УДК 004.424.5.032.24

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ПОИСКА ДАННЫХ И ИХ РЕАЛИЗАЦИЯ НА ПЛАТФОРМЕ CUDA

С. В. Скворцов, д.т.н., профессор кафедры САПР ВС РГРТУ; s.v.skvor@gmail.com
Т. А. Фетисова, аспирант РГРТУ; pyurova_ta@mail.ru
Д. В. Фетисов, аспирант РГРТУ; morzitko@gmail.com

Рассматриваются теоретические и практические вопросы реализации алгоритмов параллельного поиска данных на платформе CUDA. Актуальность данной темы обусловлена постоянным возрастанием объемов обрабатываемой информации, что приводит к необходимости повышения скорости обработки данных.

Целью работы *является модификация известных алгоритмов внутреннего поиска данных для их использования на графическом процессоре с последующим анализом эффективности параллельных версий разработанных программных модулей. Полученные результаты показывают, что использование графического процессора позволяет значительно увеличить скорость работы прикладных программ, использующих алгоритмы поиска данных, причем величина ускорения существенно зависит от размера блока потоков, используемого графическим процессором в блочном режиме распараллеливания.*

Ключевые слова: *мультипроцессор, графический процессор, графическая память, ядро, блок, поток, центральный процессор, поиск, ускорение.*

DOI: 10.21667/1995-4565-2018-65-3-55-62

Введение

В настоящее время вычислительные технологии развиваются быстрыми темпами, научно-технический прогресс сопровождается постоянной сменой одного поколения вычислительной техники другим. С увеличением возможностей компьютеров общество находит им применения в научной и прикладной сфере деятельности. Особенно повышаются требования к аппаратным ресурсам в связи с появлением таких задач, как рендеринг и обработка потокового визуального контента высокого качества, обработка экспериментальных данных в реальном времени, сортировка и поиск в больших массивах данных и многое другое. Одним из возможных подходов к решению таких задач стала технология использования графического процессора (ГП) видео-платы – GPGPU, задействующая ее ресурсы для выполнения общих неграфических вычислений. Среди производителей основными конкурентами являются фирма NVIDIA с комплексом CUDA [1] и фирма AMD с ATI Stream Technology [2]. Поэтому программирование, использующее ресурсы видеоплаты, невероятно актуально, поскольку можно получить большую производительность за минимальную цену.

Целью данной работы являются модификация и исследование алгоритмов поиска данных для их эффективной реализации на платформе CUDA в виде многопоточных приложений, сравнение их по производительности с последовательными версиями программ, исполняемых только на центральном процессоре (ЦП), а также систематизация полученных результатов.

Проблема организации эффективных вычислений средствами графических процессоров представляется новой по сравнению как с последовательным (линейным) программированием, так и с многопоточным программированием для многоядерных процессоров [3, 4]. Каждый аспект данной проблемы требует внимательного изучения с последующей реализацией и оптимизацией программ [5, 6].

Методика разработки неграфических многопоточных программ на платформе CUDA

Возможными вариантами применения технологии CUDA являются как создание новых параллельных приложений для решения прикладных задач, так и модификация существующего последовательного программного обеспечения. Модель программирования, используемая в CUDA, отличается от традиционных API тем, что полностью скрывает графический конвейер от программиста, позволяя ему писать программы в более привычных для него «терминах» на

расширенной версии языка Си. Кроме того, CUDA предоставляет программисту более удобную модель работы с памятью. В частности, нет необходимости хранить данные в 128-битных текстурах, так как CUDA позволяет читать данные напрямую из памяти видеоплаты.

В состав NVIDIA CUDA входят два API: высокого уровня (CUDA Runtime API) и низкого (CUDA Driver API). При необходимости задействовать низкоуровневые функции графического процессора программист всегда может отказаться от Runtime API в пользу Driver API. Стоит заметить, что использование обоих API в одной программе не является возможным.

