquChars; i += 2) {
            if ((m_nEquChars[i] == (int)c1 && m_nEquChars[i + 1] == (int)c2) ||
                (m_nEquChars[i] == (int)c2 && m_nEquChars[i + 1] == (int)c1))
                  return 0;
      }
      return c1 != c2;
}
```

```
mov         ecx,dword ptr [rax-4]
cmp         ecx,edx
jne         …
cmp         dword ptr [rax],r8d
je          …
cmp         ecx,r8d
jne         …
cmp         dword ptr [rax],edx
je          …
add         r9d,2
add         rax,8
cmp         r9d,r10d
jl          …
```

Рис. 1. Реализация исходного варианта функции substCost.

```
int get_substitutionCost64(uint16_t c1, uint16_t c2) {
      unsigned __int64 nEquChar;
      unsigned __int64 nC_1 = (((unsigned __int64)c1) << 32) |
                              (unsigned __int64)c2;
      unsigned __int64 nC_2 = (((unsigned __int64)c2) << 32) |
                              (unsigned __int64)c1;
      for (int i = 0; i < m_nCountEquChars; i += 2) {
            nEquChar = *(unsigned __int64*)(m_nEquChars + i);
            if (nEquChar == nC_1 || nEquChar == nC_2)
                  return 0;
      }
      return c1 != c2;
}
```

```
mov         rcx,qword ptr [rax]
cmp         rcx,r10
je          …
cmp         rcx,r9
je          …
add         rax,8
cmp         rax,rdx
jb          …
```

Рис. 2. Реализация оптимизированного варианта функции substCost.

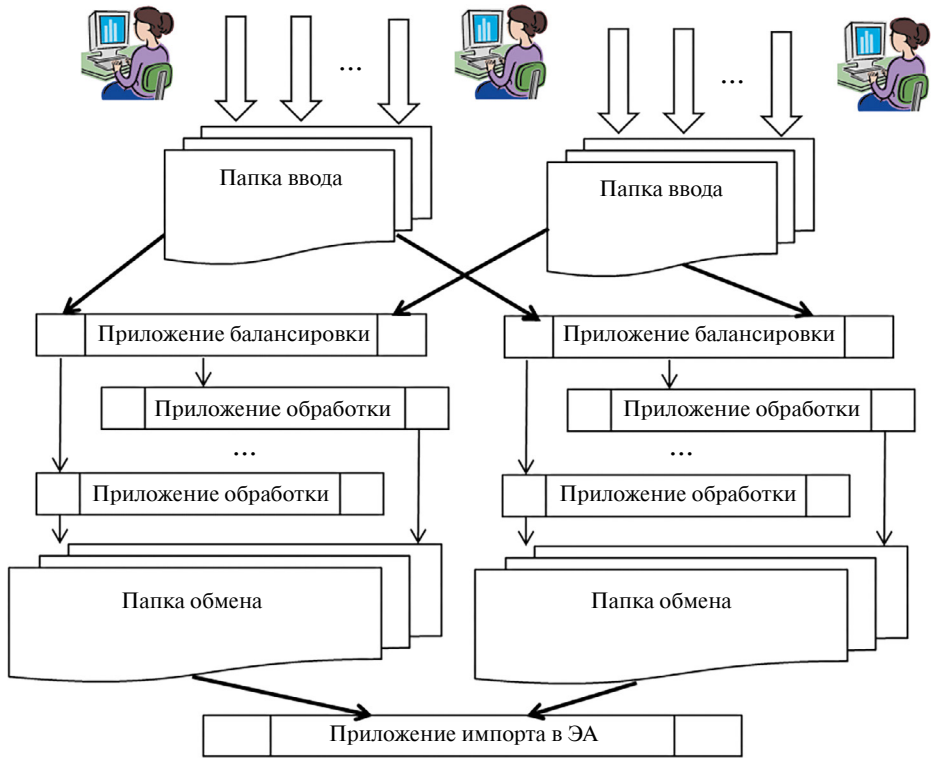

Рис. 3. Параллельная реализация системы сортировки.

Основным режимом работы SDK Smart Document Engine является параллельный режим. Параллелизация обеспечивается автоматически с помощью библиотеки Intel TBB [17].

Рассмотрим другой пример оптимизации быстродействия для задачи сортировки большого потока деловых документов (300 000 страниц за 8 ч). После распознавания OCR Tesseract [18] проводилась классификация для 45 типов известных документов. Отметим, что число классов, равное 45, существенно превышает число классов в публичных датасетах [19, 20]. Высокоуровневая оптимизация алгоритмов классификации проводилась с помощью выбора параметров и представления данных для компоненты OCR Tesseract. Существенный эффект был достигнут за счет ограничения области распознавания в каждой из страниц. Для этого на обучающем множестве была выбрана область, в которой находились необходимые для классификации всех документов текстовые особые точки. Результативной оказалась процедура бинаризации образов страниц перед распознаванием. Первоначальной целью бинаризации мы считали улучшение собственно точности распознавания благодаря снятию сложного фона и морфологическим операциям.

