УДК 004.93'1, 004.272.23, 004.258, 004.272.45

ЭФФЕКТИВНАЯ ДЕТЕКЦИЯ ЛИЦ НА МНОГОЯДЕРНОМ ПРОЦЕССОРЕ EPIPHANY[1]

А.А. Сухинов, Г.Б. Остроброд

В статье рассматривается возможность использования энергоэффективного микропроцессора Epiphany для решения актуальной прикладной задачи — детекции лиц на изображении. Этот микропроцессор представляет собой многоядерную вычислительную систему с распределенной памятью, выполненную на одном кристалле. Из-за малой площади кристалла микропроцессор обладает существенными аппаратными ограничениями (в частности, он имеет всего 32 килобайта памяти на ядро), которые ограничивают выбор алгоритма и затрудняют его программную реализацию. Для детекции лиц адаптирован известный алгоритм, основанный на каскадном классификаторе, использующем LBP-признаки (Local Binary Patterns). Показано, что микропроцессор Epiphany, имеющий 16 ядер, может на этой задаче в 2,5 раза обогнать одноядерный процессор персонального компьютера той же тактовой частоты, при этом потребляя лишь 0,5 ватта электрической мощности.

Ключевые слова: детекция лиц, локальные бинарные шаблоны, параллельная обработка данных, специализированные микропроцессоры, распределенная память.

Введение

Совершенствование технологий производства процессоров в плане миниатюризации привело к повсеместному использованию многоядерных микропроцессоров, в том числе в мобильных приложениях. Типичные многоядерные процессоры с точки зрения прикладного программирования являются системами с общей памятью, однако их низкоуровневая архитектура — это система с распределенной памятью; каждое ядро имеет свою кэш-память; кэш-памяти отдельных ядер синхронизируются при помощи того или иного протокола когерентности кэшей [1, 2]. Такой подход облегчает создание программного обеспечения, однако существенно усложняет процессор, увеличивая требуемую площадь кристалла и энергопотребление. Кроме того, снижается возможная эффективность использования процессора; при правильном программировании можно получить большее быстродействие от системы с распределенной памятью, чем от «ненастоящей» системы с общей памятью [3].

Энергоэффективный микропроцессор Epiphany производства Adapteva является образцом другого подхода: каждое ядро имеет память небольшого объема, явно управляемую прикладной программой. Ядра связаны между собой сетями передачи данных.

Цель данной статьи — демонстрация возможности эффективного использования микропроцессоров Epiphany для решения актуальной прикладной задачи — детекции лиц на изображении. Будет показано, что микропроцессор Epiphany, имеющий 16 ядер, может на этой задаче в 2,5 раза обогнать одноядерный процессор персонального компьютера той же тактовой частоты, при этом потребляя лишь 0,5 ватта электрической мощности.

Программный код, описанный в статье, доступен на странице проекта в Интернете [4].

[1]Статья рекомендована к публикации программным комитетом Международной научной конференции «Параллельные вычислительные технологии — 2014».

1. Архитектура Epiphany

Многоядерные микропроцессоры Epiphany представляют собой IP (intellectual property) блоки, предназначенные для разработки систем-на-кристалле. Блок может быть сконфигурирован на количество ядер от 1 до 4096. Типичная тактовая частота — 1 ГГц, объем памяти ядра — 32 или 64 килобайта. Кроме того, возможно подключение внешней памяти объемом до 2-х гигабайт. Микропроцессор оптимизирован для снижения энергопотребления и занимаемой площади кристалла: при быстродействии 1 GFLOPS одно ядро имеет площадь 0,3 мм^2 и энергопотребление всего 14 мВт.

Архитектура Epiphany активно совершенствуется, поэтому у актуальных экземпляров микропроцессоров характеристики могут отличаться от приведенных.

Малый объем памяти ядра и отсутствие виртуальной памяти не позволяют запускать на микропроцессорах Epiphany популярные операционные системы, поэтому микропроцессор позиционируется как ускоритель вычислений для мобильных приложений, либо как центральный процессор устройства, не имеющего операционной системы.

Тестовые экземпляры микропроцессоров Epiphany доступны в виде готовых модулей Parallela [5], представляющих собой компьютер архитектуры ARM, имеющий размеры кредитной карты. В таком модуле Epiphany установлен в качестве дополнительного процессора, поэтому далее будем называть его *сопроцессором*. Каждое ядро Epiphany имеет:

- 32 или 64 килобайта *локальной памяти*, разделенной на банки по 8 килобайт; каждый банк в один такт может обслуживать только одного потребителя.
- *Арифметико-логическое устройство*, способное за один такт выполнять операции над 32-битными числами (целыми или с плавающей точкой).
- *64 регистра общего назначения.*
- *Двухканальный контроллер прямого доступа к памяти* (DMA), способный самостоятельно выдавать команды пересылки данных.
- *2 таймера* для подсчета различных событий (например, тактов сопроцессора).
- *Контроллер прерываний*, позволяющий устанавливать обработчики событий.
- *Контроллер пересылки данных между ядрами* (маршрутизатор).

Логически память всех ядер объединена в единое 32-битное адресное пространство, что позволяет на программном уровне единообразно передавать данные между любыми ячейками памяти любой пары ядер. В адресное пространство также включена внешняя память и регистры ядер, что позволяет из одного ядра модифицировать регистры другого ядра. Единое адресное пространство позволяет ядру выполнять не только «свой» код, но и код, расположенный в памяти другого ядра или даже во внешней памяти.

Физически ядра связаны в прямоугольную решетку тремя сетями передачи данных:

- *Сеть записи* передает пакеты, содержащие адрес и данные. Когда пакет достигает целевого ядра, данные записываются в требуемые ячейки памяти или регистры.
- *Сеть чтения* передает пакеты, содержащие адрес чтения и адрес записи данных. Такие пакеты направляются ядру, содержащему адрес чтения. По достижению цели необходимые данные считываются из памяти или регистров, и формируется ответный пакет, передаваемый через сеть записи. Интересной особенностью описанного подхода является возможность пересылки данных между двумя адресами, внешними по отношению к ядру, выдавшему команду на такую пересылку.

