$$\frac{\left(1+x_i\right)^2}{d\left(x_i\right)} \le \frac{4}{d_2},\ i=\overline{2,n}. \tag{18}$$

При $x_i \in K_1$, $i=\overline{2,n}$, неравенства (18) обращаются в равенства. В силу (18)

$$D\left\{\tilde{\theta}_1\right\} \ge \frac{d_2}{4\left(n-1\right)},\ i=\overline{2,n}. \tag{19}$$

Нижняя граница в (19) достигается, когда $x_i \in K_1, i=\overline{2,n}.$ Утверждение (14) доказано, и с вероятностью 1 точный *D*-оптимальный план экспериментов имеет вид (12). При $x_i \in K_1, i=\overline{2,n},$ оценка (16) обращается в оценку (13) для параметра θ_1. Доказательство теоремы завершено.

Следствие. *Если множество* $K_1 \setminus \{1\}$ *пусто, то в условиях теоремы 4 D-оптимальный план экспериментов принимает вид*

$$\varepsilon_n^0 = \begin{Bmatrix} -1, & 1 \\ 1, & n-1 \end{Bmatrix}. \tag{20}$$

Оценки неизвестных параметров, построенные по плану (20)*, не зависят от значений дисперсий и равны*

$$\tilde{\theta}_1 = \frac{1}{2}\left(\frac{1}{n-1}\sum_{i=2}^{n} y_i - y_{(-1)}\right),\ \tilde{\theta}_0 = \tilde{\theta}_1 + y_{(-1)},$$

где y_i *– значения наблюдений в точке 1.*

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Федоров В. В.* Теория оптимального эксперимента. М., 1971.

2. *Кирлица В. П. D*-оптимальные планы экспериментов, робастные относительно изменения дисперсии наблюдений, для линии регрессии // Вестн. БГУ. Сер. 1, Физика. Математика. Информатика. 2008. № 3. С. 89–92.

Поступила в редакцию 03.02.2015.

Валерий Петрович Кирлица – кандидат физико-математических наук, доцент кафедры математического моделирования и анализа данных факультета прикладной математики и информатики БГУ.

УДК 004.021

А. А. КОЗИК, А. П. ПОБЕГАЙЛО

УСКОРЕНИЕ ПАРАЛЛЕЛЬНОГО ПОИСКА В ШИРИНУ В ГРАФАХ ПРИ ВЫЧИСЛЕНИЯХ НА GPU

Решается задача ускорения работы алгоритма поиска в ширину на графах при вычислениях с помощью графических процессоров. Ускорение алгоритма достигается посредством кодирования матрицы смежности графа специальным образом, учитывающим программно-аппаратную архитектуру CUDA, что дает возможность значительно улучшить производительность рассматриваемого алгоритма при исполнении на этой платформе. Представленный способ кодирования матрицы смежности позволяет снизить накладные расходы на операции чтения/записи из глобальной памяти, необходимый объем памяти для представления графа, а также существенно уменьшить время, затрачиваемое на загрузку данных, которое в других случаях может превышать время выполнения самого алгоритма.

Предложенная реализация параллельного алгоритма поиска в ширину может быть использована при решении задач, включающих поиск кратчайшего пути в графах, для случайных графов с высокой вероятностью появления ребра. Несмотря на избыточность вычислений, при высоких значениях вероятности *p* – появления ребра графа – полезная нагрузка на GPU остается на уровне 96,7 %.

Ключевые слова: алгоритмы на графах; параллельный поиск в ширину; кодирование матрицы смежности; GPU; CUDA.

The article solves the problem of accelerating parallel breadth first search algorithm on GPU. Acceleration of the algorithm is obtained by special coding of graph adjacency matrix, taking into account CUDA architecture. This coding improves the algorithm efficiency on CUDA platform because reduces the number of read/write operations from the global memory and the time needed to load data. The proposed implementation of the algorithm can be used for solving problems on graph, which use breadth first search as basis computation and have not very sparse adjacency matrix. Despite the redundancy calculation, if the probability of graph edges occurrences is high then the payload on GPU remains at the level 96,7 %.

Key words: algorithms on graphs; parallel breadth first search; coding adjacency matrix; GPU; CUDA.

Одним из базовых алгоритмов в теории графов является поиск в ширину, который используется как автономно, так и в качестве основы для решения различных алгоритмических задач в теории графов. Поэтому особенно актуально ускорение работы данного алгоритма при использовании графических процессоров (GPU). Стоит также отметить, что этот алгоритм применяется в тестах на производительность Graph 500 [1]. В настоящей статье решается задача ускорения работы параллельного алгоритма поиска в ширину в графах при вычислениях на GPU. Ускорение, по сравнению с известными ранее параллельными версиями алгоритма, достигается посредством кодирования матрицы смежности графа специальным образом, учитывающим программно-аппаратную архитектуру CUDA. Предложенная в статье реализация параллельного алгоритма поиска в ширину может быть использована при решении задач, включающих поиск кратчайшего пути в графах с высокой вероятностью появления ребра.

