ДВУМЕРНАЯ РЕКУРСИВНАЯ ФИЛЬТРАЦИЯ В GPU СИСТЕМАХ НА ОСНОВЕ МОДЕЛИ НЕЖЕСТКОГО СЛЕДОВАНИЯ

А.В. Никоноров, С.Б. Попов, В.А. Фурсов

1. Введение

В работе [1] рассматривалась информационная технология определения характеристик и обработки изображений с использованием двумерных фильтров с бесконечной импульсной характеристикой (БИХ-фильтров). В частности, для решения задачи идентификации параметров фильтра в указанной работе предложено использовать малые тестовые фрагменты, которые формируются из искаженного изображения с использованием априорной информации о геометрической форме регистрируемых объектов. При этом двумерный БИХ-фильтр, обеспечивающий компенсацию сильных искажений с использованием опорной области небольших размеров, строится в виде параллельного соединения физически реализуемых фильтров, которые используют соответствующий квадрант в опорной области.

В указанной выше работе для реализации технологии идентификации параметров и обработки крупноформатных изображений с использованием БИХ-фильтров предложена архитектура распределенной вычислительной системы, в которой параллелизм реализуется как на уровне данных, за счет декомпозиции крупноформатного изображения, так и на функциональном уровне, распараллеливая структуру самого БИХ-фильтра. Была показана эффективность предложенной технологии для обработки данных дистанционного зондирования Земли.

Вопросы повышения производительности обработки одномерными БИХ-фильтрами за счет использования параллельных вычислительных ресурсов достаточно хорошо исследованы. В работе [2] рассматривалась реализация одномерных БИХ-фильтров в многопоточных вычислительных средах. В работе [3] описывется GPU-реализация одномерного БИХ-фильтра. В двумерном случае возможна декомпозиция процедуры обработки, в результате которой задача сводится к последовательному применению одномерных фильтров [4]. К сожалению, в общем случае, такую декомпозицию провести нельзя. Задача параллельной реализации двумерного БИХ-фильтра похожа на задачу распараллеливания вложенных циклов. Однако, при ее реализации на GPU подход, предложенный в работе [7], оказался неприменим из-за специфики вычислительной модели GPU.

В настоящей работе предлагается универсальная массивно-многопоточная процедура двумерной рекурсивной обработки данных, с использованием GPU.

2. Формулировка проблем массивно-многопоточной реализации фильтров

В задачах обработки крупноформатных изображений (например, данных дистанционного зондирования Земли) широко используются линейные фильтры. Простейший линейный фильтр с конечной импульсной характеристикой (КИХ-фильтр) реализует свертку, т.е. за оценку выходного сигнала $\hat{y}(n_1, n_1)$ принимается взвешенная сумма конечного числа отсчетов наблюдаемого сигнала $x(n_1, n_1)$:

$$\hat{y}(n_1, n_1) = \sum \sum_{k_1, k_2 \in D} h(k_1, k_2) x(n_1 - k_1, n_2 - k_2) . \quad (1)$$

КИХ-фильтр всегда устойчив и физически реализуем. Однако в ситуации, когда динамические искажения значительны, опорная область D также имеет значительные размеры, что приводит к существенному возрастанию вычислительной сложности. Попытка преодолеть эту трудность путем уменьшения размеров опорной области приводит к ухудшению качества восстановления изображения.

В этом отношении более предпочтительным является фильтр с бесконечной импульсной характеристикой (БИХ-фильтр). В общем случае выражение, определяющее способ вычисления выходного отсчета $y(n_1, n_1)$ двумерного БИХ-фильтра, имеет вид [5]:

$$\hat{y}(n_1, n_1) = \sum_{r_1} \sum_{r_2} a(r_1, r_2) x(n_1 - r_1, n_2 - r_2) - \sum_{k_1, k_2 :} \sum_{k_1 k_2 \neq n_1 n_2} b(k_1, k_2) y(n_1 - k_1, n_2 - k_2) .$$

Нетрудно заметить, что в данном случае требование рекурсивной вычислимости не выполняется, поскольку для вычисления выходного отсчета используются отсчеты, которые еще не вычислены. Для физической реализуемости двумерного БИХ-фильтра он представляется в виде параллельного соединения четырех БИХ-фильтров, использующих различные развертки и соответствующие им квадранты опорной области [6].