— *Транзитная сеть* передает данные от границы до границы блока, состоящего из 4×4 ядер. Трафик транзитной сети не затрагивает трафик других сетей, что дает возможность формирования блоков размерами 4×4 ядра, работающих в реальном времени.

Хотя логически передача данных возможна между любой парой ядер, физически за один такт данные передаются только между соседними ядрами по горизонтали или вертикали. Это позволяет избежать необходимости синхронизации фазы тактовой частоты между ядрами, расположенными далеко друг от друга на кристалле. Вместо этого тактовая частота распространяется волной от точки ее ввода в кристалл, что снижает энергопотребление.

Команда записи данных завершается за один такт; никаких действий, подтверждающих запись, не производится. Алгоритмы маршрутизации гарантируют, что пакеты записи будут доставлены получателю (включая внешнюю память) в том же порядке, в котором они были посланы отправителем (при условии, что у этих пакетов один и тот же отправитель).

В связи с необходимостью прохождения ответного пакета, чтение из памяти другого ядра производится значительно медленнее, чем запись в память другого ядра.

На границах кристалла сети передачи данных соединены с электрическими выводами сопроцессора. Это позволяет подключить внешнюю память, основной процессор, а также соединить несколько сопроцессоров в прямоугольную решетку, тем самым увеличив количество ядер в адресном пространстве.

2. Прикладное программирование сопроцессора Epiphany

Для программирования сопроцессора используется адаптированный компилятор GCC, принимающий исходный код на языке Си. Для каждого ядра используется собственный исходный код. Двоичный файл, предназначенный для загрузки в сопроцессор, представляет собой образ памяти всех ядер. Так как регистры ядер отображены в адресное пространство, такой подход позволяет задать сопроцессору нужное состояние на момент старта.

Стандартная библиотека языка Си присутствует, однако ее исполняемый код и структуры данных находятся во внешней памяти, что означает медленность работы функций стандартной библиотеки, а также невозможность использования функций `malloc` и `free` для работы с локальной памятью. Поэтому память каждого ядра целесообразно распределять статически при помощи файла LDF (Linker Definition File), содержащего указания компоновщику по размещению в памяти всех глобальных объектов программы. При программировании следует учитывать следующие факторы:

— Сопроцессор Epiphany может выполнять чтение, запись, и арифметические операции только с 32-битными числами (целыми или с плавающей точкой). Операции над другими типами данных (в том числе меньшего размера) эмулируются компилятором, и поэтому требуют нескольких тактов сопроцессора.

— Невыровненный запрос к памяти (адрес не кратен размеру типа данных) приводит к остановке программы.

— При этом у Epiphany нет проблем с управляющей логикой: вызов функции и возврат из нее происходят быстро, условные переходы не вызывают серьезного снижения быстродействия (их предсказание делается компилятором в статическом режиме).

В связи с указанными особенностями было решено реализовать алгоритм детекции лиц (ту его часть, что работает на сопроцессоре) в рамках следующих ограничений:

— Изображения хранятся в виде восьмибитных целых чисел; все остальные данные хранятся в виде выровненных в памяти 32-битных целых чисел;

— Для вычислений используются только 32-битные целые числа.
— В экземпляре сопроцессора, с которым мы работали, не было операций целочисленного умножения и деления, поэтому из арифметических операций можно было использовать только сложение, вычитание, и битовые операции.

Как будет показано ниже, при правильном выборе алгоритма и его реализации указанные ограничения не приводят к существенному усложнению или замедлению кода.

3. Алгоритм детекции лиц

Для реализации было решено взять один из нескольких алгоритмов детекции лиц, использованных в популярной библиотеке обработки изображений OpenCV [6]. Реализация распространенного алгоритма позволит объективно сравнить производительность процессора Epiphany с другими процессорами, на которых работает OpenCV. В указанной библиотеке алгоритмы детекции лиц имеют следующую структуру:

1. По исходному черно-белому изображению строится *пирамида*, состоящая из *слоев* (изображений) уменьшающегося разрешения. Слои высокого разрешения нужны для детекции лиц маленького размера; слои низкого разрешения используются для детекции крупных лиц. Типичное отношение размеров соседних слоев составляет 1,2.
2. Производится сканирование каждого слоя пирамиды *окном* небольшого размера (например, 24×24 пикселя). При сканировании окна располагаются с перекрытием.
3. Для каждого положения окна вызывается *классификатор*, который решает, является ли предоставленный ему фрагмент изображения лицом. Для принятия решения используются числовые *признаки*, получаемые на основе пикселей окна.
4. Производится *группировка детекций*: окна, получившие положительные отклики классификатора, группируются по геометрическому сходству. Группы, имеющие менее определенного числа элементов (например, трех), отбрасываются. Для оставшихся групп вычисляются средние окна, которые признаются положительными детекциями.

Наибольшее время тратится на детекцию мелких лиц (обработку слоев пирамиды высокого разрешения), поэтому минимальный размер детектируемых лиц нужно ограничивать.

Интересно отметить, что алгоритмы данной структуры ощутимо проигрывают по точности человеческому зрению, несмотря на высокую точность используемых в них классификаторов. Причиной такого проигрыша является недостаточность информации в окне для выполнения решения лицо/не лицо; человек для такого решения использует контекст — большое количество дополнительной информации о наблюдаемой сцене.

Основным элементом алгоритма детекции лиц является классификатор, так как именно он принимает решение лицо/не лицо для текущего окна. Классификатор представляет собой статистическую математическую модель, которая параметризуется автоматически на основе обучающей выборки, в рамках процесса, называемого *машинным обучением* [7]. Машинное обучение выходит за рамки этой статьи, поэтому все коэффициенты классификатора, упомянутые ниже, будем считать известными.

Ввиду сильно ограниченного объема памяти сопроцессора, среди алгоритмов детекции лиц библиотеки OpenCV мы выбрали для реализации самый экономичный в плане памяти. Таковым оказался алгоритм, основанный на LBP-признаках (Local Binary Patterns [8]).

