УДК: 004.021
DOI: 10.25559/SITITO.14.201802.408-418

ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ ТОЧНЫХ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧ ДИСКРЕТНОЙ ОПТИМИЗАЦИИ НА ГРАФИЧЕСКИХ УСКОРИТЕЛЯХ

М.В. Попов, М.А. Посыпкин
Федеральный исследовательский центр «Информатика и управление» Российской академии наук, г. Москва, Россия

EFFECTIVE REALIZATION OF EXACT ALGORITHMS FOR SOLVING DISCRETE OPTIMIZATION PROBLEMS ON GRAPHIC ACCELERATORS

Michael V. Popov, Mikhail A. Posypkin
Federal Research Center Computer Science and Control of the Russian Academy of Sciences, Moscow, Russia

© Попов М.В., Посыпкин М.А., 2018

Ключевые слова

Задача о ранце; графический ускоритель; параллельные вычисления; CUDA; GPU.

Аннотация

Большинство задач дискретной оптимизации относятся к классу NP-полных задач. Это означает, что алгоритмы, позволяющие найти их точное решение, вообще говоря могут работать с экспоненциальной сложностью относительно длины входных данных. Благодаря прогрессу, сегодня появились технологии, которые пока еще не достаточно широко использовались для реализации методов прикладной оптимизации. К числу таких технологий можно отнести GP GPU (General Purposed Graphical Processing Unit) . Применение данной технологии к хорошо известным алгоритмам может помочь добиться большей эффективности работы. Цель данной работы – исследование возможностей применения параллельных вычислений на видеокартах для решения задач дискретной оптимизации. В качестве целевой задачи выбрана задача об одномерном булевом ранце. Для решения задачи рассмотрены методы получения точного решения – алгоритм полного перебора, являющийся начальной точкой в исследовании, и метод «ветвей и границ», позволяющему сократить перебор благодаря отсеву заведомо неподходящих решений. Рассмотренные алгоритмы оценены с точки зрения количества операций и времени выполнения, реализованы в однопоточной конфигурации центрального процессора, после чего распараллелены на видеокарте. По результатам реализации данных методов, был создан комбинированный алгоритм, объединяющий в себе оба алгоритма для достижения большей эффективности. Для распараллеливания вычислений на графической карте, выбрана технология CUDA. Алгоритмы реализованы на языке C. После реализации алгоритмов, проведено тестирование на различных наборах данных и разных конфигурациях целевой платформы. Представлены результаты экспериментальных исследований, исследовано ускорение работы при использовании параллельных вычислений и проведён сравнительный анализ эффективности работы алгоритмов.

Keywords

Knapsack problem; Graphic accelerator; Parallel calculations; CUDA; GPU.

Abstract

Most of the problems of discrete optimization belong to the class of NP-complete problems. This means that algorithms that can find their exact solution, in general, can work with exponential complexity relative to the length of the input data. Thanks to progress, today there are technologies that have not yet been widely used to implement applied optimization methods. Among these technologies is GP GPU (General Purposed Graphical Processing Unit). The application of this technology to well-known algorithms can help to achieve greater efficiency. The purpose of this paper is to investigate the possibilities of using parallel computations on video cards to solve discrete optimization problems. The problem of a one-dimensional Boolean knapsack was chosen as the target problem. To solve the problem, methods for obtaining an exact solution are considered - the full search algorithm, which is the starting point in the study, and the

Об авторах:

Попов Михаил Владимирович, сотрудник отдела прикладных проблем оптимизации, Вычислительный центр им. А.А. Дородницына, Федеральный исследовательский центр «Информатика и управление» Российской академии наук (119333, Россия, г. Москва, ул. Вавилова, д. 40), ORCID: http://orcid.org/0000-0003-1646-8487, alvopim@gmail.com

Посыпкин Михаил Анатольевич, доктор физико-математических наук, доцент, главный научный сотрудник отдела прикладных проблем оптимизации, Вычислительный центр им. А.А. Дородницына, Федеральный исследовательский центр «Информатика и управление» Российской академии наук (119333, Россия, г. Москва, ул. Вавилова, д. 40), ORCID: http://orcid.org/0000-0002-4143-4353, mposypkin@gmail.com

«branches and boundaries» method, which allows to reduce the search by eliminating obviously inappropriate solutions. The algorithms considered are estimated in terms of the number of operations and execution time, implemented in a single-threaded configuration of the central processor, and then parallelized on a video card. Based on the results of these methods, a combined algorithm was created that combines both algorithms to achieve greater efficiency. For parallelizing the calculations on the graphics card, the CUDA technology is chosen. Algorithms are implemented in C. After the implementation of the algorithms, testing was carried out on various data sets and different configurations of the target platform. The results of experimental studies are presented, the acceleration of work is investigated with the use of parallel computations and a comparative analysis of the efficiency of the algorithms is carried out.

Введение

Задача о ранце [1] – одна из типовых задач дискретной оптимизации, находящая применение в различных областях. Методы, разработанные для ее решения, могут применяться к более широкому классу задач. Задача о ранце формулируется следующим образом: из конечного множества элементов с параметрами «вес» и «стоимость», необходимо найти некоторое подмножество, обладающее наибольшей суммой стоимостей, при этом не превосходящее ограничение по суммарному весу.

Приведем формальную постановку задачи. Имеется N элементов, каждый элемент обладает свойствами: вес – w_i, стоимость – p_i . Также задано ограничение вместимости C. Необходимо найти вектор x для которого сумма стоимостей $\sum_{(i=1)}^{N} = p_i\, x_i$ достигает максимума при заданном ограничении суммарного веса C, $\sum_{(i=1)}^{N} = w_i\, x_i \le C$. Вектор x представляет собой набор из 0 и 1, где $x_i \in \{0,1\}$– индикатор наличия элемента в наборе:

$$\begin{cases} \sum_{i=1}^{N} p_i\, x_i \to max, \\ \sum_{i=1}^{N} w_i\, x_i \le C, \\ x_i \in \{0,1\}. \end{cases}$$

В процессе изучения задачи о ранце было предложено множество методов, предполагающих получение как точного, так и приближенного решения. Очевидный и, вероятно, первым способом, является метод полного перебора, который прост в реализации, но не является оптимальным по времени работы. В 1967 году Питер Колесар [2] применил метод ветвей и границ для решения поставленной задачи. Этот метод был развит в дальнейшем в работах Горовица, Сахни и Писсингера. Также к задаче о ранце применялись методы динамического программирования [3], генетические алгоритмы [4]. С появлением многопроцессорных вычислительных систем, алгоритмы были заново рассмотрены и реализованы для параллельных архитектур. Например, метод ветвей и границ для задачи о ранце был адаптирован для параллельной реализации [5].

По мере распространения технологии GPGPU (General-purpose computing for graphics processing units) [6] было предложено использование графических ускорителей для многопоточных вычислений, многие вычислительные алгоритмы были реализованы на видеокартах. В частности, с использованием технологии CUDA [7, 8] были созданы параллельные реализации метода динамического программирования [9], и метода ветвей и границ [10].