Важным достоинством БИХ-фильтра является возможность восстановления изображений, имеющих значительные динамические искажения, с использованием опорных масок небольших размеров на входном и

выходном изображениях. Это свойство является особенно ценным при использовании технологии определения параметров фильтра путем решения задачи идентификации по малым тестовым фрагментам, описанной в работе [1].

Поквадрантная реализация БИХ-фильтра [1] позволяет естественным образом провести распределение вычислений на четыре потока. В настоящей работе предлагается массивно-многопоточный алгоритм реализации БИХ-фильтра. Такая реализация в первую очередь ориентирована на использование в GPU-системах, однако также может быть использована в современных многоядерных CPU-системах.

Особенностью архитектуры GPU является то, что вычисления выполняются блоками потоков, в рамках каждого блока потоки взаимодействуют между собой посредством разделяемой памяти. Связь между блоками реализуется только посредством глобальной памяти, средства взаимодействия и синхронизации крайне ограничены.

Такая архитектура может быть описана *двухуровневой массивно-многопоточной моделью* вычислений. Первый уровень модели составляют потоки (threads) в рамках одного блока (block). Второй уровень составляют блоки. Взаимодействие между потоками одного блока является максимально быстрым, поскольку осуществляется посредством разделяемой памяти, расположенной непосредственно на кристалле GPU, и имеет механизмы синхронизации. Взаимодействие между потоками разных блоков возможно лишь через глобальную память, что гораздо медленнее. Специальные примитивы синхронизация потоков разных блоков отсутствуют, синхронизация с данном случае может быть реализована через использование атомарных операций при работе с глобальной памятью.

<u>Ключевые особенности массивно-многопоточной архитектуры CUDA/OpenCL</u> следующие.

1. *Вычислительные единицы* – потоки, исполняются параллельно.
2. Потоки объединяются в блоки – *вычислительные узлы*, блоки также выполняются параллельно.
3. Потоки одного блока имеют доступ к быстрой *разделяемой* памяти, доступна функция барьерной синхронизации потоков __syncthreads(). Объем разделяемой памяти 16-48 КБ на блок.
4. Потоки любого блока имеют доступ к *медленной* глобальной памяти. Обращение к глобальной памяти соседних потоков одного блока может быть объединено в одну транзакцию, что ускоряет доступ до 8 раз.
5. Обмен данными различных потоков происходит за счет использования глобальной памяти.
6. Синхронизация доступа к глобальной памяти потоков из разных блоков возможна за счет использования атомарных операций с глобальной памятью, таких как atomicAdd(), atomicExch(), atomicCAS().
7. Запуском блоков и потоков на исполнение управляет системный планировщик, алгоритм планирования построен так, что одновременность и определенная очередность запуска потоков *не обеспечиваются*.
8. Планировщик реализует невытесняющее управление, т.е. не один запущенный на выполнение блок не будет остановлен планировщиком, пока все его потоки не завершат выполнение.

Указанные особенности приводят к тому, что полностью рекурсивная схема вычислений не может быть реализована на GPU[8]. Однако для задач локальной обработки изображений скользящим окном (например, с помощью КИХ-фильтров) эти ограничения можно обойти. Так в работе [9] была предложена процедура эффективной реализации рекуррентной обработки изображения локальным окном. С небольшими изменениями эта процедура может быть использована для решения задачи двумерной КИХ-фильтрации. Несколько иная реализация КИХ-фильтра на GPU была предложена в работе [11].

Процедура массивно-многопоточной реализации БИХ-фильтра на GPU кроме описанной поквадрантной модификации использует представление квадрантного фильтра в виде последовательного соединения двух фильтров-компонент. Первый КИХ-компонент зависит только от входного сигнала и соответствует числителю передаточной функции фильтра. Второй, зависит только от выходного сигнала и соответствует знаменателю. Будем называть его *рекурсивным компонентом* фильтра.