LBP-признак (в реализации OpenCV) — это целое число от 0 до 255, которое ставится в соответствие прямоугольной области изображения (являющейся подобластью окна детек-

ции), ширина и высота которой кратны трем:

$$\begin{array}{rcccccccc} \mathrm{LBP}(X,Y,W,H) & \stackrel{\text{def}}{=} & 2^7\cdot[s_{00}\geqslant s_{11}] & + & 2^6\cdot[s_{10}\geqslant s_{11}] & + & 2^5\cdot[s_{20}\geqslant s_{11}] & + \\ & + & 2^0\cdot[s_{01}\geqslant s_{11}] & & + & & 2^4\cdot[s_{21}\geqslant s_{11}] & + \\ & + & 2^1\cdot[s_{02}\geqslant s_{11}] & + & 2^2\cdot[s_{12}\geqslant s_{11}] & + & 2^3\cdot[s_{22}\geqslant s_{11}] & . \end{array} \tag{1}$$

Здесь:

(X,Y) — координаты левого верхнего пикселя рассматриваемой области;

$(W,H)=(3w,3h)$ — размеры области для вычисления LBP-признака;

$s_{i,j}\stackrel{\text{def}}{=}\sum\limits_{x=0}^{w-1}\sum\limits_{y=0}^{h-1}p(X+iw+x,Y+jh+y)$ — сумма пикселей по прямоугольнику;

$p(x,y)$ — значение пикселя изображения (столбец x, строка y);

$$[a\geqslant b]\stackrel{\text{def}}{=}\begin{cases}1, \text{ если } a\geqslant b;\\ 0, \text{ иначе.}\end{cases}$$

Смысл LBP-признака простой: рассматриваемая область делится на 3×3 прямоугольника, и отношения яркостей (больше-меньше) восьми нецентральных прямоугольников к центральному кодируются в виде восьмибитного двоичного числа. Следует заметить, что LBP-признаки здесь используются не для формирования гистограмм [8], а как замена признаков Хаара, часто используемых в алгоритмах детекции лиц [9].

Сам классификатор состоит из нескольких *стадий*, и поэтому называется *каскадным*. Только положительное прохождение всех стадий классификатора означает положительный результат классификации («обнаружено лицо»). Такой ход принятия решений позволяет отсеивать большинство окон на ранних стадиях. Положительное прохождение j-й стадии ($j=1,\ldots,m$) определяется следующим неравенством:

$$\sum_{i=1}^{n^j}w_i^j\geqslant T^j,\quad w_i^j\stackrel{\text{def}}{=}\begin{cases}P_i^j, \text{ если } \mathrm{LBP}\left(X_i^j,Y_i^j,W_i^j,H_i^j\right)\in\mathbf{S}_i^j;\\ N_i^j, \text{ иначе.}\end{cases} \tag{2}$$

Здесь:

n^j — количество признаков, используемых в стадии j классификатора;

w_i^j — вес, который ставится в соответствие признаку номер i стадии j;

$\mathbf{S}_i^j$ — множество «желательных» значений признака для области $\left(X_i^j,Y_i^j,W_i^j,H_i^j\right)$;

P_i^j — вес, который приобретает область с «желательным» значением признака;

$N_i^j<P_i^j$ — вес, который дается области с «нежелательным» значением признака;

T^j — вес, который должен быть накоплен для успешного прохождения стадии.

Все коэффициенты классификатора собраны в следующем выражении:

$$\mathbf{K}_{\mathrm{CV}}\stackrel{\text{def}}{=}\left(\left(X_i^j,Y_i^j,W_i^j,H_i^j,\mathbf{S}_i^j,P_i^j,N_i^j\right)_{i=1}^{n^j},T^j\right)_{j=1}^{m}. \tag{3}$$

Классификатор фронтальных лиц в OpenCV имеет $m=20$ стадий. На начальной стадии вычисляются 3 признака, на последней — 10 признаков. Суммарный объем коэффициентов в выражении (3) составляет 8 килобайт, однако профилирование программы, использующей библиотеку OpenCV, показало, что загруженный классификатор занимает 300 килобайт оперативной памяти. Это вызвано использованием нескольких буферов памяти (хранящих стадии классификатора, признаки, веса и другие данные), ссылающихся друг на друга при помощи индексов и указателей, использованием классов с таблицами виртуальных функций, а также хранением данных, не нужных для решения данной задачи.

4. Программная реализация детектора лиц

4.1. Расположение структур данных

Прежде всего нужно решить, где располагать *данные классификатора*. Были проработаны следующие варианты:

Данные классификатора во внешней памяти. Это самый простой для реализации вариант. Однако ввиду того, что вычисления, связанные с классификацией (выражения (1), (2)), достаточно простые, а чтение данных из внешней памяти выполняется медленно, большую часть времени ядра будут простаивать в ожидании получения данных. Ситуация усугубляется тем, что все ядра требуют данные одновременно, конкурируя за канал обмена сопроцессора с внешней памятью.

Стадии классификатора распределены между ядрами. При таком подходе каждое окно детекции обрабатывается ядрами сопроцессора по конвейеру. Однако обработка большинства окон прерывается на ранних стадиях; для того чтобы ядра были постоянно обеспечены работой, требуется, чтобы для первых стадий классификатора было выделено больше ядер, чем для последних стадий. Требуется нетривиальная синхронизация и балансировка загрузки (текстурированные участки изображения проходят больше стадий, чем гладкие), что представляется сложным для программирования.

Копия классификатора в каждом ядре. Это самый предпочтительный вариант, так как ядра могут выполнять независимые подзадачи — классификацию различных окон. Однако в нашем распоряжении был сопроцессор, имеющий всего 32 килобайта памяти на ядро, и для хранения классификатора оставалось не более 8-ми килобайт, что гораздо меньше измеренного объема данных классификатора OpenCV (300 килобайт). Как будет показано далее, нам удалось очень компактно сохранить данные классификатора, поэтому именно этот вариант был выбран для реализации.