Целью данной работы является разработка и исследование методов решения задачи о ранце, ориентированных на графические ускорители. Рассматриваются методы полного перебора и методов ветвей и границ (вариант Горовица-Сахни). Разработана реализация алгоритма полного перебора с использованием технологии CUDA. Также создан комбинированный алгоритм, совмещающий достоинства обоих подходов. Проведено экспериментальное исследование эффективности разработанных методов.

Архитектура GPU

GPU – (Graphic Processing Unit) графический ускоритель, который изначально применялся для обработки графической информации, после добавления в архитектуру программируемых шейдерных блоков, получил возможность вычислений более широкого спектра применения. Известно несколько технологий для применения графических карт в параллельных вычислениях: Nvidia CUDA, OpenACC, Direct Compute, OpenCL, AMD Firestream, C++ AMP. В данной работе рассмотрены реализации только с применением технологии CUDA.

Основным отличием архитектуры видеокарты от классической архитектуры центрального процессора является наличие большого числа потоковых мультипроцессоров, обладающих множеством параллельно работающих вычислительных ядер. Несмотря на то, что тактовая частота таких ядер существенно меньше частоты ядер центрального процессора, высокая эффективность достигается за счёт параллелизма архитектуры. Различие в архитектурах центрального процессора и графического ускорителя отображено на Рис.1.

Вычисления на ядрах графических ускорителей производятся по схеме SIMD – Single Instruction Multiple Data, что означает один набор команд для выполнения на различных наборах поступающих данных. Для вычислений создаётся функция-ядро «kernel», общая для всех мультипроцессоров и потоков, содержащая программный код. При вызове данной функции необходимо задать параметры сетки (grid), состоящей из блоков потоков и потоков внутри блока. Поток – единичный вычислительный элемент, выполняющийся на скалярном процессоре в составе потокового мультипроцессора. Для вычислений, потоки объединяются в блоки потоков – множества потоков, обрабатываемых одним потоковым мультипроцессором. Блоки потоков являются частью сетки «grid» - множества блоков, которые будут задействованы в ходе вычислений всей программы.

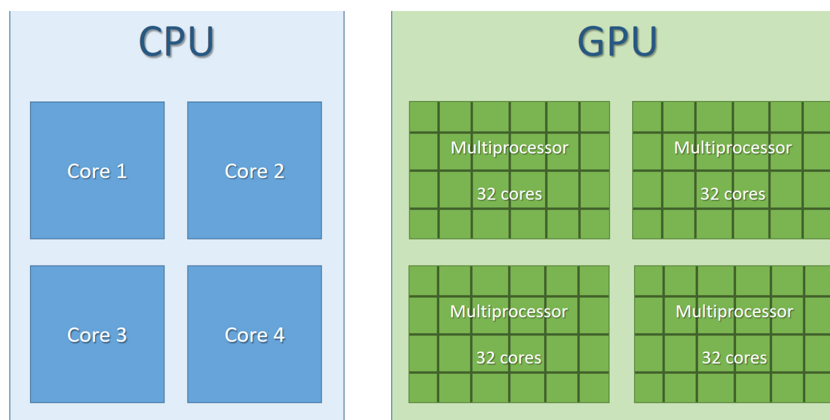

Рис. 1. Сравнение архитектур CPU и GPU
Fig. 1. Comparison of CPU and GPU architectures

Графические карты обладают несколькими типами памяти:

Глобальная обладает наибольшим объёмом, самой низкой скоростью доступа, «срок жизни» - выполнение программы;

Локальная память характеризуется малым объём, невысокой скоростью доступа, «срок жизни» – выполнение 1 потока;

Разделяемая память имеет малый объём, высокую скорость доступа, «срок жизни» – выполнение 1 блока потоков;

Константная память имеет малый объём, очень высокую скорость доступа, «срок жизни» - выполнение программы;

Регистровая память занимает очень маленький объём, имеет самую высокую скорость доступа, «срок жизни» - выполнение 1 потока, и используется компилятором;

Текстурная память используется при работе с графической информацией.

Запуск операций на видеокарте (device) происходит с центрального процессора (host), в том числе, размещение данных из оперативной памяти по конкретным типам используемой памяти графического ускорителя. Способы размещения и использования данных в оперативной памяти могут оказывать существенное влияние на эффективность выполнения приложения.

Метод полного перебора

Метод полного перебора – один из методов получения точного решения в задаче о ранце. Данный метод характеризуется тем, что количество операций напрямую зависит от длины входного вектора, и не зависит от коэффициентов задачи. Сложность алгоритма составляет $O(2^n)$, где n – количество элементов (размерность задачи). Достоинством алгоритма полного перебора является то, что количество операций и время выполнения на конкретном устройстве можно вычислить заранее.

Описание последовательного алгоритма:

1) Вычисляется общее число операций 2^n;

2) В цикле от 0 до 2^n каждое число переводится из десятичной системы в двоичную, то есть превращается в бинарный набор длины n;

3) Каждый коэффициент стоимости и веса умножается на соответствующий его индексу 0 или 1 из бинарного набора, производится сложение полученных значений для данного набора;

4) Если сумма веса не превосходит вместимость *C*, то сумма стоимостей сравнивается с максимально достигнутым значением на данном шаге (если превосходит, то записывается вместо максимального);

5) По окончании цикла выводится максимальное значение суммарной стоимости, достигнутого веса, номера набора. Из номера набора путём бинаризации можно получить искомый вектор $\vec{x}$ (Вместо поиска набора по числу, можно сохранять набор, полученный на шаге 4).

При параллельной реализации с использованием технологии CUDA, первые 3 этапа алгоритма подвергаются небольшим изменениям относительно последовательной версии для выполнения на потоках графической карты. Поиск максимального значения стоимости осуществляется с учетом особенностей архитектуры. Приведем описание алгоритма:

1. Вычисляется общее число операций 2^n;
2. Исходя из этого числа, определяется количество используемых в вычислениях блоков потоков;
3. Вызывается функция на GPU, которая, исходя из значения номера потока и блока потоков на видеокарте, вычисляет свой собственный уникальный бинарный набор;
4. Каждый коэффициент стоимости и веса умножается на соответствующий его индексу элемент (0 или 1) из бинарного набора, производится сложение полученных значений для данного набора;
5. Если сумма веса, полученная на потоке, не превосходит вместимость *C*, то сумма стоимостей записывается в разделяемую память видеокарты;
6. По окончании вычислений одного блока потоков, производится редукционный поиск максимального значения в блоке;
7. Из множества полученных максимальных значений для каждого блока потоков, путём редукции, определяется единственное максимальное число.

Редукция на CUDA – метод, используемый при взаимодействии потоков, который, благодаря особенностям архитектуры видеокарты, позволяет найти максимальное значение в блоке потоков быстрее по отношению к перебору и попарному сравнению значений. Редукционный поиск максимального значения заключается в том, что всё множество полученных значений на блоке потоков делится пополам. После этого, левая половина значений попарно сравнивается с правой, и, если значение из

правой части превосходит соответствующее ему значение из левой части, оно записывается в левую часть. Таким образом, количество параллельных итераций для поиска максимального значения сокращается до , где t – количество обрабатываемых значений. Схема метода представлена на Рис. 2.