Первым шагом при переносе существующего приложения на CUDA является определение участков кода, снижающих скорость работы приложения в целом. Если среди таких участков есть подходящие для быстрого параллельного исполнения, то выполнение этих функций переносится на графический ускоритель с помощью расширений CUDA для языка Си. Для решения такой задачи могут использоваться методы и алгоритмы анализа зависимостей по данным и управлению, позволяющие выявить скрытый параллелизм в программах [7-9].

При использовании технологии CUDA прикладная программа формируется с помощью поставляемого NVIDIA компилятора, который генерирует код и для ЦП, и для графического ускорителя. При работе программы центральный процессор выполняет свои процедуры, а графический процессор выполняет CUDA код с параллельными вычислениями. Эта часть программы, предназначенная для графического процессора, по терминологии CUDA называется ядром (kernel). В ядре определяются операции, которые будут исполнены над данными в разных параллельных потоках, причем они могут объединять в блоки потоков, которые в свою очередь образуют сетки блоков.

Разработка программного модуля / приложения требует точного разделения понятий программного ядра – подпрограммы (kernel) и вычислительного ядра – части мультипроцессора. При создании приложения программист имеет возможность задавать количество блоков и потоков. Оптимальным является использование от 64 до 512 потоков в блоке. Группировка блоков в сетки позволяет применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у графического процессора недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеоплатах разного уровня.

Программа для графического ускорителя, написанная на платформе CUDA, разделяется на две части: код для исполнения на графическом устройстве (device code) и код для исполнения на центральном процессоре (host code). Код для исполнения на графическом устройстве (kernel) может быть написан как на специальном низкоуровневом языке (PTX), так и на расширении языка Си (Си для CUDA). В последнем случае ядро имеет вид функции языка Си, описывающей поведение одного потока.

Платформа CUDA предлагает два разных способа вызова ядер. Более простой способ (Си для CUDA) является расширением языка Си. Вызов ядра при этом похож на вызов функции Си, но с передачей дополнительных параметров. Эти параметры определяют число потоков в блоке (размер блока) и количество одновременно исполняемых блоков в сетке (grid). Каждый поток работает со своими данными, которые определяются номером потока в блоке и номером блока в сетке.

Более сложным способом вызова ядер является Driver API. В этом случае ядра необходимо предварительно скомпилировать в бинарный формат, после чего используются функции для загрузки и выполнения ядер. Применение Driver API может привести к появлению ошибок, так как требует ручного выполнения многих действий, которые автоматизированы в Си для CUDA. Пример таких действий – передача параметров ядра. В целом Driver API дает разработчику больший контроль над выполнением программы, но требует более высокой квалификации программиста.

В данной работе рассмотрены алгоритмы бинарного и интерполяционного поиска, предложены их параллельные модификации, ориентированные на применение технологии CUDA, использующей распараллеливание данных, а также реализованы и исследованы многопоточные приложения в сравнении с последовательными версиями соответствующих программных модулей.

Бинарный поиск

Бинарный поиск – это нахождение заданного элемента (ключа) на упорядоченном множестве, которое осуществляется путем многократного деления этого множества на две примерно равные части таким образом, что искомый элемент оказывается в одной из этих частей [10]. Процедура поиска завершается при совпадении нужного элемента с элементом, являющимся

Таблица 1 – Бинарный поиск (последовательный)

Шаг	Элементы массива																	
	1	3	5	6	10	13	14	17	28	35	47	52	55	63	*69*	72	75	88
1	1	3	5	6	10	13	14	17	**28**	35	47	52	55	63	*69*	72	75	88
2										35	47	52	55	**63**	*69*	72	75	88
3															*69*	**72**	75	88
4															***69***			

Таблица 2 – Бинарный поиск (параллельный)

Этап (число потоков)	Элементы массива																	
	1	3	5	6	10	13	14	17	28	35	47	52	55	63	*69*	72	75	88
Этап 1 (4 блока)	1	3	**5**	6	10													
						13	14	**17**	28	35								
											47	52	**55**	63	*69*			
																72	**75**	88
Этап 2 (2 из 4 возможных блоков)														**63**				
															69			

границей между подмножествами, или при его отсутствии. В упорядоченном массиве может встречаться несколько элементов, значения которых равны ключу. Тогда результатом работы алгоритма поиска будет первый совпавший с ключом элемент.