Затем надо определиться со способом формирования и хранения *пирамиды изображений*:

Размещение в общей памяти только исходного изображения. Каждое ядро обрабатывает фрагмент этого изображения (будем называть его *тайлом*), и строит в локальной памяти соответствующую часть пирамиды. Тайлы должны быть маленькими, чтобы уместиться в памяти ядра, и должны перекрываться, чтобы не потерялись возможные детекции на их стыках. При формировании слоя пирамиды более низкого разрешения ширина перекрытия оказывается недостаточной; ядра должны обмениваться данными, передавая друг другу недостающие пиксели для обработки стыков. Однако все изображение может не вместиться в сопроцессор, часть пикселей, требуемых для обработки границ тайлов, отсутствует, и неясно, как эти пиксели вычислять.

Вся пирамида находится в общей памяти. Вначале сопроцессор строит пирамиду из исходного изображения, располагая ее в общей памяти, а затем слои обрабатываются перекрывающимися тайлами. Это представляется достаточно эффективным, так как обработка тайла занимает гораздо больше времени, чем требуется для его пересылки. Был выбран данный подход, однако соответствующий код не вместился в память ядра сопроцессора, поэтому построение пирамиды выполняется центральным процессором.

В итоге в общей памяти располагаются следующие структуры данных (рис. 1): пирамида изображений, очередь заданий (описаны далее) со счетчиками взятых и выполненных заданий, и данные классификатора. Также в общей памяти располагается счетчик запускаемых ядер сопроцессора. Со стороны Epiphany счетчики защищены мьютексом.

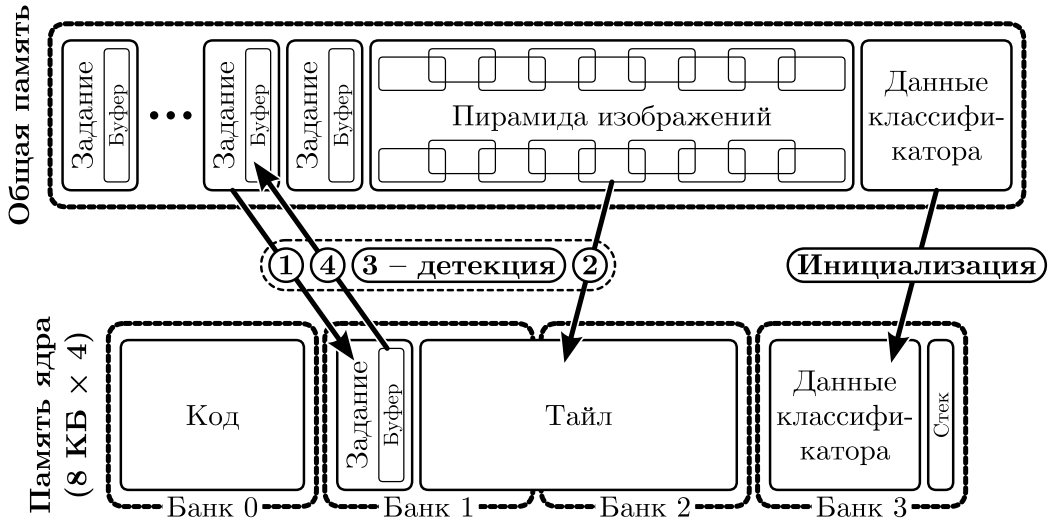

Рис. 1. Схема распределения памяти. Стрелками показаны пересылки данных

Память каждого ядра сопроцессора распределена следующим образом (рис. 1): 8 килобайт выделено на код, 16 килобайт на копию задания и обрабатываемый тайл, и 8 килобайт на данные классификатора и стек выполнения программы.

4.2. Параллельный алгоритм

В общих чертах, *алгоритм работы центрального процессора* следующий:

1. Записать в общую память классификатор.
2. Построить в общей памяти пирамиду изображений.
3. Логически разделить каждый слой пирамиды на частично перекрывающиеся прямоугольные тайлы размерами примерно 128×128 пикселей, и поставить в соответствие каждому тайлу задание, состоящее из координат тайла на изображении, адреса тайла в памяти, и буфера для записи результатов детекции лиц в тайле.
4. Установить счетчик запускаемых ядер в ненулевое значение и ждать, когда счетчик выполненных заданий достигнет общего количества заданий.
5. Выполнить группировку детекций, находящихся в очереди заданий.

Алгоритм работы ядра сопроцессора следующий (рис. 1): когда счетчик запускаемых ядер станет ненулевым, уменьшить его на единицу, скопировать данные классификатора в локальную память, и перейти в цикл обработки заданий. Задание выполняется так:

1. Наращивается счетчик взятых заданий, и соответствующее задание копируется в локальную память. Когда все задания взяты, цикл обработки заданий завершается.
2. Тайл, расположение которого указано в задании, копируются в локальную память.
3. Выполняется детекция лиц в тайле.
4. Результаты копируются в общую память, наращивается счетчик выполненных заданий.

Пересылки данных между внешней памятью и ядром производятся при помощи контроллера DMA. Между ядрами пересылаются только данные, нужные для работы мьютекса.

4.3. Реализация пирамиды изображений

Для пирамиды с равномерным шагом масштаба размеры (w_i, h_i) слоя номер $i = 1, \ldots, n$ определяются следующими соотношениями:

$$(w_1, h_1) = \frac{(w, h)}{s}, \quad (w_{i+1}, h_{i+1}) = \frac{(w_i, h_i)}{k}, \tag{4}$$

где (w, h) — размеры исходного изображения; $s \geqslant 1$ — масштаб начального слоя (определяется требуемым минимальным размером детектируемого лица); $k > 1$ — отношение масштабов соседних слоев (типичное значение: $k \approx 1{,}2$). Количество слоев n обычно наращивается, пока в последнем слое помещается хотя бы одно окно детекции.