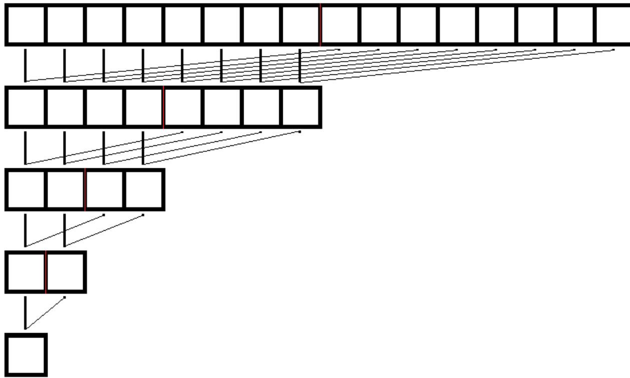

Рис. 2. Редукция
Fig. 2. Reduction

Вычисления на видеокарте производятся каждым потоком независимо, однако за 1 раз 1 потоковый мультипроцессор обрабатывает «warp» – небольшой блок, состоящий из 32 потоков. Таким образом, за выполнение одного warp'а будет получено 32 значения суммарных весов и стоимостей на 1 мультипроцессоре. Следовательно, при наличии только 1 мультипроцессора количество параллельных итераций сокращается с до с 2^n до $2^{(n-5)}$. При наличии процессоров на графическом ускорителе, данное число сокращается до $2^{(n-5)}/k$ итераций.

Реализованный на CUDA алгоритм устроен так, что программа выполнится вне зависимости от используемой видеокарты и её параметров, поскольку максимально допустимое количество выполняемых блоков и потоков автоматически вычисляется исходя из показателей графического ускорителя:

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, 0);
int threads_per_block = deviceProp.maxThreadsDim[0];
int max_blocks = deviceProp.maxGridSize[0]/2 + 1;
```

«threads_per_block» - максимальное количество потоков в блоке;

«max_blocks» - максимальное количество блоков на 1 мультипроцессор.

Используемые данные – коэффициенты веса, стоимости и вместимость ранца, вводятся вручную либо перенаправлением потока ввода из файла. В первоначальной реализации коэффициенты копировались в глобальную память видеокарты, однако в ходе исследований и тестирования было установлено, что копирование данных коэффициентов в константную память видеокарты даёт прирост производительности около 10%, поскольку скорость обращения к константной памяти на видеокарте является самой быстрой. Также на копирование коэффициентов в константную память повлиял тот факт, что коэффициенты не изменяются в ходе работы программы.

Далее выполняется функция-ядро, отвечающая за выполнение 1 блока потоков. Запускается блок из максимального (обычно 1024) числа потоков. В разделяемой памяти видеокарты создаётся 2 массива – один (sh_maxs) для значений сумм стоимостей предметов, вычисляемых на потоках, второй (indices) для записи номеров потоков.

```
extern __shared__ float sh_array[];
float* sh_maxs = (float*)sh_array;
long int* indices = (long int*)&sh_maxs[threads_per_block];
indices[threadIdx.x] = threadIdx.x;
```

Ключевое слово «extern» - означает, что количество выделяемой памяти задаётся при вызове функции.

«__shared__» - использование разделяемой памяти.

«sh_array[]» - массив, размер которого задаётся выделяемым объёмом памяти в вызове функции. В силу особенностей архитектуры, под динамические массивы выделяется один общий участок памяти, который потом делится на части, в зависимости от необходимых объёмов.

«sh_maxs» - суммы стоимостей каждого потока, записываемые в разделяемую память видеокарты.

«indices» - индексы потоков в блоке; используются в процессе редукции для поиска оптимального набора – вектора $\vec{x}$.

Разделяемая память (shared memory) является второй по скорости обращения, но её объём очень мал по сравнению с объёмом глобальной памяти, обращение к которой идёт намного дольше. Поэтому имеет смысл помещать в разделяемую память те данные, к которым происходят частые обращения. Соответственно, выгодно поместить туда массивы сумм и индексов в блоке, поскольку при редукции они будут постоянно перемещаться по выделенным участкам разделяемой памяти. Далее, каждый поток вычисляет свой бинарный набор, исходя из номера потока, и номера блока потоков, после чего выполняет умножение коэффициентов весов и стоимостей на двоичный вектор:

```
#pragma unroll
 for (uint i = 0; i < arraySize; i++){
   th_bin[i] = ((num_to_bin) >> i) % 2;
   th_w_sum += th_bin[i] * coefs[i];
   th_v_sum += th_bin[i] * coefs[i+arraySize];
}
```

«#pragma unroll» - выражение, которое после прочтения компилятором, «разворачивает» цикл, т.е. все операции будут произведены одновременно, а не в цикле.

«th_bin[i]» - массив размерности n (число предметов), вектор

«th_w_sum» - сумма весов на наборе потока.

«th_v_sum» - сумма стоимостей на наборе потока.

«coefs» - коэффициенты веса или стоимости, в зависимости от индекса. Находятся в константной памяти GPU.

Затем выполняется отсев «лишних» значений:

```
sh_maxs[threadIdx.x] = (th_w_sum > C ) ? 0 : th_v_sum;
```

В разделяемую память блока записываются только те значения, которые удовлетворяют условию ограничения вместимости *C*. Иначе записывается 0.

После этого выполняется синхронизация потоков в блоке и начинается редукционный поиск максимального значения целевой функции из вычисленных потоками, входящими в блок:

```
for (uint offset = blockDim.x >> 1; offset >= 1; offset >>= 1){
    if (threadIdx.x < offset){
        if (sh_maxs[threadIdx.x] < sh_maxs[threadIdx.x + offset]){
            sh_maxs[threadIdx.x] = sh_maxs[threadIdx.x + offset];
            indices[threadIdx.x] = indices[threadIdx.x + offset];
        }
    }
    __syncthreads ();}
```

«blockDim» - размер блока – количество потоков в блоке.

«offset» - сдвиг для сравнения элементов; равен половине размера обрабатываемого блока.

После проведения цикла операций, в ячейках разделяемой памяти с индексом 0 оказываются максимальная сумма стоимостей из блока и номер потока, после чего они записываются в глобальную память видеокарты, а вычисления переходят на следующий блок потоков.

После выполнения данных операций на всех блоках, производится повторная редукция по полученным максимальным значениям блоков, которые перезаписываются из глобальной памяти видеокарты в разделяемую. Вместе с максимальными значениями в разделяемой памяти находятся сохранённые индексы, от которых в процессе редукции остаётся единственный оптимальный индекс. Итоговое значение максимального значения и его индекса возвращается в оперативную память ПК для последующего сохранения и вывода. По полученному индексу восстанавливается искомый бинарный вектор $\vec{x}$ оптимальный набор элементов, удовлетворяющий поставленным условиям.

Метод ветвей и границ Горовица-Сахни

Метод ветвей и границ часто работает существенно быстрее по отношению к алгоритму полного перебора, благодаря отсеву заранее неподходящих наборов элементов, однако просчитать количество операций и время работы невозможно. Это происходит из-за того, что число шагов данного алгоритма сильно зависит от коэффициентов задачи. Алгоритм Горовица-Сахни является вариантом «поиска в глубину». Отличием от обычного метода ветвей и границ, и достоинством метода Горовица-Сахни является то, что выполнение алгоритма использует очень мало памяти, поскольку все вычисления производятся на одном двоичном векторе. В изначальном виде между итерациями алгоритма имеются информационные зависимости, из-за чего возможности распараллеливания достаточно ограничены.

Приведем описание последовательного алгоритма:

Все элементы сортируются в порядке уменьшения удельной стоимости – отношения цены к весу предмета;

Начинается «движение вперёд» - аналог «жадного алгоритма» - поместить как можно больше предметов подряд, до того, как сумма весов превысит вместимость ранца. Элементы вектора, «положенные в ранец», помечаются цифрой 1;

Когда суммарный вес превышает вместимость, делается «шаг назад» (backtrack) – удаляется последний положенный элемент, вершина помечается как 0. Выполняется проверка того, какие элементы могут поместиться в рюкзак, и улучшит ли это достигнутое максимальное значение целевой функции. Для этого решается задача линейной релаксации, которая заключается в том, что берётся не весь элемент, а его часть, умещающаяся в остаточную вместимость рюкзака;

Алгоритм выполняется до тех пор, пока станет невозможным сделать «шаг назад».

Чем больше разница у очередной пары «удельных стоимостей», тем эффективнее работа алгоритма, и наоборот. В работе 11] был предложен вариант набора коэффициентов, при котором классический вариант метода ветвей и границ работает неэффективно, выполняя $\frac{2^n}{\sqrt{n}}$ по порядку операций. В этом примере все коэффициенты (и стоимости и веса) равны 2, а вместимость определяется по формуле $2\left\lceil\frac{n}{2}\right\rceil+1$, где – количество предметов. Таким образом, первоначальная постановка задачи о ранце превращается в следующую:

$$\sum_{i=1}^{n} 2\,x_i \rightarrow max,\ \sum_{i=1}^{n} 2\,x_i \le 2\left\lceil\frac{n}{2}\right\rceil + 1,\quad x_i \epsilon\{0,1\}.$$

В данном случае, когда все коэффициенты чётные, а вместимость – нечётная, решение задачи релаксации будет вызываться на каждом «шаге назад», из-за чего не происходит отсев заведомо неоптимальных наборов. В результате эффективность становится меньше, чем у алгоритма полного перебора.

Один из способов распараллеливания вычислений – разбиение исходной задачи на подзадачи, которые в свою очередь передаются на другие вычислительные юниты (потоки, ядра, процессоры) [5]. Иллюстрации к работе данного алгоритма (Рис. 3) показывают организацию бинарного дерева, направления обхода, а также разницу между последовательной и параллельной реализациями.

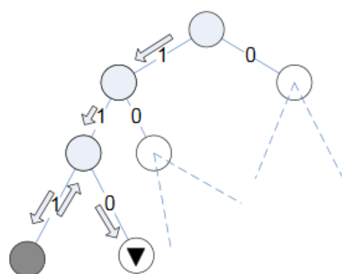

Рис. 3а. Вектор 110

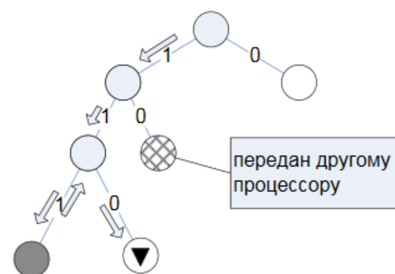

Рис. 3б. Вектор 120

Рис. 3. Метод Горовица-Сахни. а) – последовательный рекурсивный алгоритм; б) – параллельная реализация с разбиением на подзадачи

Fig. 3a. Vector 110 Fig. 3b. Vector 120

Fig. 3. The Horowitz-Sahni method. a) a sequential recursive algorithm; b) a parallel implementation with a subdivision into subtasks

В силу особенностей архитектуры графического ускорителя такой способ хорошо подходит для вычислений на многоядерном центральном процессоре, но не на GPU. Для реализации параллельной версии с использованием графической карты была предложена идея совмещения переборного алгоритма с данным методом, названная в данной работе «Гибридный алгоритм».

Гибридный алгоритм

Данный алгоритм заключается в том, чтобы «зафиксировать» некоторый начальный набор элементов, после чего вычесть из общей вместимости полученный суммарный вес, а на оставшихся предметах использовать метод Горовица-Сахни.

Фиксированная часть. Решение методом полного перебора.	Нефиксированная часть. Решение методом Горовица-Сахни.

Реализация метода Горовица-Сахни на одном потоке GPU не отличается от её реализации на CPU, поэтому параметры решения данной задачи на GPU зависят только от количества элементов, обрабатываемых методом полного перебора.

Описание алгоритма:

1. Определяется номер элемента «k» для деления набора [11];
2. Каждый поток видеокарты создаёт часть бинарного набора от 0 до k;
   2.1) В разделяемую память блока записывается индекс потока;
3. Производятся операции перемножения коэффициентов веса и стоимости на первых k элементах;
4. Каждый поток вычисляет свою «остаточную» вместимость – разность общей вместимости, и полученной суммы веса на шаге 3;
5. Начиная с «k + 1»-го до «n»-го элемента выполняется алгоритм метода Горовица-Сахни для остаточной вместимости;
6. Полученные значения потоков записываются в разделяемую память;