Реализация КИХ-компонента для массивно-многопоточной GPU-архитектуры может быть выполнена аналогично предложенному в [9] алгоритму рекуррентной локальной обработки изображения скользящим окном или согласно методу, изложенному в [11].

Далее рассматривается массивно-многопоточная реализация рекурсивного компонента. Будем рассматривать один квадрант, (x>0, y>0), для остальных трех квадрантов процедура выполняется аналогично. Передаточная функция рекурсивного компонента имеет вид:

$$\frac{1}{\sum_{i=0,j=0}^{m-1} b_{i,j} z_1^{-1} z_2^{-1}}, b_{0,0}=0 .$$

Тогда поэлементное преобразование изображения описывается соотношением:

$$y_{k,l}=\sum_{i=0,j=0}^{m-1,m-1} b_{i,j}\, y_1(k-i)\, y_2(l-j)\,, \tag{2}$$

где m – размер окна фильтра.

Преобразование всего изображения можно представить следующим псевдокодом

Листинг 1.

```
for(imageColumn = 0; imageColumn < N; imageColumn++ ){
      for(imageRow = 0; imageRow < N; imageRow++ ){
            image[imageRow, imageColumn] = doFilter(...);
      }
}
```

Процедура doFilter() реализует соотношение (3) для каждого отсчета выходного изображения. В двухуровневой архитектуре CUDA естественно распараллелить процедуру Filter() по потокам одного блока, а два цикла по строкам и столбцам обрабатываемого изображения распараллелить на уровне блоков.

Существует подход, получения параллельной реализации вложенных циклов достаточно общей структуры, основанный на теореме о гиперпланах Лесли Лэмпорта [7]. Однако предлагаемый подход эффективен при глубине вложенности циклов более 3, для нашего случая это не так (листинг 1). Получение четырех вложенных циклов возможно, если представить процедуру doFilter() в виде двух циклов, однако в таком случае теряется двухуровневая структура задачи.

Ввиду того, что применение классического подхода порождает указанные сложности, в настоящей работе предлагается альтернативный метод реализации фильтра в двухуровневой массивно-многопоточной среде. Предлагаемый подход основан на двух составляющих – «каскадной интерпретации» алгоритма рекурсивной фильтрации и вычислительной модели нежесткого следования.

3. Рекурсивные массивно-многопоточные вычисления с нежестким следованием

Идея каскадного рекурсивного алгоритма состоит в том, чтобы итерации внешнего цикла листинга 1 по столбцам изображения исполнялись блоками параллельно. При этом блок, перед тем как перейти к вычислению отсчета некоторой строки должен дождаться завершения вычисления отсчета в этой строке предыдущим блоком (т.е. в предыдущем столбце).

Алгоритм, выполняемый каждым блоком можно представить в виде упрощенного листинга 1, при этом *j*-тый элемент массива columns содержит номер строки последнего обработанного отсчета для *j*-того столбца:

Листинг 2.

```
curRow = 0;
while(curRow < rowCount){
      if(columns[blockIndex - 1] < curRow)
            continue;
      doFilter(image, curRow, blockIndex);
      columns[blockIndex] = curRow;
      curRow = curRow + 1;

}
```

Таким образом, блок с индексом j^b начинает обработку отсчета i^c-той строки j^c-того столбца, когда все отсчеты в своем столбце с индексами $i<i^c$ и все отсчеты (j^c-1)-вого столбца с индексами $i\leq i^c$ уже обработаны. Правило определения пары (i^c, j^c) текущих индексов строки и столбца обрабатываемого отсчета следующее:

$$i^c \leqslant i : (i, j^c-1)\in P, (i^c-1, j^c)\in P, (i^c, j^c)\notin P\,, \tag{3}$$

$$j^c = j^b\,. \tag{4}$$

Здесь *P* – множество пар номеров строк и столбцов (*i,j*) обработанных элементов.

В современной GPU системе согласно пункту 7 архитектурных особенностей GPU очередность запуска блоков непредсказуема, а также заведомо не все блоки выполняются одновременно. В таких условиях описанный каскадный рекурсивный алгоритм неизбежно будет приводить к блокировкам, при которых *j*-тый блок будет бесконечно долго ожидать (*j-1*)-вый блок. Если при этом (i-1) блок будет приостановлен планировщиком, то j-тый поток будет находиться в состоянии бесконечной блокировки, а соответственно, и

(j+1)-вый. Также все остальные блоки с номерами больше i также будут находиться в состоянии бесконечной блокировки.