Схема реализации алгоритма бинарного поиска представлена в таблице 1, где в качестве ключа выбирается элемент со значением 69, отмеченный курсивом, а сравниваемый элемент выделен полужирным шрифтом.

Идея распараллеливания алгоритма бинарного поиска заключается в том, чтобы изначально запустить на параллельное выполнение максимальное количество блоков, которое позволяет архитектура видеоплаты для осуществления поиска. Технологически исходный массив разбивается на блоки.

В каждом блоке размещается часть массива. Поиск в блоке осуществляется независимо от других блоков в отдельном потоке средствами графического ускорителя. Таблица 2 наглядно иллюстрирует данный процесс, причем после каждого разделения некоторой части массива степень параллелизма (число параллельных потоков) уменьшается.

Функция ядра данного алгоритма Kernel имеет следующий вид:

```
__global__ void Kernel(float *data, int n, int k, int *result, int x)
{
   int low = 0; // нижняя граница
   int up = n-1; // верхняя граница
   int step; // шаг
   int idx; // текущий поток
   while (up - low > k ) // пока количество элементов между границами больше количества потоков (в данном случае потоков k)
   {
      step = (up - low) / k;

      idx = low + (blockIdx.x * blockDim.x + threadIdx.x) * step;
      if ( data[idx] == x ) // найден искомый элемент
        {
         result[0] = data [idx];
         result[1] = idx; break;
         } else if ( data[idx] > x )
           {
             if(data[idx] < up)
   up = data[idx]; //минимальный среди больших
         } else if(data[idx] < x)
         {
           if(data[idx] > low)
             low =  data[idx]; // максимальный среди меньших
         }}}
```

Предложенный подход к организации многопоточного приложения позволяет заметно ускорить бинарный поиск по сравнению с последовательной программной реализацией, так как здесь уменьшение зоны поиска осуществляется пропорционально количеству параллельных потоков. В каждом потоке зона поиска сокращается примерно вдвое на каждой итерации. Исходное число параллельных потоков зависит от размера массива и характеристик видеоплаты:

$$block = \max\{size, amount\}, \quad (1)$$

где *block* – число параллельных потоков, *size* – размер массива или подмассива, *amount* – максимальное количество блоков видеокарты.

Интерполяционный поиск

Интерполяционный поиск непосредственно связан с понятием «интерполяция», что предполагает процесс нахождения неизвестных значений на основе имеющихся. Если при бинарном поиске массив разбивается на две примерно равные части, то интерполяционный поиск работает по-другому – для определения позиции искомого элемента используются значения элементов массива с известными индексами [10]. На каждой итерации происходит разбиение массива таким образом, чтобы найти ближайший элемент к ключу, причем точка разбиения определяется по следующей формуле:

$$Pos = left + (key - mas[left]) * \\ * (right - left) / (mas[right] - mas[left]), \quad (2)$$

где *Pos* – позиция элемента, сравниваемого с ключом; *left* – индекс левой границы поиска; *right* – индекс правой границы поиска; *mas[ ]* – исходный массив поиска.

Схема последовательной реализации алгоритма интерполяционного поиска представлена в таблице 3, где курсивом отмечен ключ поиска, а сравниваемый элемент выделен полужирным шрифтом.

Основная идея распараллеливания заключается в выделении максимально возможного количества блоков, содержащих элементы массива. К каждому блоку (подмассиву) применяется интерполяционный алгоритм поиска, который запускается на исполнение в отдельном потоке. Если ключ поиска совпадает со сравниваемым элементом при первом запуске подпрограммы Kernel, то поиск завершается. Если ключ поиска не совпадает со сравниваемым элементом, то происходит повторный запуск подпрограммы.

В подпрограмму передается только та часть массива, в которой содержится искомый элемент. Эта часть массива также делится на блоки, число которых определяется согласно формуле (1). Данная процедура повторяется до тех пор, пока не будет найден ключевой элемент.