Существующие подходы к параллельной реализации поиска в ширину

Рассмотрим два основных подхода к реализации параллельных алгоритмов поиска в ширину в графах. Первый базируется на использовании очередей при обходе вершин графа. Этот подход предназначен для реализации на параллельных системах, имеющих MIMD-архитектуру, которые обеспечивают развитые средства синхронизации и атомарные операции для доступа к общей памяти параллельных нитей. Как показывают исследования, параллельные алгоритмы, построенные на основе очередей, иногда проигрывают по производительности даже своим последовательным аналогам [2]. Это происходит в силу затрат на организацию очереди, частое использование синхронизации нитей и атомарных операций для доступа к общей памяти. Второй подход предназначен для реализации на параллельных системах, имеющих SIMD-архитектуру, которые обеспечивают только барьерную синхронизацию параллельных нитей, и базируется на полном обходе всех вершин графа на каждой итерации и использовании только барьерной синхронизации потоков при завершении итерации. Графические процессоры организованы как SIMD-машины, поэтому для реализации поиска в ширину на GPU используется именно второй подход. На основе этих двух основных подходов могут быть реализованы гибридные алгоритмы [3–5].

При реализации параллельного алгоритма поиска в ширину на SIMD-машинах важным моментом является организация такого хранения матрицы смежности графа, которое обеспечивает по возможности равномерную загрузку параллельных потоков. Для хранения матрицы смежности при вычислениях на GPU в основном используется формат CSR (Compressed Sparse Row), обеспечивающий хранение матрицы смежности в трех массивах, один из которых содержит элементы матрицы смежности, а два других – номера строк и столбцов элементов матрицы смежности. Этот формат хранения матрицы смежности используется при реализации различных вариантов поиска в ширину на взвешенных графах [6–10]. В случае если граф является невзвешенным, то использование массива, содержащего элементы матрицы смежности, необязательно. Однако этот подход больше ориентирован на сжатие матрицы смежности взвешенного графа, чем на архитектуру GPU.

Ускорение параллельного алгоритма поиска в ширину на графе

Исходя из особенностей архитектуры CUDA [11, 12], можно предположить, что при обеспечении минимального ветвления нитей (threads), максимальной равномерной загрузки канатов (warps), а также снижения накладных расходов на операции с памятью можно достигнуть повышения производительности алгоритма. Для целей этой работы предлагается кодировка представления графа следующим образом. Матрица смежности графа кодируется матрицей, элементами которой являются 32-битные целые числа, содержащей N строк и $N/32$ столбцов с округлением к большему целому числу, где N – количество вершин графа. Далее, для удобства будем именовать новую матрицу CCAM (Coded for CUDA Adjacency Matrix). Каждый бит элемента CCAM-матрицы соответствует одному значению исходной матрицы смежности. Отметим важную особенность такого кодирования: побитовая индексация осуществляется построчно, а если N не кратно 32, то последний элемент строки CCAM-матрицы дополняется нулевыми битами так, чтобы общее количество битов в строке было кратно 32.

Псевдокод итеративного запуска параллельного алгоритма поиска в ширину на CPU приведен ниже.

```
foreach n in aD: aD[n] = INF;
aD[nS] = 0; nL = 0;
do:
 cuda_kernel<<<grid_dim, block_dim>>> (aG, aD, nL);
 nL++;
while (!isCompleted<<<grid_dim, block_dim>>>(aD));
```

Здесь aD – массив расстояний до каждой вершины; nL – номер текущей итерации; aG – данные о графе в формате CCAM; nS – индекс начальной вершины; cuda_kernel() – функция обхода

графа в ширину; isCompleted() – функция, определяющая изменения в массиве расстояний; grid_dim, block_dim – переменные, задающие размерности решетки и блока нитей соответственно. Функция isCompleted() просто ищет изменения в массиве расстояний, если изменений не было, то алгоритм обошел все доступные из начальной вершины и обход завершен.

Реализацию параллельного алгоритма поиска в ширину выполняет функция-ядро cuda_kernel(), код которой приведен ниже.

```
typedef unsigned short sbyte;
typedef unsigned int nbyte;
typedef unsigned long lbyte;
__global__ void cuda_kernel(lbyte* aG, int* aD, sbyte* nL) {
 Nbyte idN = blockIdx.x; // индекс вершины
 nbyte idNg;        // индекс смежной вершины
 nbyte idB;         // индекс бита
 nbyte idD;         // индекс в массиве расстояний
 nbyte node_idx = N / 32; // количество элементов в строке CCAM
 lbyte idx;         // индекс в линейном представлении матрицы
 // цикл по всем вершинам
 while(idN < N) {
  if(aD[idN] == *nL) {           // если текущая вершина не пройдена
   idNg = threadIdx.x / 32;
   idB = 1 << threadIdx.x % 32;
   while(idNg < node_idx) {      // просматриваем смежные вершины
    idx = idN * node_idx + idNg;
    if((aG[idx] & idB) > 0) {     // если вершины смежны
     idD = idNG*32 + threadIdx.x%32;
     if(aD[idD] == INF)       // если вершина не пройдена
      aD[idD] = *nL + 1;
    }
    idNg += blockDim.x / 32;
   }
  }
  idN += gridDim.x;
 } __syncthreads();
}
```

Здесь для каждой нити объявляются переменные, необходимые для индексации. Далее, для вершин, обрабатываемых на текущей итерации nL, определяем индексы и просматриваем вершины, соседние с вершинами текущего слоя. При этом в рамках одного каната рассматривается только один элемент из CCAM-матрицы.