Низкоуровневая оптимизация проводилась с помощью выбора параметров компиляции компоненты Intel C++ Compiler XE15.0. Для компилятора была указана опция оптимизации для архитектуры AVX2. Компилятор Intel позволил оптимизировать быстродействие как Tesseract, так и для всех других компонент системы, прежде всего билатеральный фильтр. Ускорение за счет высокоуровневой и низкоуровневой оптимизации составило более 50%.

Параллелизация была реализована с помощью самостоятельных компонент, позволяющих запустить на нескольких многоядерных узлах по нескольку приложений, обрабатывающих страницы в нескольких потоках, а входной поток страниц назначается этим приложениям согласно некоторому алгоритму балансировки (см. рис. 3). Система была реализована с параллелизмом без использования явного управления.

ЗАКЛЮЧЕНИЕ

В разделах 4 и 5 были рассмотрены примеры высокоуровневой и низкоуровневой оптимизации на примере программ распознавания документов. Описанные методы оптимизации быстродействия программного обеспечения пригодны для более широкого класса приложений для обработки изображений (ПОИ).

Для разработки ПОИ важен выбор готовых или вновь разрабатываемых программных компонент. Этот выбор зависит от формы разработки (проект или собственная разработка). Во всех случаях

уместны оценка сложности выбранных алгоритмов и макетирование ПОИ. Необходимым этапом оптимизации является определение требований к быстродействию программного приложения.

С самого начала разработки важнейшим инструментом высокоуровневой и низкоуровневой оптимизации является профайлер. Этот инструмент применяется для анализа профиля, выбора горячих точек и уточнения ограничений на время выполнения горячих точек. На начальных этапах разработки является полезным анализ исходного кода на эмуляторах будущей вычислительной платформы, в том числе моделирование задержек в подсистеме памяти [21]. Выбор представления изображения может внести существенный вклад в ускорение алгоритма. В качестве примера можно привести использование интегрального представления для извлечения признаков Хаара [22]. При реализации искусственных нейронных сетей для ускорения эффективен выбор размеров слоев сети и представления данных [23, 24].

Расширенные системы инструкций SIMD (MMX, XMM, NEON) эффективно ускоряют алгоритмы обработки изображений и распознавание. Для применения возможностей конкретной платформы могут использоваться как средства компиляторов, так и интринсики. Однако для различных вычислительных платформ различны не только компиляторы, но и наборы интринсиков. Последнее следует учесть при проектировании на предыдущем этапе представления данных, например, для нейронных сетей [23].

В предположении, что обработка изображения занимает время, существенно превышающее время кванта операционной системы, например, это время превышает 100 мс, возможны два способа распараллеливания. Первый способ состоит в использовании средств автоматического распараллеливания [17], второй – непосредственное формирование потоков параллельного управления. Распараллеливание имеет смысл проводить после завершения высокоуровневой и низкоуровневой оптимизации ПОИ.

Предлагаемые методы оптимизации быстродействия программного обеспечения пригодны не только для реализации ПОИ, но и для вычислительных алгоритмов, в которых проводится циклическая обработка информации большого объема.

СПИСОК ЛИТЕРАТУРЫ

1. *Acar U.A., Blelloch G.E., Harper R.* Selective memorization. ACM SIGPLAN Notices. 2003. V. 38. № 1. P. 14–25. https://doi.org/10.1145/640128.604133
2. *Tatarowicz A.L., Curino C., Jones E.P.C. and Madden S.* Lookup Tables: Fine-Grained Partitioning for Distributed Databases. IEEE28th International Conference on Data Engineering. 2012. P. 102–113. https://doi.org/10.1109/ICDE.2012.26
3. *Harris D.M., Harris S.L.* Digital Design and Computer Architecture, 2nd Edition. Morgam Kaufmann is an imprint of Elsevier Inc., Waltham, 2013. 720 p.
4. *Rusiñol M., Frinken V., Karatzas D., Bagdanov A.D., Lladós J.* Multimodal page classification in Administrative document image streams. In: IJDAR. 2014. V. 17. № 4. P. 331–341. https://doi.org/10.1007/s10032-014-0225-8
5. *Slavin O.A., Pliskin E.L.* Method for analyzing the structure of noisy images of administrative documents. Bulletin of the South Ural State University. Ser. Mathematical Modelling, Programming & Computer Software (Bulletin SUSU MMCS). 2022. V. 15. № 4. P. 80–89. https://doi.org/10.14529/mmp220407
6. *Slavin O.A., Farsobina V., Myshev A.V.* Analyzing the content of business documents recognized with a large number of errors using modified Levenshtein distance. Cyber-Physical Systems: Intelligent Models and Algorithms. Springer Nature Switzerland AG. 2022. V. 417. P. 267–279. https://doi.org/10.1007/978-3-030-95116-0
7. *Bellavia F.* SIFT Matching by Context Exposed. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2022. https://doi.org/10.1109/TPAMI.2022.3161853
8. *Bay H., Tuytelaars T., Van Gool Luc.* SURF: Speeded Up Robust Features. Computer Vision and Image Understanding – CVIU. 2003. V. 110. № 3. P. 404–417.