Идея параллельной многопоточной реализации алгоритма интерполяционного поиска иллюстрируется в таблице 4, где предполагается использование нисходящего распараллеливания. Каждая строка соответствует подмассиву элементов, в котором поиск осуществляется в отдельном потоке независимо от других потоков.

Ядро данного алгоритма поиска представлено ниже.

```
    __global__ void Kernel ( float * data, int
n, int find, int rezult )
    {
       int idx; // текущий поток
       int low =0; // нижняя граница
       int up = n; // верхняя граница
      // пока искомый элемент не найден или пре-
делы поиска еще существуют
     while ((data[low] < find) && (data[up] >
find))
       {
    //вычисление интерполяцией следующего эле-
мента, который будет сравниваться с искомым
        idx = blockIdx.x + ((find - da-
ta[blockIdx.x]) * (blockIdx.x * blockDim.x +
threadIdx.x - blockIdx.x)) / (data[blockIdx.x *
blockDim.x + threadIdx.x] - data[blockIdx.x]);
        //Получение новых границ облас-ти, если
искомый элемент не найден
        if (data[idx] < find) low = idx + 1;
        else if (data[idx] > find) up = idx - 1;
        else result=idx;
        //Если искомый элемент найден на грани-
цах области поиска
        if (data[low] == find) rezult=low;
        else if (data[up] == find) rezult=up;
        else result=-1;
       }}
```

Скорость работы параллельного алгоритма интерполяционного поиска зависит от числа блоков на каждой итерации. По сравнению с бинарным алгоритмом, где зона поиска сокращается вдвое на каждой итерации в каждом потоке, здесь область поиска сокращается неравномерно, что позволяет получить дополнительное ускорение.

Таблица 3 – Интерполяционный поиск (последовательный)

Шаг	Элементы массива																	
	1	3	5	6	10	13	14	17	28	35	47	52	55	63	*69*	72	75	88
1	1	3	5	6	10	13	14	17	28	35	47	52	55	**63**	*69*	72	75	88
2															***69***	72	75	88

Таблица 4 – Интерполяционный поиск (параллельный)

Этап (число потоков)	Элементы массива																	
	1	3	5	6	10	13	14	17	28	35	47	52	55	63	*69*	72	75	88
Этап 1 (4 блока)	1	3	5	6	10													
						13	14	17	28	35								
											47	52	55	63	***69***			
																72	75	88

Аналитическая оценка эффективности

Чтобы проанализировать производительность и ускорение разработанных алгоритмов поиска данных, необходимо сначала оценить время выполнения последовательных (линейных) версий алгоритмов. Для бинарного поиска полное время работы алгоритма оценивается как

$$T_{sb} = 1 + \log_2(n),$$

где n – количество элементов исходного массива.

Интерполяционный поиск выполняется гораздо быстрее за счет того, что массив делится не на равные части, а с учетом ключей элементов согласно формуле (2). В соответствии с этим время выполнения интерполяционного поиска будет пропорционально величине

$$T_{si} = \log_2(\log_2(n)) .$$

Для анализа производительности и ускорения параллельных алгоритмов поиска необходимо учитывать следующие факторы [11]:

1) затраты на формирование исходных данных;
2) затраты на упорядочивание блоков данных;
3) затраты на передачу данных с центрального процессора на графический процессор и обратно;
4) вычислительные затраты графического процессора.

Время формирования исходных данных при параллельной реализации равно времени на формирование исходных данных в последовательной версии:

$$T_{p1} = f * t_s,$$

где f – часть работ, выполняемых одинаково в последовательной и параллельной версиях алгоритма поиска данных; t_s – время выполнения линейной версии алгоритма бинарного или интерполяционного поиска.

Временные затраты подготовительного этапа вычислений (второй и третий пункты) можно оценить следующей формулой:

$$T_{p2} = \frac{n}{2p} * \log_2(\frac{n}{2p}),$$

где p – количество блоков.