7. Производится поблоковая редукция для поиска максимальной суммы стоимостей;
   7.1) При поиске максимального значения в блоке, записывается номер потока, получившего максимальное значение;
   7.2) Полученный оптимальный бинарный набор на потоке с максимумом, записывается в глобальную память видеокарты;
8. Проводится последняя редукция для определения максимальной суммы стоимостей и определения искомого набора элементов – вектора

Пункт 7.2 добавлен, поскольку полный бинарный набор невозможно восстановить путём перевода номера потока из десятичной системы в двоичную. Приходится записывать каждый оптимальный набор отдельно. Запись идёт в глобальную память, что позволяет хранить большое количество таких наборов.

Реализация гибридного алгоритма будет эффективной при решении задач с «неудачными» с точки зрения стандартного алгоритма метода ветвей и границ, коэффициентами. Это происходит благодаря тому, что фиксация некоторой части вектора позволяет уменьшить размерность подзадачи, решаемой методом ветвей и границ.

Экспериментальное исследование

Наборы коэффициентов генерировались по способу, предложенному в [1]. Коэффициенты генерировались 3 способами с разными параметрами:

Без корреляции:
p_i и w_i – случайные числа в диапазоне $[1: v]$.

Слабая корреляция:
w_i– случайное число в диапазоне $[1: v]$;
p_i – случайное число в диапазоне $[w_i - r : w_i + r]$.

Сильная корреляция:
w_i– случайное число в диапазоне $[1: v]$;
p_i – число равное $w_i + r$.

Ограничение вместимости вычислялось 2 способами:
1) $W = \frac{1}{2}\sum_{i=1}^{n} w_i$
2) $W = 2\,v, v$ – максимум диапазона генерации случайных чисел.

Для каждого способа генерации использовалось по 3 разных диапазона генерации случайных чисел. Помимо данных способов задания коэффициентов, были проведены эксперименты с использованием «примера Финкельштейна» [12]. Для тестирования использовалась размерность задачи в 31 предмет. Данное число стало переходным в решении задачи, поскольку индексы/количество операций перешли с int на long (с 4 байт на 8).

Ниже приведены средние показатели времени выполнения программ на различных конфигурациях ЭВМ для различных наборов коэффициентов. Для каждого набора указывается 2 показателя времени в зависимости от способа вычисления ограничения вместимости ранца. Тестирование реализованных алгоритмов проводилось на 3 разных конфигурациях ЭВМ.

Первая конфигурация:

GPU - Nvidia GeForce 610m, 1 SM, 48 cores, 1024 threads, Architecture - Fermi
CPU – Intel Core I3, 2.3GHZ, single thread
Compiler – gcc 5.4
Cuda compute capability 2.0, Toolkit v7.5
Вторая конфигурация:
GPU - Nvidia GeForce GTX 1060, 10 SM, 1152 cores, 1024 threads, Architecture - Pascal
CPU – Intel Core I7, 2.8-3.8GHZ, single thread
Compiler – clang 4.0
Cuda compute capability 6.1, Toolkit v9.1
Третья конфигурация:
GPU - Nvidia GeForce GTX 1070, 15 SM, 1920 cores, 1024 threads, Architecture - Pascal
CPU – AMD Ryzen 5, 1,6-3.2GHZ, single thread
Compiler – gcc 4.0
Cuda compute capability 6.1, Toolkit v9.1

Сравнение эффективности

Полученные данные собраны в таблицы в порядке исследуемых алгоритмов.

Пояснения к таблицам:

ПК1-ПК3 – конфигурации тестовых ЭВМ

Б.К. – «без корреляции»

Сл.К. – «слабая корреляция»

Сил.К. – «сильная корреляция

Финк. – «пример Финкельштейна»

Цифры 1 и 2 после типа данных указывают на используемое ограничение по весу (указано в главе 3)

В ячейках таблицы указано время выполнения в секундах.

Таблица 1. Полный перебор на CPU. Время в секундах

Table 1. Full brute force on the CPU. Time in seconds

	Б.К.1	Б.К.2	Сл.К.1	Сл.К.2	Сил.К.1	Сил.К.2	Финк.
ПК1	600	600	600	600	600	600	600
ПК2	360	360	360	360	360	360	360
ПК3	420	420	420	420	420	420	420

Для данного алгоритма время выполнения не зависит от коэффициентов, поэтому всё время усреднено для каждого тестового ПК.

Таблица 2. Полный перебор на GPU. Время в секундах

Table 2. Full brute force on the GPU. Time in seconds

	Б.К.1	Б.К.2	Сл.К.1	Сл.К.2	Сил.К.1	Сил.К.2	Финк.
ПК1	28,9	28,9	28,9	28,9	28,9	28,9	28,9
ПК2	1,25	1,25	1,25	1,25	1,25	1,25	1,25
ПК3	0,87	0,87	0,87	0,87	0,87	0,87	0,87

Время выполнения для данного алгоритма также не зависит от коэффициентов задачи.

Таблица 3. Метод ветвей и границ Горовица-Сахни на CPU. Время в секундах

Table 3. The method of branches and boundaries of Horowitz-Sahni on the CPU. Time in seconds

	Б.К.1	Б.К.2	Сл.К.1	Сл.К.2	Сил.К.1	Сил.К.2	Финк.
ПК1	3,45	0,44	2,29	0,4	0,37	1,17	1600
ПК2	1,37	0,3	1,38	0,35	0,49	0,72	900
ПК3	3,95	1,73	1,49	0,39	0,55	0,83	1000

Время работы программ в реализации данного алгоритма очень сильно зависит от коэффициентов, поэтому в некоторых случаях алгоритм может оказаться менее эффективным, чем метод полного перебора, распараллеленный на GPU. Несмотря на то, что метод ветвей и границ считается более оптимальным, при «неудачном» наборе коэффициентов, производятся дополнительные («лишние») действия в силу особенностей алгоритма, относительно стабильного числа операций при полном переборе. Особенно явно эта проблема выражена показателями времени работы при выполнении задачи Финкельштейна.

Таблица 4. Гибридный алгоритм. Время в секундах

Table 4. Hybrid algorithm. Time in seconds

	Б.К.1	Б.К.2	Сл.К.1	Сл.К.2	Сил.К.1	Сил.К.2	Финк.
ПК1	2,73	0,93	0,73	0,51	1,75	0,97	81,2
ПК2	0,28	0,31	0,24	0,11	0,37	0,87	6,09
ПК3	0,31	0,25	0,24	0,07	0,22	0,81	5,7

В среднем, алгоритм на обычных (псевдослучайных) данных работает эффективнее, чем метод Горовица-Сахни. На задаче Финкельштейна среднее ускорение времени работы составляет более 15 раз. Более подробно ускорение описано в таблицах 5-8.

Таблица 5. Относительное ускорение на случайных данных

Table 5. Relative Acceleration on Random Data

		Полный перебор на CPU (А)			Полный перебор на GPU (Б)			Метод Горовица-Сахни (В)			Гибридный алгоритм (Г)		
		П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3
А	П.К.1	1	1,67	1,43	20,76	480	689,6	437,9	771,2	402,7	472,4	1666	1818
	П.К.2	0,6	1	0,86	12,46	288	413,8	262,7	462,7	241,6	283,5	1000	1090
	П.К.3	0,7	1,17	1	14,53	336	482,8	306,6	539,8	281,9	330,7	1167	1273
Б	П.К.1	0,05	0,08	0,069	1	23,12	33,21	21,09	37,14	19,39	22,75	80,27	87,57
	П.К.2	21E-4	35E-4	3E-3	0,043	1	1,44	0,91	1,61	0,84	0,98	3,47	3,78
	П.К.3	14E-4	24E-4	21E-4	0,03	0,69	1	0,63	1,12	0,58	0,68	2,42	2,63
В	П.К.1	23E-4	38E-4	32E-4	0,047	1,1	1,57	1	1,76	0,92	1,08	3,81	4,15
	П.К.2	13E-4	21E-4	18E-4	0,026	0,62	0,89	0,57	1	0,52	0,61	2,16	2,36