Пирамида изображений должна иметь достаточно малый шаг масштаба (параметр k близкий к единице), чтобы для любого лица нашелся слой, на котором оно будет иметь размер, близкий к оптимальному (тому размеру, на котором был обучен классификатор). Для пирамиды с равномерным шагом масштаба максимально возможное отклонение размера проверяемого лица от оптимального составляет $\sqrt{k}$. Чем ближе это значение к единице, тем выше качество детекции лиц, но тем больше требуется слоев.

Очередной слой пирамиды можно получить путем передискретизации исходного изображения или одного из вычисленных ранее слоев большего разрешения. Чтобы уменьшить алиасинг [10] передискретизацию целесообразно выполнять интегрированием пикселей исходного изображения по площади пикселей уменьшенного изображения (метод Area в терминологии OpenCV). Вычислительная сложность подобной процедуры примерно пропорциональна сумме площадей исходного и уменьшенного изображений.

Для снижения объема вычислений возникает желание вычислять каждый слой пирамиды на основе предыдущего. К сожалению, это приводит к появлению муара [10], выражающегося в виде волнообразного изменения четкости изображения с периодом $1/(k-1)$. Сам по себе муар — явление незначительное, однако при последовательном уменьшении изображений он накапливается, существенно снижая качество детекции.

В библиотеке OpenCV проблемы алиасинга и муара при построении пирамиды решены следующим образом: каждый слой пирамиды строится независимо от остальных слоев путем масштабирования исходного изображения методом Area. В итоге время построения пирамиды в OpenCV составляет $T_{\text{OpenCV}} = O(w \cdot h \cdot n)$.

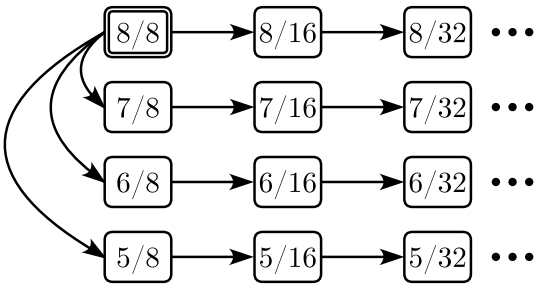

Рис. 2. Схема построения пирамиды изображений

В нашей реализации пирамида имеет фиксированные масштабные множители и строится следующим образом (рис. 2): исходное изображение разбивается на блоки размерами 8×8 пикселей, по которым вычисляются уменьшенные блоки размерами 7×7, 6×6 и 5×5 пикселей. За один проход получаются сразу 3 дополнительных слоя пирамиды, име-

ющих размеры 7/8, 6/8 и 5/8 от оригинала. Любой следующий слой, имеющий номер i, получается двукратным уменьшением (усреднением блоков 2×2 пикселя) слоя номер $i - 4$.

При таком подходе отношения масштабов соседних слоев немного различаются, однако это не влияет на качество детекции лиц. Последовательное уменьшение размера в 2 раза не приводит к накоплению погрешности и появлению муара (его период равен 1). Если нужно увеличить минимальный размер детектируемых лиц, следует при детекции пропустить нужное количество слоев высокого разрешения.

Время построения пирамиды описанным способом составляет $T_{\text{PyrEP}} = O(w \cdot h)$. Замеры показали, что OpenCV успевает построить в среднем всего два слоя пирамиды за время, требуемое для построения всей пирамиды (примерно 20 слоев) в нашей реализации.

4.4. Реализация LBP-признаков

Для быстрого вычисления сумм $s_{i,j}$ в выражении (1) библиотека OpenCV использует интегральное преобразование изображения [11]. Сумма по любому прямоугольнику вычисляется за 4 чтения из памяти и 3 арифметических операции. Интегральное изображение вычисляется за один проход по слою пирамиды и хранится в виде чисел типа `int`.

Хранение четырех байт на пиксель интегрального изображения оказалось неприемлемым из-за ограниченной памяти сопроцессора Epiphany, однако мы нашли способ эффективного вычисления LBP-признаков без использования интегрального преобразования. Для этого достаточно аппроксимировать суммы $s_{i,j}$ четырьмя слагаемыми (аналогично численному интегрированию методом центральных прямоугольников). Отличия в результатах детекции лиц, вызванные такой аппроксимацией, оказались несущественными.

Можно было бы оставить от каждой суммы всего по одному центральному пикселю, и принять такие признаки за некоторые новые LBP-признаки, но такой подход уже требовал переобучения классификатора для достижения точности детекции OpenCV. Ценность нашего кода значительно выше, если он может использовать готовые обученные классификаторы из OpenCV, поэтому мы отказались от замены каждой суммы $s_{i,j}$ на одно слагаемое.

Подвыражения вида $2^7 \cdot [s_{00} \geqslant s_{11}]$ из выражения (1) в OpenCV реализованы при помощи условных операторов. В нашей реализации они вычисляются взятием знакового бита разности $s_{00} - s_{11}$, и сдвигом этого бита вправо на нужное количество разрядов, что дает дополнительный выигрыш в быстродействии.

4.5. Реализация каскадного классификатора

Требуется не только уменьшить объем данных классификатора, но и сделать структуру данных такой, чтобы исполняемый код был компактным и быстродействующим. Фактически нужно минимизировать сумму объемов данных классификатора и исполняемого кода, использующего эти данные: вместе они должны составлять менее 16-ти килобайт.

Было решено разместить все данные классификатора в одном буфере памяти в том порядке, в котором классификатор их использует. Это позволило избавиться от всех ссылок, которые использованы в OpenCV для связи буферов. В итоге мы пришли к архитектуре классификатора, напоминающей виртуальную машину. Данные классификатора представлены в виде последовательности команд. Каждая команда имеет фиксированный размер и содержит в себе необходимые для выполнения параметры, представленные в виде целых чисел. Возможны 3 вида команд:

— *Вычислить LBP-признак*, и прибавить его оценку к текущей оценке классификатора.
— *Команда окончания стадии*: сравнить текущую оценку с числом, записанным в команде. Если текущая оценка меньше, вернуть «не лицо», иначе обнулить текущую оценку.
— *Команда окончания классификации*: вернуть «лицо».