Длительность вычислений на графическом процессоре зависит от большого количества факторов, среди которых можно выделить латентность, пропускную способность и др. Для упрощения оценки затрат будем учитывать время работы подпрограммы на графическом ускорителе T_{p3} в целом. Тогда для бинарного поиска получим

$$T_{pb3} = \frac{n}{p} * \log_2(p * (k - 1)),$$

а для интерполяционного поиска:

$$T_{pi3} = \frac{n}{p} * \log_2(p * Pos),$$

где k – количество элементов в блоке для бинарного поиска; Pos – позиция элемента, определяемая по формуле (2).

В результате получаем оценки полного времени работы алгоритма параллельного поиска данных. Для бинарного поиска имеем

$$T_p = T_{p1} + T_{p2} + T_{pb3} = f * t_s + \frac{n}{2p} \times$$
$$\times \log_2(\frac{n}{2p}) + \frac{n}{p} * \log_2(p * (k - 1)),$$

для интерполяционного поиска

$$T_p = T_{p1} + T_{p2} + T_{pi3} = f * t_s + \frac{n}{2p} \times$$
$$\times \log_2(\frac{n}{2p}) + \frac{n}{p} * \log_2(p * Pos).$$

Тогда ускорение работы алгоритмов можно определить как

$$\alpha_b = \frac{T_{sb}}{T_p}$$

для бинарного поиска и

$$\alpha_i = \frac{T_{si}}{T_p}$$

для интерполяционного поиска.

Экспериментальная оценка программных приложений

Для оценки эффекта, полученного от применения параллельных алгоритмов поиска данных, реализованных на платформе CUDA, были проведены экспериментальные исследования разработанных программных приложений, представленных в двух версиях: последовательная реализация на ЦП; многопоточная реализация на платформе CUDA. Эксперимент проводился с использованием следующих технических средств: видеоплата NVIDIA GeForcegtx 470, процессор IntelCore2 QuadQ6600.

Цели эксперимента:

– сравнение скорости работы параллельных и последовательных версий программ, определение полученных ускорений;

– сравнение аналитических оценок ускорений с экспериментальными результатами;

– анализ полученных результатов в целом для определения рациональных режимов практического применения разработанных многопоточных приложений поиска данных.

В процессе эксперимента выполнялись измерения времени работы указанных версий программ, реализующих бинарный и интерполяционный поиск. Исходные данные, представлен-

ные в виде одномерных массивов целых чисел, формировались случайным образом и упорядочивались по возрастанию. В процессе эксперимента изменялись значения *n* – размер массива данных и *p* – число блоков.

Для каждого сочетания параметров n и p выполнено по 10 экспериментов с разными данными. Величина *n* изменялась в пределах от 10^5 до 10^6 с шагом 10^5, величина *p* принимала значения 256, 512 и 1024, где последняя величина является максимально возможной для указанной видеоплаты.

Графики зависимостей усредненных значений времени (в секундах) работы программ бинарного поиска и интерполяционного поиска приведены на рисунках 1 и 2 соответственно.

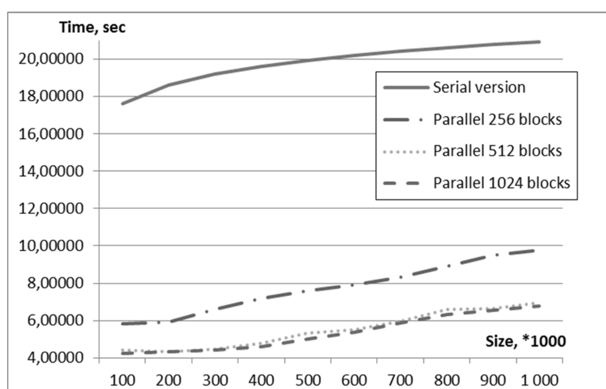

Рисунок 1 – Зависимости среднего времени работы программ бинарного поиска от размера массива