9. *Du X., Wumo P., Bui T.D.* Text line segmentation in handwritten documents using Mumford–Shah model. Pattern Recognition. 2009. V. 42. P. 3136–3145. https://doi.org/10.1016/j.patcog.2008.12.021
10. *Maraj A., Martin M.V., Makrehchi M.* A More Effective Sentence-Wise Text Segmentation Approach Using BERT. In: Lladós J., Lopresti D., Uchida S. (eds) Document Analysis and Recognition – ICDAR2021. Lecture Notes in Computer Science, Springer, Cham. 2021. V. 12824. https://doi.org/10.1007/978-3-030-86337-1_16
11. *Kravets A.G., Salnikova N.A., Shestopalova E.L.* Development of a Module for Predictive Modeling of Technological Development Trends. Cyber-Physical Systems. 2021. P. 125–136. https://doi.org/10.1007/978-3-030-67892-0_11
12. *Sabitov A., Minnikhanov R., Dagaeva M., Katasev A., Asliamov T.* Text Classification in Emergency Calls Management Systems. Cyber-Physical Systems. 2021. P. 199–210. https://doi.org/10.1007/978-3-030-67892-0_17
13. *Deza M.M., Deza E.* Encyclopedia of distances. Springer-Verlag, Berlin, xiv+590 pp. (2009)

14. *Yujian L., Bo L.* A Normalized Levenshtein Distance Metric // IEEE Transactions on Pattern Analysis and Machine Intelligence. V. 29. № 6. P. 1091–1095. https://doi.org/10.1109/TPAMI.2007.1078 (2007)
15. Intel® VTune™ Profiler Performance Analysis Cookbook. https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023–2/overview.html. Accessed 23 Sep. 2023.
16. Smart Document Engine – automatic analysis and data extraction from business documents for desktop, server and mobile platforms. https://smartengines.com/ocr-engines/document-scanner. Accessed 23 Sep. 2023.
17. Intel(R) oneAPI Threading Building Blocks (oneTBB) Developer Guide and API Reference. https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021–10/overview.html. Accessed 23 Sep. 2023.
18. OCR Tesseract. https://github.com/tesseract-ocr/tesseract. Accessed 23 Sep. 2023.
19. NIST Special Database. https://www.nist.gov/srd/nist-special-database-2. Accessed 23 Sep. 2023.
20. Tobacco-3482. https://www.kaggle.com/patrickaudriaz/tobacco3482jpg. Accessed 23 Sep. 2023.
21. *Kravets A.G., Egunov V.* The Software Cache Optimization-Based Method for Decreasing Energy Consumption of Computational Clusters // Energies [Special Issue Smart Energy and Sustainable Environment]. 2022. V. 15. № 20. P. 7509. https://doi.org/10.3390/en15207509
22. *Crow F.C.* Summed-area tables for texture mapping ACM SIGGRAPH Computer Graphics. 1984. V. 18. № 3. P. 207–212.
23. *Trusov A., Limonova E., Nikolaev D., Arlazarov V.V.* 4.6-bit Quantization for Fast and Accurate Neural Network Inference on CPUs // Mathematics. 2024. V. 12. № 5. P. 651. https://doi.org/10.3390/math12050651
24. *Rybakova E.O., Limonova E.E., Nikolaev D.P.* Fast Gaussian Filter Approximations Comparison on SIMD Computing Platforms // Applied Sciences. 2024. V. 14. № 11. P. 4664. https://doi.org/10.3390/app14114664

OPTIMIZATION OF SOFTWARE PERFORMANCE FOR CLASSIFICATION AND LINKING OF ADMINISTRATIVE DOCUMENTS

© 2024 O. A. Slavin[a, b]

[a] *Federal Research Center "Computer Science and Control" of the Russian Academy of Sciences*
44-2 Vavilova str, Moscow, 119333 Russia
[b] *LLC Smart Engines Service*
9 Prospect 60-Letiya Oktyabrya, Moscow, 117312 Russia

The paper discusses technologies for optimizing software performance. Optimization methods are divided into high-level and low-level, as well as parallelization. An algorithm for classifying and linking fields in a recognized image of an administrative document is described. The features of the implementation of classification and linking tasks are listed, consisting of the use of constellations of text feature points and the modified Levenshtein distance. SDK Smart Document Engine and OCR Tesseract were used. Several ways are described to optimize the performance of the functions for classifying and linking document content. Optimization of the performance of the system for sorting a stream of images of administrative documents is also described. The proposed methods for optimizing software performance are suitable not only for implementing image processing algorithms but also for computational algorithms in which cyclic information processing is carried out. The method can be applied in modern CAD systems to analyze the content of recognized textual files.

Keywords: document recognition, flexible document, rigid document, text feature keypoint, acceleration