	П.К.3	24E-4	41E-4	35E-4	0,051	1,19	1,71	1,09	1,92	1	1,17	4,14	4,51
Г	П.К.1	21E-4	35E-4	0,003	0,04	1,02	1,47	0,925	1,64	0,85	1	3,53	3,84
	П.К.2	6E-4	0,001	8,6E-4	0,012	0,29	0,41	0,26	0,46	0,24	0,28	1	1,09
	П.К.3	5,5E-4	9,1E-4	7,8E-4	0,01	0,26	0,38	0,24	0,42	0,22	0,26	0,92	1

Данная таблица содержит отношение среднего времени работы на всех наборах в зависимости от сравниваемых конфигураций ЭВМ (П.К.1-П.К.3). Эти показатели ускорения работы позволяют оценить возможности применения алгоритмов на обычных, близких к реальным прикладным задачам, данных. Помимо измерения эффективности работы самих алгоритмов, таблица позволяет изучить используемые архитектуры и оценить возможности конкретных компонентов тестовых ЭВМ. Если исключить фактор влияния конфигурации тестового ПК, обобщив результаты работы каждого алгоритма, можно получить следующую таблицу (таблица 6):

Таблица 6. Обобщённое ускорение по алгоритмам на случайных данных

Table 6. General Acceleration by Algorithms on Random Data

	Полный перебор на CPU	Полный перебор на GPU	Метод Горовица-Сахни	Гибридный алгоритм
Перебор на CPU	1	44,487	382,377	707,69
Перебор на GPU	0,0224	1	8,595	15,907
М. Горовица-Сахни	0,00261	0,1163	1	1,8507
Гибридный алгоритм	0,00141	0,0628	0,5403	1

Данная таблица наглядно показывает, что на обычных (псевдослучайных) данных, алгоритмы в порядке быстродействия располагаются следующим образом:

1) Полный перебор на CPU;

2) Полный перебор на GPU;

3) Метод ветвей и границ Горовица-Сахни (на CPU);

4) Гибридный алгоритм (на GPU).

Таким образом, неоптимизированный алгоритм полного перебора на одном потоке центрального процессора оказался наименее эффективным для обычных задач. Метод полного перебора с реализацией на графическом ускорителе оказался намного быстрее, но оптимизированный алгоритм метода ветвей и границ на основе алгоритма Горвица-Сахни работает эффективнее на реальных данных. Однако, комбинирование алгоритма полного перебора и метода Горвица-Сахни оказалось быстрее, поскольку метод перебора на GPU выполняет заранее известное количество команд, благодаря чему размерность метода ветвей и границ уменьшается, а соответственно, уменьшается количество «лишних» действий алгоритма. Количество операций, выполняемых на методе перебора в гибридном алгоритме, зависит от номера элемента, на котором происходит разделение задачи на 2 части. Оно равно , k – номер разделяющего элемента. Правильный выбор числа является сложной задачей, требующей нахождения баланса между распараллеливанием и эффективностью сокращения поиска.

Теперь рассмотрим эффективность алгоритмов на «примере Финкельштейна» (Таблица 7).

Таблица 7. Относительное ускорение на задаче Финкельштейна

Table 7. Relative Acceleration on the Finkelstein problem

		Полный перебор на CPU (А)			Полный перебор на GPU (Б)			Метод Горовица-Сахни (В)			Гибридный алгоритм (Г)		
		П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3	П.К.1	П.К.2	П.К.3
А	П.К.1	1	1,67	1,43	20,76	480	689,6	0,375	0,67	0,6	7,39	98,52	105,3
	П.К.2	0,6	1	0,86	12,46	288	413,8	0,225	0,4	0,36	4,43	59,11	63,15
	П.К.3	0,7	1,17	1	14,53	336	482,8	0,262	0,47	0,42	5,17	68,96	73,68
Б	П.К.1	0,048	0,08	0,069	1	23,12	33,21	0,018	0,032	0,029	0,356	4,75	5,07
	П.К.2	0,002	35E-4	29E-4	0,043	1	1,44	7,8E-4	14E-4	12E-4	0,015	0,205	0,219
	П.К.3	14E-4	24E-4	20E-4	0,03	0,69	1	5,4E-4	9,6E-4	8,7E-4	0,011	0,143	0,153
В	П.К.1	2,66	4,44	3,82	55,5	1282	1852	1	1,77	1,6	19,70	262,7	280,7
	П.К.2	1,49	2,5	2,12	31,25	714,2	1042	0,56	1	0,9	11,08	147,8	157,9
	П.К.3	1,67	2,77	2,38	34,48	833,3	1149	0,625	1,1	1	12,32	164,7	175,4
Г	П.К.1	0,135	0,22	0,19	2,8	66,6	90,9	0,05	0,09	0,08	1	13,33	14,24
	П.К.2	0,01	0,016	0,014	0,21	4,88	6,99	38E-4	67E-4	6E-3	0,075	1	1,07
	П.К.3	0,009	0,015	0,013	0,19	4,57	6,54	35E-4	63E-4	57E-4	0,07	0,934	1

По данной таблице, метод ветвей и границ оказывается наименее эффективным при решении задачи о ранце на условиях примера Финкельштейна. Стоит отметить, что даже метод полного перебора на одном потоке центрального процессора, который на случайных данных оказался неэффективен, превосходит метод ветвей и границ в несколько раз. Для большей наглядности, проведём такую же операцию агрегирования и усреднения производительности вне зависимости от конфигурации вычислительной системы, чтобы оценить быстродействие самих алгоритмов. Данные обобщения производительности приведены в таблице 8:

Таблица 8.Обобщённое ускорение по алгоритмам на примере Финкельштейна

Table 8. Generalized acceleration by algorithms on the example of Finkelstein

	Полный перебор на CPU	Полный перебор на GPU	Метод Горовица-Сахни	Гибридный алгоритм
Перебор на CPU	1	44,487	0,394	9,225
Перебор на GPU	0,0224	1	0,0086	0,2098
М. Горовица-Сахни	2,536	112,823	1	23,6796
Гибридный алгоритм	0,1071	4,7646	0,0422	1

Полученная таблица показывает, что параллельная реализация на GPU алгоритма полного перебора оказалась намного быстрее оптимизированного алгоритма метода ветвей и границ. В свою очередь, гибридный алгоритм, совмещающий в себе перебор и алгоритм Горовица-Сахни, реализованный на графическом ускорителе, оказался менее эффективным. Данные результаты показывают, что в зависимости от конкретного примера, эффективность методов изменяется, а соответственно, необходимо подбирать методы, исходя из поставленных условий.

Таким образом, для примера Финкельштейна, алгоритмы можно упорядочить по увеличению производительности следующим образом:

1) Метод ветвей и границ на основе алгоритма Горвица-Сахни;

2) Полный перебор на CPU;

3) Гибридный алгоритм;

4) Полный перебор на GPU;

Несмотря на то, что метод Горовица-Сахни является одним из самых быстрых при решении обычных задач, в некоторых случаях он может оказаться крайне неэффективным, что стоит учитывать при выборе метода решения прикладной задачи, основанной на задаче о ранце.

Предположим, что если усреднить полученное ускорение, вне зависимости от коэффициентов, будет получена средняя эффективность, которая будет характеризовать исследуемые реа-

лизации алгоритмов, то эти коэффициенты объективно покажут производительность алгоритмов по количеству операций, не смотря на «лишние операции» метода Горовица-Сахни. Средние значения отображены в таблице 9:

Таблица 9. Усреднённая эффективность алгоритмов

Table 9. Average efficiency of algorithms

	Полный перебор на CPU	Полный перебор на GPU	Метод Горовица-Сахни	Гибридный алгоритм
Перебор на CPU	1	44,487	191,3855	358,4575