Для минимизации количества данных, содержащихся в команде 1, были выполнены описанные далее модификации. Прежде всего, условие (2) можно переписать следующим образом:

$$\sum_{i=1}^{n^j} w_i^j \geqslant T^j - \sum_{i=1}^{n^j} N_i^j, \quad w_i^j \stackrel{\text{def}}{=} \begin{cases} P_i^j - N_i^j, \text{ если LBP}\left(X_i^j, Y_i^j, W_i^j, H_i^j\right) \in \mathbf{S}_i^j; \\ 0, \text{ иначе.} \end{cases} \tag{2'}$$

Вводя новые коэффициенты

$$T_*^j \stackrel{\text{def}}{=} T^j - \sum_{i=1}^{n^j} N_i^j, \quad V_i^j \stackrel{\text{def}}{=} P_i^j - N_i^j, \tag{5}$$

и размещая координаты $\left(X_i^j, Y_i^j, W_i^j, H_i^j\right)$ в четырех байтах числа F_i^j типа `int`, получим условие прохождения стадии, не имеющее весов N_i^j:

$$\sum_{i=1}^{n^j} V_i^j \cdot \left[\text{LBP}\left(F_i^j\right) \in \mathbf{S}_i^j\right] \geqslant T_*^j, \quad \left[\text{LBP}\left(F_i^j\right) \in \mathbf{S}_i^j\right] \stackrel{\text{def}}{=} \begin{cases} 1, \text{ если LBP}\left(F_i^j\right) \in \mathbf{S}_i^j; \\ 0, \text{ иначе.} \end{cases} \tag{6}$$

В итоге исходный набор коэффициентов (3) сократился до

$$\mathbf{K}_{\text{EP}} \stackrel{\text{def}}{=} \left(\left(F_i^j, \mathbf{S}_i^j, V_i^j\right)_{i=1}^{n^j}, T_*^j\right)_{j=1}^{m}. \tag{7}$$

Наибольший объем памяти (по 256 бит) здесь занимают множества $\mathbf{S}_i^j$ «желательных» значений признаков $\text{LBP}\left(F_i^j\right)$. Эти множества можно сжать, однако это приведет к значительному замедлению программы, поэтому они были оставлены как есть. Коэффициенты V_i^j и T_*^j определены с точностью до произвольного сомножителя. Если их привести к достаточно большому диапазону (например, $[0; 10^5]$), и округлить, то результаты классификации не изменятся. Поэтому алгоритм (6) может быть полностью целочисленным.

Так как $\left[\text{LBP}\left(F_i^j\right) \in \mathbf{S}_i^j\right] \in \{0; 1\}$, можно избавиться от нежелательного умножения в выражении (6). В данном случае умножение можно заменить унарным минусом и битовым «И», так как число -1 в двоично-дополнительной записи имеет все единичные биты:

$$\sum_{i=1}^{n^j} V_i^j \,\&\, \left(-\left[\text{LBP}\left(F_i^j\right) \in \mathbf{S}_i^j\right]\right) \geqslant T_*^j. \tag{6'}$$

Подход, описанный в данном разделе, позволил сохранить данные классификатора в непрерывном участке памяти объемом 6,15 килобайта, что лишь на 10% больше объема коэффициентов (7). Это означает, что разработанная структура данных имеет всего 10% накладных расходов. Напомним, что OpenCV требовалось 300 килобайт для хранения классификатора.

Следует также упомянуть особенности сканирования изображения окном детекции. В OpenCV несколько первых слоев сканируются с шагом 2 пикселя в горизонтальном и вертикальном направлениях (четверть возможных положений окна). В то же время остальные слои сканируются плотно — с шагом один пиксель. Такой подход, по-видимому, сделан с целью повышения быстродействия, однако он приводит к снижению качества детекции

маленьких лиц (имеющих размер от 24-х до 48-ми пикселей), а также к проблемам с группировкой детекций. Из-за различной плотности сканирования на слоях низкого разрешения группы детекций имеют в среднем в 4 раза больше элементов, чем на слоях высокого разрешения. Поэтому не удается подобрать «хорошее» значение для параметра, соответствующего минимальному количеству детекций в группе: мы получаем либо много ложноположительных детекций для крупных лиц, либо много пропусков маленьких лиц.

В нашей реализации окно детекции на всех слоях пирамиды проходит по пикселям в соответствии с узором «шахматная доска» (левый верхний угол окна проходит по половине пикселей). В итоге на предмет наличия лиц проверяется на 40% больше окон, чем проверяется в OpenCV. Это приводит к лучшим результатам детекции (как за счет большего числа окон, так и за счет равномерного их распределения по слоям).

5. Результаты

Несмотря на то, что код, написанный для сопроцессора Epiphany, может быть практически без изменений запущен на центральном процессоре, для сравнения с OpenCV имело смысл сделать еще один, упрощенный вариант кода, предназначенный только для центрального процессора. Этот упрощенный вариант не имеет очереди заданий, разбиения изображения на тайлы и пересылки данных. При запуске на центральном процессоре упрощенный код работает немного быстрее полного кода, и потребляет меньше памяти.

Были проведены измерения на изображениях, имеющих различное разрешение и лица разного размера. Наблюдаемые во всех случаях результаты и соотношения были примерно одинаковыми, поэтому далее рассматривается только одно изображение.

На рис. 3 показаны результаты детекции лиц на изображении размерами 1600×1065 пикселей. Результаты OpenCV показаны квадратами, результаты нашей реализации показаны окружностями. Наша реализация нашла больше лиц за счет упомянутого выше более плотного сканирования слоев пирамиды высокого разрешения. При детекции крупных лиц (48 пикселей и более) наши результаты практически не отличаются от OpenCV.