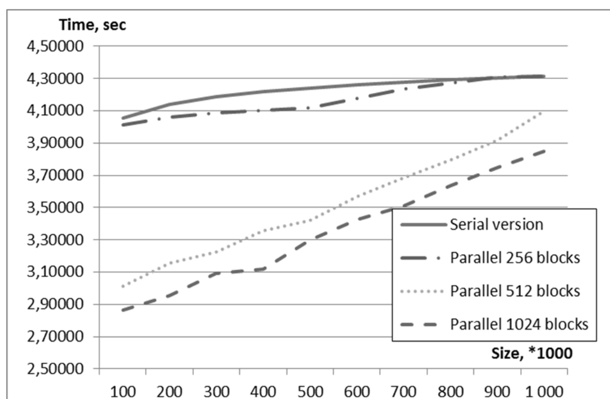

Рисунок 2 – Зависимости среднего времени работы программ, реализующих алгоритм интерполяционного поиска, от размера массива

Анализ времени работы программ бинарного и интерполяционного поиска показывает, что по сравнению с последовательными версиями выигрыш появляется сразу и данный показатель улучшается при увеличении количества блоков потоков на платформе CUDA. Это связано с тем, что изначально данные в ядре распределяются между большим количеством блоков, что позволяет уменьшить число итераций поиска.

Однако на небольшом количестве элементов в массиве параллельную реализацию алгоритма интерполяционного поиска использовать нецелесообразно. Это объясняется значительными затратами времени на формирование блоков данных и копирование данных из памяти ЦП в память ГП и обратно, что приводит к заметному росту общего времени выполнения программы.

На рисунках 3 и 4 представлены графики, позволяющие сравнить результаты ускорения алгоритмов поиска, отражающие аналитические и экспериментальные результаты исследования.

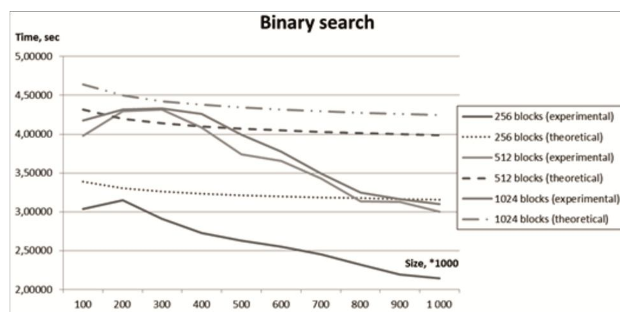

Рисунок 3 – Зависимости ускорения работы программ бинарного поиска от размера массива

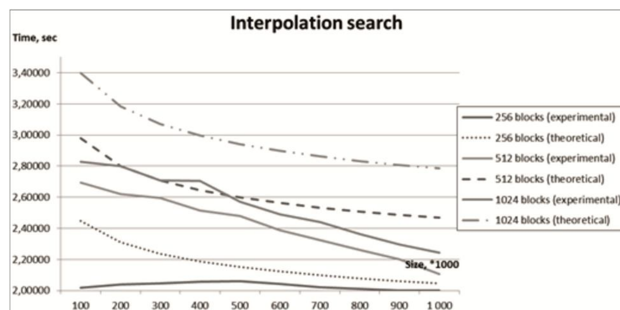

Рисунок 4 – Зависимости ускорения работы программ, реализующих алгоритм интерполяционного поиска, от размера массива

Анализ результатов исследования показывает, что аналитически рассчитанные ускорения превосходят ускорения, полученные экспериментальным путем. В частности, для алгоритма бинарного поиска теоретическое усредненное значение ускорения многопоточных приложений составляет 3,89 раза, а экспериментальное – 3,35 раза. Для алгоритма интерполяционного поиска аналогичное теоретическое значение ускорения равно 2,59 раза, а экспериментальное – 2,33 раза.

Полученные графики показывают, что аналитические оценки в полной мере отражают тенденции поведения экспериментальных зависимостей, а расхождения аналитических и экспериментальных данных можно объяснить, например, зависимостью последних от технических характеристик аппаратных средств вычислительной системы.

Присутствующие на графиках выпуклые области характеризуют упорядоченность исходного массива, т.е. при случайной генерации массива данных элементы в нем упорядочены неравномерно, что позволяет повысить скорость выполнения алгоритма.