Таким образом, для GPU систем любые схемы, построенные на ожидании блоком некоторого конкретного предыдущего блока, нереализуемы. В настоящей работе, чтобы преодолеть эту проблему предлагается *модель нежесткого следования*.

В этой модели номер столбца j, который обрабатывает блок, не фиксирован. Для того, чтобы определить, какой столбец изображения требует обработки, блок выбирает блок с минимальным индексом, для которого выполняется условие (3).

Если за блоком не закреплен номер обрабатываемого столбца, то при обработке будут возникать ситуации гонки за ресурс – когда два блока начнут обработку одного и того же столбца. Избежать такой ситуации возможно за счет дополнительного массива блокировок столбцов, расположенного в глобальной памяти. Блок при начале обработки захватывает блокировку столбца, при завершении обработки – освобождает.

Для модели нежесткого следования с блокировками соотношения (3)-(4) примут вид:

$$i^{c} \leqslant i : (i, j^{c}-1) \in P, (i^{c}-1, j^{c}) \in P, (i^{c}, j^{c}) \notin P, \quad (5)$$

$$j^{c} \} \; j^{b}, j \notin G \cdot \quad (6)$$

Здесь G – множество столбцов с захваченными блокировками. Условие (4), фиксирующее блок в исходной версии алгоритма, преобразуется в (6) – условие останова для алгоритма нежесткого следования.

Упрощенный алгоритм, реализующий каскадную рекурсивную обработку изображения с использованием нежесткого следования, приведен в листинге 3.

Листинг 3.

```
curRow = 0;
while(curRow < rowCount){
      curCol = blockIndex;
      while(columns[curCol - 1] < curRow
      && curCol > 0 && curRow > 0){
            curCol = curCol – 1;
            curRow = columns[curCol] + 1;
      }
      lockState = atomicCAS(&locks[curCol], 0, 1);
      if(lockState) continue;

      doFilter(image, curRow, curCol);
      columns[blockIndex] = curRow;
      atomicExch(&locks[curCol], 0);
      curRow = curRow + 1;
}
```

Массив columns реализует множество P номеров обработанных элементов, массив locks реализует множество блокировок G. Полученный рекурсивный алгоритм не зависит от порядка и количества параллельно исполняемых блоков. Для обеспечения когерентности при работе с массивом блокировок в глобальной памяти – locks, используются атомарные операции CUDA/OpenCL. Причем для выполнения операции проверки и захвата блокировки столбца в одной транзакции целесообразно использовать функцию atomicCAS [8,10], а для освобождения блокировки – atomicExch [8,10]. Листинг 3 отражает лишь основную идею алгоритма, без таких частных моментов как, например, обработка начальных и граничных условий.

Утверждение: Каскадно-рекурсивный алгоритм с нежестким следованием для вычислительной модели CUDA, соответствующей пунктам 1-8 архитектурных особенностей GPU, обладает следующими свойствами:

a) ни один блок алгоритма не попадает в состояние бесконечной блокировки,
b) после завершения работы алгоритма все элементы изображения обработаны.

Докажем первую часть утверждения. Действительно, рассмотрим произвольный столбец, он может быть в двух состояниях – заблокированным и готовым к обработке. В заблокированном состоянии столбец не может находиться бесконечно, согласно пункту 8 архитектурных особенностей GPU систем. Таким образом, любой блок либо найдет блок доступный для обработки, либо все данные будут обработаны. Т.е. любой блок сможет завершить обработку за конечное время.

Вторую часть докажем от противного. Предположим, что элемент i,j не обработан, но тогда блок j+1 не сможет обработать элемент (i+1,j+1), и соответственно, не сможет завершить обработку, это противоречит первой части утверждения.