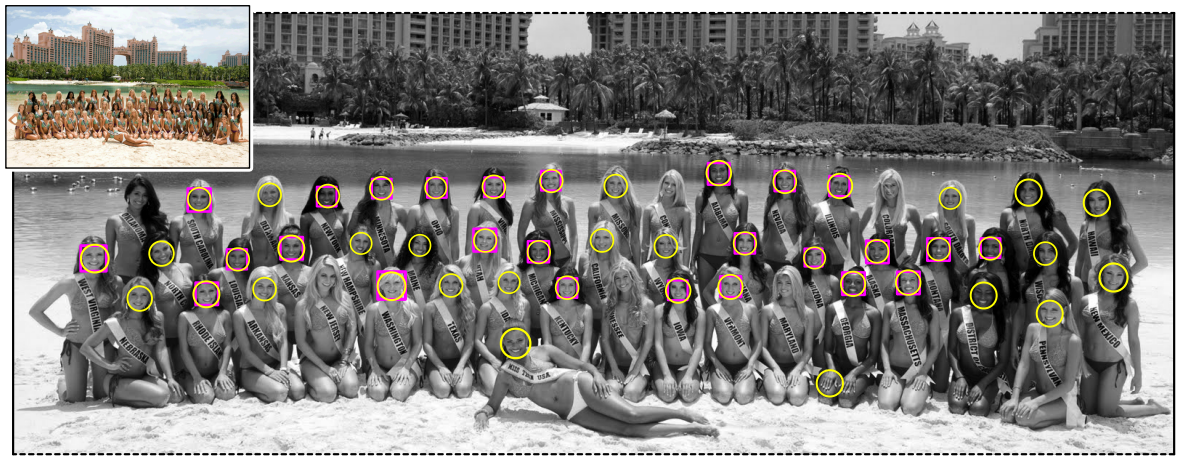

Рис. 3. Фрагмент изображения с результатами детекции лиц (исходное изображение показано слева). Квадраты — результат OpenCV; окружности — результат нашей программы

Сравним количество потребляемой памяти у обеих реализаций. Профилирование показало, что наша реализация при обработке указанного изображения потребовала 3,8 мегабайта дополнительной памяти, в то же время OpenCV потребовала более 8,5 мегабайт.

Приходим к выводу, что представленная реализация потребляет примерно в 2,2 раза меньше оперативной памяти, чем популярная библиотека OpenCV.

Время обработки изображения у OpenCV и у нашей реализации оказалось примерно одинаковым — 0,2 секунды на процессоре Intel Core I7 (2 ядра, 4 потока исполнения, тактовая частота 2,4 ГГц). Учитывая, что наш детектор проверяет на 40% больше окон, мы можем заключить, что наша реализация примерно на 40% быстрее, чем OpenCV (построение пирамиды составляет малую часть от времени детекции), или же обеспечивает лучшую точность детекции при том же времени работы.

Перейдем к экспериментам с Epiphany. Мы работали с прототипной системой, содержащей центральный процессор AMD E-350 (2 ядра, тактовая частота 800 МГц), и раннюю модификацию сопроцессора Epiphany, имеющую 16 ядер и тактовую частоту 400 МГц.

Эксперимент 1: детекция на одном ядре сопроцессора. Построение пирамиды на центральном процессоре потребовало 0,038 секунды. Время детекции лиц, измеренное на сопроцессоре: 16,08 секунды (сюда не включено время синхронизации и пересылки данных). Общее время работы алгоритма, измеренное на центральном процессоре: 16,46 секунды.

Эксперимент 2: детекция 16-ти ядрах сопроцессора. Построение пирамиды на центральном процессоре заняло 0,039 секунды; время детекции лиц, измеренное на ядрах сопроцессора (без учета времени синхронизации и пересылки данных): $1{,}0 \pm 0{,}01$ секунды. Общее время детекции, измеренное на центральном процессоре: 1,08 секунды.

Эксперимент 3: детекция на одном ядре центрального процессора. Использован почти тот же код, что и на сопроцессоре, все вычисления ведутся в памяти центрального процессора. Время построения пирамиды: 0,032 секунды. Общее время детекции лиц: 1,34 секунды.

Из представленных экспериментов можно сделать следующие выводы:

- Синхронизация и передача данных между общей памятью и сопроцессором занимает 2,3% времени в первом эксперименте, и 7,4% времени во втором эксперименте. Рост процентного соотношения накладных расходов закономерен, так как объем передаваемых данных тот же, а общее время работы снижается.
- Эффективность работы кода, запущенного на 16-ти ядрах, равна 95,3%. Это высокий показатель, означающий целесообразность использования всех 16-ти ядер сопроцессора для решения данной задачи.
- Из данных третьего эксперимента следует, что вычислительная сложность построения пирамиды составляет около 2,4% от сложности всего алгоритма, поэтому вынос этого кода на центральный процессор не влияет существенно на общее время работы.
- Из второго и третьего экспериментов не следует делать вывод о нецелесообразности использования сопроцессора, так как, во-первых, в нашем распоряжении был сопроцессор с заниженной тактовой частотой, и, во-вторых, процессор AMD E-350 предназначен для ноутбуков, а Epiphany — для мобильных телефонов. Для адекватного сравнения нужно учесть разницу тактовых частот, а также площадь кристалла и энергопотребление. Получаем, что ядро Epiphany с тактовой частотой 800 МГц примерно в 6 раз медленнее на данной задаче, чем ядро AMD E-350. Это хороший показатель, учитывая, что ядро Epiphany имеет площадь 0,3 мм^2 и энергопотребление 14 мВт, а ядро AMD E-350 имеет площадь 30 мм^2 и энергопотребление 8 Вт.

Заключение

Микропроцессор Epiphany имеет архитектуру, пригодную для решения широкого класса задач. Он оптимизирован для снижения энергопотребления, поэтому имеет перспективы применения в мобильных телефонах и планшетных компьютерах.

Наличие сквозной адресации позволяет реализовывать параллельные алгоритмы с различными схемами передачи данных, находясь в рамках модели программирования языка Си (производитель заявляет, что C++ уже тоже поддерживается).

Кроме указанных преимуществ, микропроцессор имеет недостатки, прежде всего связанные с малым объемом памяти на каждом его ядре, и значительным падением быстродействия при использовании в программе арифметики, не поддерживаемой аппаратно. Эти особенности делают невозможным эффективное использование кодов, разработанных для других процессоров; требуется тщательно продумывать структуры данных и их расположение в памяти, и соответственно переписывать код.