Перебор на GPU	0,0224	1	4,3019	8,0584
М. Горовица-Сахни	1,269	56,4701	1	12,7651
Гибридный алгоритм	0,00542	2,4137	0,2912	1

Данные показатели являются наиболее объективными, при предположении, что начальные условия типа задачи Ю.Ю. Финкельштейна используются с такой же частотой, как и реальные (псевдослучайные) данные. Из этой таблицы следует, что:

1. Наиболее эффективным алгоритмом для решения задачи о ранце является «Гибридный алгоритм», совмещающий в себе метод ветвей и границ и метод полного перебора с использованием вычислительных возможностей графических ускорителей;
2. Применение метода ветвей и границ на основе алгоритма Горовица-Сахни является более эффективным, чем метод полного перебора на видеокарте. Однако показатели ускорения достаточно близки, что позволяет предположить равноценность данных;
3. Метод полного перебора, реализованный на одном потоке центрального процессора, является самым простым по реализации, но самым неэффективным.

Выводы

Задача о ранце может быть решена различными способами. В данной работе были рассмотрены 2 самых известных способа – метод полного перебора и метод ветвей и границ, и 2 алгоритма, использующие возможности графических ускорителей для вычислений. Показано, что использование видеокарты для вычислений значительно увеличивает эффективность работы алгоритмов, вне зависимости от постановки задачи. Метод полного перебора, реализованный на центральном процессоре являлся отправной точкой в исследовании, однако, при проведении тестов, выяснилось, что метод ветвей и границ может работать менее эффективно, поскольку его производительность сильно зависит от начальных условий задачи.

Для распараллеливания метода ветвей и границ было принято решение реализовать комбинированный алгоритм, совмещающий в себе метод полного перебора с алгоритмом Горвица-Сахни, использующий для вычислений возможности графических ускорителей. Было проведено множество различных тестов на разных начальных условиях, результаты которых подтвердили начальную гипотезу – совмещение передовых вычислительных технологий и отлаженных оптимизированных алгоритмов, позволит добиться более эффективных реализаций алгоритмов в области дискретной оптимизации.

Благодарности

Работа выполнена при финансовой поддержке Программы фундаментальных исследований Президиума РАН № 27 «Фундаментальные проблемы решения сложных задач практических задач с помощью суперкомпьютеров»

Список использованных источников

[1] *Martello S., Toth P.* Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., New York, NY, USA. 1990.

[2] *Kolesar P.J.* A Branch and Bound Algorithm for the Knapsack Problem // Management science. 1967. Vol. 13, issue 9. Pp. 609-772. DOI: 10.1287/mnsc.13.9.723

[3] *Martello S., Pisinger D., Toth P.* Dynamic programming and strong bounds for the 0-1 knapsack problem // Management Science. 1999. Vol. 45, issue 3. Pp. 297-454. DOI: 10.1287/mnsc.45.3.414

[4] *Chu P.C., Beasley J.E.* A Genetic Algorithm for the Multidimensional Knapsack Problem // Journal of Heuristics. 1998. Vol. 4, issue 1. Pp. 63-86. DOI: 10.1023/A:100964240

[5] *Posypkin M.A., Sigal I.K.* A combined parallel algorithm for solving the knapsack problem // Journal of Computer and Systems Sciences International. 2008. Vol. 47, issue 4. Pp. 543-551. DOI: 10.1134/S1064230708040072

[6] *Harris M.* Mapping computational concepts to GPUs // ACM SIGGRAPH 2005 Courses (SIGGRAPH '05), John Fujii (Ed.). ACM, New York, NY, USA, 2005. Article 50. DOI: 10.1145/1198555.1198768

[7] Официальное руководство по программированию на CUDA, вер. 1.1 [Электронный ресурс] // CUDA Programming Guide. Chapter 1. Introduction to CUDA → 1.2 CUDA: A New Architecture for Computing on the GPU. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (дата обращения: 26.04.2018).

[8] Параллельные вычисления на GPU. Архитектура и программная модель CUDA / А.В. Боресков, А.А. Харламов, Н.Д. Марковский и др. Москва: Издательство МГУ, 2015. 336 с.

[9] *Boyer V., El Baz D., Elkihel M.* Solving knapsack problems on GPU // Computers & Operations Research. 2012. Vol. 39, issue 1. Pp. 42-47. DOI: 10.1016/j.cor.2011.03.014

[10] *Lalami M.E., El-Baz D.* GPU Implementation of the branch and bound method for knapsack problems // Proceedings of 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Shanghai, China, 2012. Pp. 1769-1777. DOI: 10.1109/IPDPSW.2012.219

[11] *Колпаков Р.М., Посыпкин М.А.* О наилучшем выборе переменной ветвления в задаче о сумме подмножеств // Дискретная математика. 2017. Т. 29, №. 1. С. 51-58. DOI: 10.4213/dm1405

[12] *Финкельштейн Ю.Ю.* Приближенные методы и прикладные задачи дискретного программирования. М.: Наука, 1976. 265 с.

[13] *Сигал И.Х., Иванова А.П.* Введение в прикладное дискретное программирование: модели и вычислительные алгоритмы. Изд. 2-е, исп. и доп. М.: ФИЗМАТЛИТ, 2007. 304 с.

[14] *Keller H., Pfershy U., Pisinger D.* Knapsack Problems. Springer, Berlin, Heidelberg, 2004. 548 p. DOI: 10.1007/978-3-540-24777-7

[15] *Левитин А.В.* Алгоритмы. Введение в разработку и анализ. М.: Издательский дом «Вильямс», 2006. 576 с.

[16] *Седжвик Р.* Фундаментальные алгоритмы на С. Пер.с англ. М. и др.: ДиасофтЮП, 2003. 670 с.

[17] *Suri B.* Accelerating the knapsack problem on GPUs. Linköping, Sweden, 2011. 87 p.

[18] *Ristovski Z.* et al. Parallel implementation of the modified subset sum problem in CUDA // Proceedings of 2014 22nd Telecommunications Forum Telfor (TELFOR), Belgrade, 2014. Pp. 923-926. DOI: 10.1109/TELFOR.2014.7034556

[19] *Kellerer H., Pferschy U., Pisinger D.* Knapsack Problems. Springer-Verlag Berlin Heidelberg, 2004. 548 p. DOI: 10.1007/978-3-540-24777-7

[20] *Jaros J., Pospichal P.* A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark / Di Chio C. et al. (eds) // Applications of Evolutionary Computation. EvoApplications 2012. Lecture Notes in Computer Science, vol. 7248. Springer, Berlin, Heidelberg, 2012. Pp. 426-435. DOI: 10.1007/978-3-642-29178-4_43

[21] *Li K.* et al. A cost-optimal parallel algorithm for the 0-1 knapsack and its performance on multicore CPU and GPU implementations // Parallel Computing. 2015. Vol. 43. Pp. 27-42. DOI: 10.1016/j.parco.2015.01.004

[22] *Boukedjar A., Lalami M.E., El-Baz D.* Parallel branch and bound on a CPU-GPU system // Proceedings of 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing. Garching, 2012. Pp. 392-398. DOI: 10.1109/PDP.2012.23