Потоки каждого блока выполняют суммирование (2) параллельно, по схеме каскадного суммирования

или редукции[8, 10]. При таком суммировании используется $2^{2]\log_2 m[}$ потоков, где m – размер окна фильтра, а значит количество операций выполняемых потоками параллельно составит

$$O_1 = 2]\log_2 m[\ ,$$

где под]m[понимается наименьшее целое, большее m.

Если один блок обрабатывает один столбец изображения, то количество вычислений для каждого блока составит:

$$O_2 = 2N]\log_2 m[\ .$$

Если N потоков обрабатывают N столбцов изображения, то согласно каскадно-рекурсивной схеме, последний блока должен ожидать выполнения N-1 блока. Таким образом, для последнего блока, а, следовательно, и для всего алгоритма количество операций можно оценить как :

$$O = 2(2N-1)]\log_2 m[\ . \qquad (7)$$

Количество операций, требуемое для последовательной реализации алгоритма, составляет $N^2 m^2$. Таким образом, теоретическая оценка ускорения параллельной реализации позволяет предположить значительное ускорение.

Сложная массивно-многопоточная модель современных GPU и сложность алгоритма с нежестким следованием затрудняют получение оценки сложности для этого алгоритма. Однако структура алгоритма позволяет предположить, что сложность будет так же, как и для (7) линейно зависеть от *N*. Эта гипотеза подтверждается экспериментально, что показано в следующем разделе.

4. Результаты экспериментов

Экспериментальные исследования проводились для задачи повышения качества изображения на основе двумерного БИХ фильтра с окном размером m*m на квадратном изображении размера N*N.

На рисунке 1 приведена зависимость времени вычислений от размера изображения N. Результаты получены на бытовом GPU GF 465. Характер зависимости близок к линейному, что близко к теоретической оценки сложности (7) для каскадно-рекурсивного алгоритма. Отклонения от линейного характера показывают, что затраты на организацию нежесткого следования возрастают с ростом N.

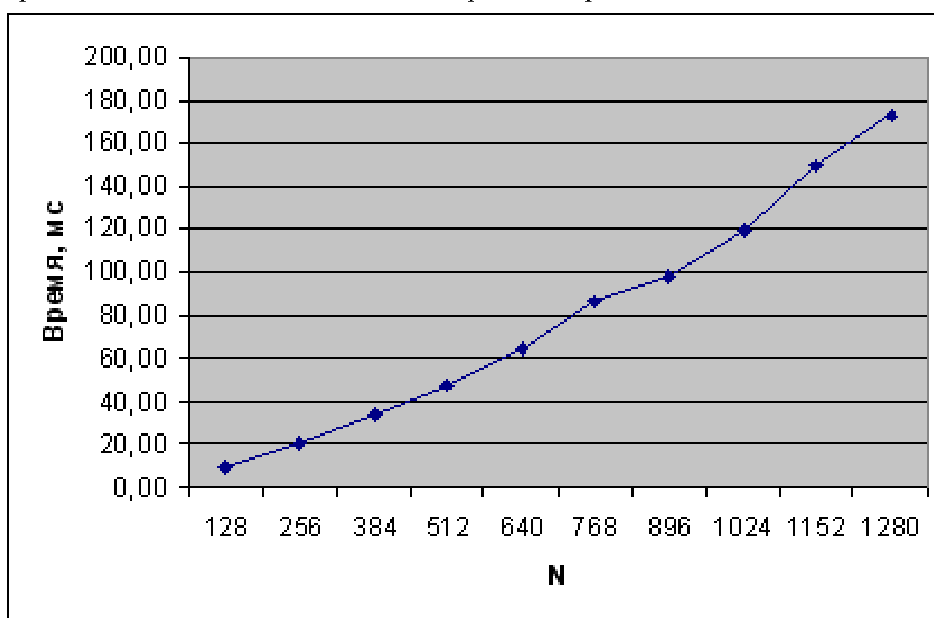

Рис. 1 Зависимость времени вычислений от размера изображения.

Оценить чередование блоков в рекурсивном алгоритме с нечетким следованием можно из таблицы 1. В таблице приведены номера блоков для достаточно небольшого фрагмента изображения. По горизонтали приведены номера столбцов, по вертикали – номера строк, в ячейках приводятся номера блоков, выполнявших обработку.