Несмотря на эти сложности, нам удалось разработать и реализовать эффективный алгоритм детекции лиц, совместимый (по данным классификатора) с алгоритмом детекции лиц из распространенной библиотеки OpenCV. Продемонстрирована целесообразность использования параллельной реализации (ускорение решения задачи) и ее высокая эффективность при работе на 16-ти ядрах микропроцессора Epiphany.

Работа выполнена при поддержке Российского научного фонда, грант 14-11-00659.

Литература

1. Papamarcos, M.S. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories / M.S. Papamarcos, J.H. Patel // Proceedings of the 11th annual international symposium on Computer architecture ISCA'84. — 1984. — P. 348–354.

2. Archibald, J. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model / J. Archibald, J. Baer // ACM Trans. on Computer Systems. — 1986. — Vol. 4, No. 4. — P. 273–298.

3. Baumann, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems / A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, A. Singhania // Proceedings of the 22nd ACM Symposium on OS Principles (Big Sky, MT, USA). — 2009. — P. 29–44.

4. Face Detection using the Epiphany Multicore Processor / URL: `http://www.adapteva.com/white-papers/face-detection-using-the-epiphany-multicore-processor/` (дата обращения: 13.02.2014).

5. Parallela – Supercomputing for Everyone / URL: `http://www.parallella.org/` (дата обращения: 13.02.2014).

6. OpenCV / URL: `http://opencv.org/` (дата обращения: 13.02.2014).

7. Abu-Mostafa, Y.S. Learning from Data / Y.S. Abu-Mostafa, M. Magdon-Ismail, H.-T. Lin. — AMLBook, 2012. — 213 p.

8. Ojala, T. Performance Evaluation of Texture Measures with Classification Based on Kullback Discrimination of Distributions / T. Ojala, M. Pietikäinen, D. Harwood // Proceedings of the

12th IAPR International Conference on Pattern Recognition (ICPR 1994). — 1994 — Vol. 1. — P. 582–585.

9. Viola, P. Rapid Object Detection Using a Boosted Cascade of Simple Features / P. Viola, M. Jones // Computer Vision and Pattern Recognition. — 2001. — Vol. 1. — P. 511–518.
10. Mitchell, D.P. Reconstruction Filters in Computer-Graphics / D.P. Mitchell, A.N. Netravali // ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques. — 1988. — Vol. 22, No. 4. — P. 221–228.
11. Crow, F.C. Summed-Area Tables for Texture Mapping / F.C. Crow // Proceedings of the 11th annual conference on Computer graphics and interactive techniques. — 1984. — P. 207–212.

Сухинов Антон Александрович, к.ф. м.н., научный сотрудник Сколковского института науки и технологий (Сколково, Российская Федерация), soukhinov@gmail.com.

Остроброд Георгий Борисович, программист ООО «СиВижинЛаб» (Таганрог, Российская Федерация), wdf.gost@gmail.com.

Поступила в редакцию 4 августа 2014 г.

Bulletin of the South Ural State University
Series "Computational Mathematics and Software Engineering"
2014, vol. 3, no. 3, pp. 5–19

EFFICIENT FACE DETECTION ON EPIPHANY MULTICORE PROCESSOR

A.A. Sukhinov, Skolkovo Institute of Science and Technology (Skolkovo, Russia)**,**
G.B. Ostrobrod, CVisionLab (Taganrog, Russia)

It is studied the possibility of usage of energy-efficient Epiphany microprocessor for solving actual applied problem of face detection at still image. The microprocessor is a multicore system with distributed memory, implemented in a single chip. Due to small die area the microprocessor has significant hardware limitations (in particular it has only 32 kilobytes of memory per core) which limit the range of usable algorithms and complicate their software implementation. Common face-detection algorithm based on local binary patterns (LBP) and cascading classifier was adapted for parallel implementation. It is shown that Epiphany microprocessor having 16 cores can outperform single-core CPU of personal computer having the same clock rate by a factor of 2.5, while consuming only 0.5 watts of electric power.

Keywords: face detection, local binary patterns, parallel data processing, specialized processors, distributed memory.

References

1. Papamarcos M.S., Patel J.H. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories // Proceedings of the 11th annual international symposium on Computer architecture ISCA'84. 1984. P. 348–354.
2. Archibald J., Baer J. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model // ACM Trans. on Computer Systems. 1986. Vol. 4, No. 4. P. 273–298.

3. Baumann A., Barham P., Dagand P.-E., Harris T., Isaacs R., Peter S., Roscoe T., Schüpbach A., Singhania A. The Multikernel: A New OS Architecture for Scalable Multicore Systems // Proceedings of the 22nd ACM Symposium on OS Principles (Big Sky, MT, USA). 2009. P. 29–44.

4. Face Detection using the Epiphany Multicore Processor. URL: `http://www.adapteva.com/white-papers/face-detection-using-the-epiphany-multicore-processor/` (accessed: 13.02.2014).

5. Parallela – Supercomputing for Everyone. URL: `http://www.parallella.org/` (accessed: 13.02.2014).

6. OpenCV. URL: `http://opencv.org/` (accessed: 13.02.2014).

7. Abu-Mostafa Y.S., Magdon-Ismail M., Lin H.-T. Learning from Data. AMLBook, 2012. 213 p.

8. Ojala T., Pietikäinen M., Harwood D. Performance Evaluation of Texture Measures with Classification Based on Kullback Discrimination of Distributions // Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994). 1994. Vol. 1. P. 582–585.

9. Viola P., Jones M. Rapid Object Detection Using a Boosted Cascade of Simple Features // Computer Vision and Pattern Recognition. 2001. Vol. 1. P. 511–518.

10. Mitchell D.P., Netravali A.N. Reconstruction Filters in Computer-Graphics // ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques. 1988. Vol. 22, No. 4. P. 221–228.

11. Crow F.C. Summed-Area Tables for Texture Mapping // Proceedings of the 11th annual conference on Computer graphics and interactive techniques. 1984. P. 207–212.

Received 4 August 2014.