[23] *Crawford M., Toth D.* Parallelization of the Knapsack Problem as an Introductory Experience in Parallel Computing // Journal of Computational Science Education. 2013. Vol. 4, issue 1. Pp. 35-39. DOI: 10.22369/issn.2153-4136/4/1/6

[24] *Колпаков Р.М., Посыпкин М.А.* Верхняя и нижняя оценки трудоемкости метода ветвей и границ для задачи о ранце // Дискретная математика. 2010. Т. 22, №. 1. С. 58-73. DOI: 10.4213/dm1084

[25] *Посыпкин М.А., Сигал И.Х.* Исследование алгоритмов параллельных вычислений в задачах дискретной оптимизации ранцевого типа // Журнал вычислительной математики и математической физики. 2005. Т. 45, № 10. С. 1801-1809.

Поступила 26.04.2018; принята в печать 03.06.2018; опубликована онлайн 30.06.2018.

References

[1] Martello S., Toth P. Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc., New York, NY, USA. 1990.

[2] Kolesar P.J. A Branch and Bound Algorithm for the Knapsack Problem. *Management science*. 1967; 13(9): 609-772. DOI: 10.1287/mnsc.13.9.723

[3] Martello S., Pisinger D., Toth P. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*. 1999; 45(3):297-454. DOI: 10.1287/mnsc.45.3.414

[4] Chu P.C., Beasley J.E. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*. 1998; 4(1):63-86. DOI: 10.1023/A:100964240

[5] Posypkin M.A., Sigal I.K. A combined parallel algorithm for solving the knapsack problem. *Journal of Computer and Systems Sciences International*. 2008; 47(4):543-551. DOI: 10.1134/S1064230708040072

[6] Harris M. Mapping computational concepts to GPUs. *ACM SIGGRAPH 2005 Courses (SIGGRAPH '05)*, John Fujii (Ed.). ACM, New York, NY, USA, 2005. Article 50. DOI: 10.1145/1198555.1198768

[7] Official programming guide for CUDA, ver. 1.1. CUDA Programming Guide. Chapter 1. Introduction to CUDA → 1.2 CUDA: A New Architecture for Computing on the GPU. Available at: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf (accessed 26.04.2018).

[8] Boreskov A.V., Kharlamov A.A., Markovsky N.D. Foreword: Sadovnichiy V.A. Parallel'nye vychisleniya na GPU. Arhitektura i programmnaya model' CUDA [Parallel computing on the GPU. Architecture and software model CUDA]. Moscow: MSU. 2015. 336 p. (In Russian)

[9] Boyer V., El Baz D., Elkihel M. Solving knapsack problems on GPU. *Computers & Operations Research*. 2012; 39(1):42-47. DOI: 10.1016/j.cor.2011.03.014

[10] Lalami M.E., El-Baz D. GPU Implementation of the branch and bound method for knapsack problems. *Proceedings of 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, Shanghai, China, 2012. pp. 1769-1777. DOI: 10.1109/IPDPSW.2012.219

[11] Kolpakov R.M., Posypkin M.A. On the best choice of a branching variable in the subset sum problem. *Discrete Mathematics and Applications*. 2018; 28(1):29-34. DOI: 10.1515/dma-2018-0004

[12] Finkelstein Yu.Yu. Priblizhennye metody i prikladnye zadachi diskretnogo programmirovaniya [Approximate methods and applied problems of discrete programming]. Moscow: Nauka, 1976. 265 p. (In Russian)

[13] Sigal I.K., Ivanova A.P. Vvedenie v prikladnoe diskretnoe programmirovanie: modeli i vychislitel'nye algoritmy [Introduction to Applied Discrete Programming: Models and Computational Algorithms]. Moscow: Fizmatlit, 2007. 304 p. (In Russian)

[14] Keller H., Pfershy U., Pisinger D. Knapsack Problems. Springer, Berlin, Heidelberg, 2004. 548 p. DOI: 10.1007/978-3-540-24777-7

[15] Levitin A.V. Algoritmy: vvedenie v razrabotku i anali [Algorithms: Introduction to Development and Analysis]. M.: Williams Publishing House, 2006. 576 p. (In Russian)

[16] Sedgwick R. Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition. Addison-Wesley Professional, 1998. 752 p.

[17] Suri B. Accelerating the knapsack problem on GPUs. Linköping, Sweden, 2011. 87 p.

[18] Ristovski Z. et al. Parallel implementation of the modified

subset sum problem in CUDA. *Proceedings of 2014 22nd Telecommunications Forum Telfor (TELFOR)*, Belgrade, 2014. pp. 923-926. DOI: 10.1109/TELFOR.2014.7034556

[19] Kellerer H., Pferschy U., Pisinger D. Knapsack Problems. Springer-Verlag Berlin Heidelberg, 2004. 548 p. DOI: 10.1007/978-3-540-24777-7

[20] Jaros J., Pospichal P. A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark. In: Di Chio C. et al. (eds) Applications of Evolutionary Computation. EvoApplications 2012. Lecture Notes in Computer Science, vol. 7248. Springer, Berlin, Heidelberg, 2012. pp. 426-435. DOI: 10.1007/978-3-642-29178-4_43

[21] Li K. et al. A cost-optimal parallel algorithm for the 0-1 knapsack and its performance on multicore CPU and GPU implementations. *Parallel Computing*. 2015; 43:27-42. DOI: 10.1016/j.parco.2015.01.004

[22] Boukedjar A., Lalami M.E., El-Baz D. Parallel branch and bound on a CPU-GPU system. *Proceedings of 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Garching, 2012. pp. 392-398. DOI: 10.1109/PDP.2012.23

[23] Crawford M., Toth D. Parallelization of the Knapsack Problem as an Introductory Experience in Parallel Computing. *Journal of Computational Science Education*. 2013; 4(1):35-39. DOI: 10.22369/issn.2153-4136/4/1/6

[24] Kolpakov R.M., Posypkin M.A. Upper and lower bounds for the complexity of the branch and bound method for the knapsack problem. *Discrete Mathematics and Applications*. 2010; 20(1):95-112. DOI: 10.1515/dma.2010.006

[25] Posypkin M.A., Slgal I.Kh. Investigation of algorithms for parallel computations in knapsack-type discrete optimization problems. *Computational Mathematics and Mathematical Physics*. 2005; 45(10):1735-1742.

Submitted 26.04.2018; revised 03.06.2018;
published online 30.06.2018.

About the authors:

Michael V. Popov, employee of the department of applied optimization problems, Dorodnicyn Computing Centre, Federal Research Center Computer Science and Control of the Russian Academy of Sciences (40 Vavilova Str., Moscow 119333, Russia), ORCID: http://orcid.org/0000-0003-1646-8487, alvopim@gmail.com

Michael A. Posypkin, Doctor of Physical and Mathematical Sciences, Associate Professor, Head of the Department, Dorodnicyn Computing Centre, Federal Research Center Computer Science and Control of the Russian Academy of Sciences (40 Vavilova Str., Moscow 119333, Russia), ORCID: http://orcid.org/0000-0002-4143-4353, mposypkin@gmail.com

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted reuse, distribution, and reproduction in any medium provided the original work is properly cited.