Таблица 1. Пример порядка обработки элементов изображения различными блоками

	4	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	5	5	6	7	9	9	10	11
1	5	6	6	7	8	9	10	11
2	4	5	7	7	8	9	10	11

3	4	5	8	8	8	9	11	11
4	4	5	7	8	8	9	10	11
5	4	5	7	8	8	9	10	12
6	4	5	6	7	8	10	11	11
7	5	5	6	7	8	9	10	11

Из таблицы видно, что хотя ни один столбец не был обработан только одним блоком, в среднем, все блоки обработали равное количество элементов, и не один элемент не остался не обработанным. Это согласуется с утверждением из предыдущего раздела. Отметим, что равномерное чередование блоков при обработке в существенной степени зависит от величины ограничения в условии (6). Наиболее эффективное чередование получается при условии сто это ограничение равно номеру блока.

5. Заключение

Рассмотренная технология обработки изображений с использованием двумерных БИХ-фильтров является развитием технологии описанной в работе [1] для GPU систем. В работе рассмотрена модель GPU системы как двухуровневой массивно-многопоточной системы с рядом специфичных ограничений, затрудняющих реализацию рекурсивных процедур.

Наиболее важным результатом работы является каскадно-рекурсивный алгоритм с нежестким следованием для GPU-систем. Такой подход позволяет преодолеть традиционно существовавшее мнение, что эффективная реализация рекурсивных процедур на GPU-процессорах невозможна. Проведено экспериментальное исследование предложенного алгоритма в задаче двумерной БИХ фильтрации, получены теоретические и экспериментальные оценки эффективности. Интересным для дальнейших исследований является анализ эффективности предложенного алгоритма для реализации БИХ-фильтров в RDMA системах.

Работа выполнена при поддержке программы фундаментальных исследований Президиума РАН «Проблемы создания национальной научной распределенной информационно-вычислительной среды на основе развития GRID технологий и современных телекоммуникационных сетей», гранта Президента РФ для ведущих научных школ № НШ-7414.2010.9 и грантов Российского фонда фундаментальных исследований № 10-07-00553, 09-07-00269.

ЛИТЕРАТУРА:

1. М.Г. Милюткин, А.В. Никоноров, В.А. Фурсов "Параллельная реализация двумерных БИХ-фильтров в распределенной системе обработки изображений" // Вычислительные методы и программирование, том 11, номер 1, 2010 г., Изд-во Московского университета С. 92-98
2. W. Sung and S.K. Mitra "Effiicient Multi-Proccessor Implementtation of Recursive Digital Filters", // ICASSP, 1986
3. M.D. Mccool "Signal Processing and General-Purpose Computing and GPUs" // Signal Processing Magazine, IEEE, 2008, pp. 109-114
4. С.А. Мурызин, В.В. Сергеев, Л.Г. Фролова "Исследование эффективности двумерных параллельно-рекурсивных КИХ-фильтров" // Компьютерная оптика. - М.: МЦНТИ, 1992. - Вып.12. - С.65-71
5. "Методы компьютерной обработки изображений" / Под ред. Сойфера В.А., Москва, Физматлит, 2001
6. Д.И. Зимин, В.А. Фурсов "Построение устойчивых алгоритмов обработки изображений путем аппроксимации фильтров с бесконечной импульсной характеристикой" // Компьютерная оптика, № 28, 2005, с. 124-127
7. L. Lamport "The parallel execution of DO loops" // ACM Communication, vol. 17 issue 2, 1974, pp. 83-93
8. А. В. Боресков, А. А. Харламов "Основы работы с технологией CUDA" // ДМК Пресс, 2010, 230 с
9. S. Bibikov, V. Fursov, A. Nikonorov, P. Yakimov "Memory Optimization for Recurrent CUDA Image Processing", proc. of PRIA 2010 Proceedings, 2010, pp. 176-179
10. "NVIDIA CUDA C Best Practices Guide" / Santa Clara, 2010, 75p
11. A. Smirnov, T.-C. Chiueh "An Implementation of a FIR Filter on a GPU" // Technical Report, Stony Brook University, 2005, 8p