Проанализируем далее зависимости ускорения работы алгоритмов поиска от количества n элементов массива и числа p параллельных блоков. Для постоянного значения p увеличение значения n приводит к постепенному уменьшению ускорения. Это связано с тем, что при увеличении размера массива требуется больше времени на копирование данных из памяти ЦП в память ГП и распределение их по выделенным блокам, а также увеличивается число итераций выполнения алгоритма поиска. Однако при увеличении числа p для постоянного значения n ускорение значительно возрастает за счет того, что уменьшается вычислительная нагрузка блоков и требуется меньшее число итераций для той части алгоритма, которая выполняется параллельно на графическом ускорителе.

Если сравнивать рассмотренные алгоритмы поиска в целом, то можно сделать вывод о том, что бинарный и интерполяционный поиски поддаются распараллеливанию приблизительно одинаково. Относительно способа распараллеливания и в бинарном, и в интерполяционном поиске используется нисходящий метод, т.е. с каждым шагом поиска количество параллельных потоков уменьшается.

Заключение

Полученные результаты показывают, что использование графического ускорителя позволяет значительно увеличить скорость работы прикладных программ, использующих алгоритмы поиска данных. Отсюда можно сделать важный для практики вывод: с ростом объема массивов и размера используемых блоков время работы многопоточного приложения на графическом ускорителе значительно сокращается по сравнению с последовательной реализацией на ЦП, причем с увеличением количества блоков выигрыш возрастает.

Однако данные алгоритмы имеют потенциальные возможности для доработки, предоставляемые технологией CUDA, одна из которых заключается в выборе метода распараллеливания потоками, а не блоками. Продолжение исследований целесообразно расширить на другие алгоритмы с целью разработки их параллельных версий и получения рекомендаций по применению в конкретных практических условиях.

Библиографический список

1. **Nvidia CUDA –** неграфические вычисления на графических процессорах [Электронный ресурс]. URL: http://www.nvidia.ru (дата обращения 11.01.2018).

2. **AMD ATI** Developer Central [Электронный ресурс]. URL: http://www.amd.com (дата обращения 15.01.2018).

3. **Козлов М. А., Скворцов С. В.** Алгоритмы параллельной сортировки данных и их реализация на языке Clojure // Вестник Рязанского государственного радиотехнического университета. 2013. № 4-1 (46). С. 92-96.

4. **Скворцов С. В.** Алгоритм планирования параллельных вычислений в многоядерных процессорах // Радиотехника. 2016. № 8. С. 153-159.

5. **Скворцов С. В., Пюрова Т. А.** Параллельные алгоритмы сортировки данных и их реализация на платформе CUDA // Вестник Рязанского государственного радиотехнического университета. 2016. № 58. С. 42-48.

6. **Корячко В. П., Скворцов С. В., Шибанов А. П., Перепелкин Д. А**. Методы и технологии автоматизации проектирования высокопроизводительных систем и компьютерных сетей // Вестник Рязанского государственного радиотехнического университета. 2017. № 60. С. 94-104.

7. **Корячко В. П., Скворцов С. В., Телков И. А.** Модель планирования параллельных процессов в суперскалярных процессорах // Информационные технологии. 1997. № 1. С. 8-12.

8. **Скворцов С. В.** Целочисленные модели оптимизации кода по критерию времени // Информационные технологии. 1997. № 10. С. 2-7.

9. **Рудаков В. Е., Скворцов С. В.** Построение базового множества независимых путей потокового графа для тестирования программных модулей // Системы управления и информационные технологии. 2012. Т. 50. № 4. С. 67-70.

10. **Kvodo** – Computing Science & Discrete Match (бинарный и интерполяционный поиск) [Электронный ресурс]. URL: http://kvodo.ru (дата обращения 16.10.2017).

11. **Гергель В. П.** Высокопроизводительные вычисления для многопроцессорных многоядерных систем. М.: Издательство Московского университета, 2010. С. 544.

UDC 004.424.5.032.24

PARALLEL DATA SEARCH ALGORITHMS AND THEIR IMPLEMENTATION ON THE CUDA PLATFORM

S. V. Skvortsov, PhD (technical sciences), full professor, RSREU, Ryazan; s.v.skvor@gmail.com
T. A. Fetisova, post-graduate student, RSREU, Ryazan; pyurova_ta@mail.ru
D. V. Fetisov, post-graduate student, RSREU, Ryazan; morzitko@gmail.com

Theoretical and practical issues of parallel data search algorithms and their implementation on CUDA platform are considered. The relevance of this topic is caused by constant increase in the volume of processed information at present time, as well as the demand for increased speed of data processing.

The aim of the work *is to modify the known algorithms for finding data in one-dimensional array for their use on CPU, followed by the analysis of the efficiency of parallel versions of software modules developed. Thus, the results show that the use of graphics accelerator can significantly increase the speed and, accordingly, the acceleration of applications using data search algorithms. As the amount of data and thread blocks used increases, the performance time of multithreaded application on graphics accelerator is significantly reduced compared to serial implementation on CPU.*

Key words: *multiprocessor, device, graphic memory, kernel, block, thread, host, search, acceleration.*

DOI: 10.21667/1995-4565-2018-65-3-55-62

References

1. **Nvidia CUDA** – negraficheskie vychislenija na graficheskih processorah [Jelektronnyj resurs]. URL: http://www.nvidia.ru (data obrashhenija 11.01.2018) (in Russian).
2. **AMD ATI** – Developer Central [Jelektronnyj resurs]. URL: http://www.amd.com (data obrashhenija 15.01.2018) (in Russian).
3. **Kozlov M. A., Skvorcov S. V.** Algoritmy parallel'noj sortirovki dannyh i ih realizacija na jazyke Clojure. Vestnik Rjazanskogo gosudarstvennogo radiotehnicheskogo universiteta. 2013, no. 4-1 (46), pp. 92-96. (in Russian).
4. **Skvorcov S. V.** Algoritm planirovanija parallel'nyh vychislenij v mnogojadernyh processorah. Radiotehnika. 2016, no. 8, pp. 153-159 (in Russian).
5. **Skvorcov S. V., Pjurova T. A.** Parallel'nye algoritmy sortirovki dannyh i ih realizacija na platforme CUDA. Vestnik Rjazanskogo gosudarstvennogo radiotehnicheskogo universiteta. 2016. no. 58, pp. 42-48 (in Russian).
6. **Korjachko V. P., Skvorcov S. V., Shibanov A. P., Perepelkin D. A.** Metody i tehnologii avto-matizacii proektirovanija vysokoproizvoditel'nyh sistem i komp'juternyh setej. Vestnik Rjazanskogo gosudarstvennogo radiotehnicheskogo universiteta. 2017. no. 60, pp. 94-104 (in Russian).
7. **Korjachko V. P., Skvorcov S. V., Telkov I. A.** Model' planirovanija parallel'nyh processov v superskaljarnyh processorah. Informacionnye tehnologii. 1997. no. 1, pp. 8-12 (in Russian).
8. **Skvorcov S. V.** Celochislennye modeli optimizacii koda po kriteriju vremeni. Informacionnye tehnologii. 1997. no. 10. pp. 2-7 (in Russian).
9. **Rudakov V. E., Skvorcov S. V.** Postroenie bazovogo mnozhestva nezavisimyh putej potokovogo grafa dlja testirovanija programmnyh module. Sistemy upravlenija i informacionnye tehnologii. 2012. Vol. 50, no. 4, pp. 67-70 (in Russian).
10. **Kvodo** – Computing Science & Discrete Match (binarnyj i interpoljacionnyj poisk) [Jelektronnyj resurs]. URL: http://kvodo.ru (data obrashhenija 16.10.2017) (in Russian).
11. **Gergel' V. P.** Vysokoproizvoditel'nye vychislenija dlja mnogoprocessornyh mnogojadernyh si-stem. M.: Izdatel'stvo Moskovskogo universiteta. 2010, 544 p. (in Russian).