На каждой итерации организуется 1024 (максимально возможное число, если использовать одномерную индексацию) блока нитей. Один, *i*-й, блок составляет вычисления, реализующие обработку строк матрицы смежности с номерами i, $i + 1024$, $i + 2048$, … . В каждом блоке организуется 1024 нити.

Отличительная особенность предлагаемого подхода связана с адаптацией к вычислениям на GPU и CUDA [13]. Так, 32-битное представление было выбрано с учетом того, что канат – это группа 32 нитей. Кроме этого, нити одного каната обращаются к одной и той же области памяти, что позволяет кешировать эти данные. Это дает возможность максимизировать полезную нагрузку на один канат, обеспечив распределение данных, а также снизить затраты на операции чтения данных. Стоит отметить, что если вероятность p появления ребра больше чем $N/(16(N-1))$, то использование CCAM-матрицы будет потреблять меньшее количество памяти, чем формат CSR. Поскольку $\lim(N/(16(N-1))) = 0{,}062\,5$ при N, стремящихся к бесконечности, то при $N \geq 32$ пороговым значением вероятности p является приблизительно 6,25 %. Следовательно, при значении вероятности p больше чем 0,062 5 использование CCAM является более эффективным с точки зрения занимаемой памяти. При этом предполагается, что в CSR-формате для хранения индексов ненулевого элемента матрицы смежности используется 4 Б памяти – по 2 на каждый индекс, и каждый элемент CCAM-матрицы также хранится в 4 Б.

Тестирование алгоритма

Тестирование алгоритма проводилось на ориентированных графах при довольно высоких вероятностях появления ребра графа. Для генерации графов использовался алгоритм, принимающий в качестве параметров количество вершин N в графе и вероятность появления ребра графа p. Количество ребер в графе приблизительно равно $pN(N-1)$, что при высоких значениях вероятности p обеспечивает относительно большое количество ребер. Характеристики используемых для тестирования графов представлены в таблице.

Характеристики использованных графов

Наименование графа	Значение вероятности (p)	Количество вершин (N)	Количество ребер (E)	Количество элементов (N^2)
G16K-P05	0,05	16 384	13 421 102	268 435 456
G16K-P15	0,15	16 384	40 263 040	268 435 456
G16K-P32	0,32	16 384	85 879 327	268 435 456
G20K-P05	0,05	20 480	20 961 074	419 430 400
G20K-P15	0,15	20 480	62 919 715	419 430 400
G20K-P32	0,32	20 480	134 204 923	419 430 400
G24K-P05	0,05	24 576	30 196 005	603 979 776
G24K-P15	0,15	24 576	90 597 539	603 979 776
G24K-P32	0,32	24 576	193 241 105	603 979 776

В тестах применялись следующие представления графа: BASIC – классическое представление матрицы смежности; CSR – формат Compressed Sparse Row; CCAM – предлагаемая кодированная для CUDA матрица смежности. Алгоритмы, использующие эти структуры данных, следующие: CPU_BASIC – последовательный алгоритм, выполняемый на CPU на основе очередей и использующий BASIC-формат; GPU_BASIC – параллельный алгоритм на GPU, использующий BASIC-формат; GPU_CSR – параллельный алгоритм на GPU с CSR-форматом; GPU_CCAM – параллельный алгоритм на GPU, задействующий CCAM-формат. Тестирование осуществлялось для сравнения производительности, времени, затраченного для загрузки данных на GPU, требуемого объема памяти, достаточного для представления данных, а также степени равномерности загрузки блоков потоков и канатов. Для измерения производительности использовалась характеристика TEPS (Traversed Edge per Second), введенная в тесте Graph 500. В качестве аппаратной платформы применялись CPU Intel Core i5-2430M; RAM SO-DIMM DDR3 PC-10600; GPU nVidia GeForce GT 555M (Fermi 2.1), а для компиляции – программный пакет CUDA ToolKit 6.0. Замеры характеристик осуществлялись при помощи nVidia Visual Profiler.

На рис. 1 для сравнения представлена производительность описанных алгоритмов. Как и предполагалось, производительность предлагаемого алгоритма GPU_CCAM значительно выше сравниваемых аналогов. Подобные показатели достигаются благодаря учету особенностей архитектуры графических процессоров. Наиболее динамические характеристики у алгоритма GPU_CSR. Рост производительности при повышении значения вероятности p связан с увеличением равномерности нагрузки на используемые канаты, а также снижением ветвления нитей. Производительность GPU_CCAM является относительно стабильной и зависит в первую очередь от характеристик аппаратуры.

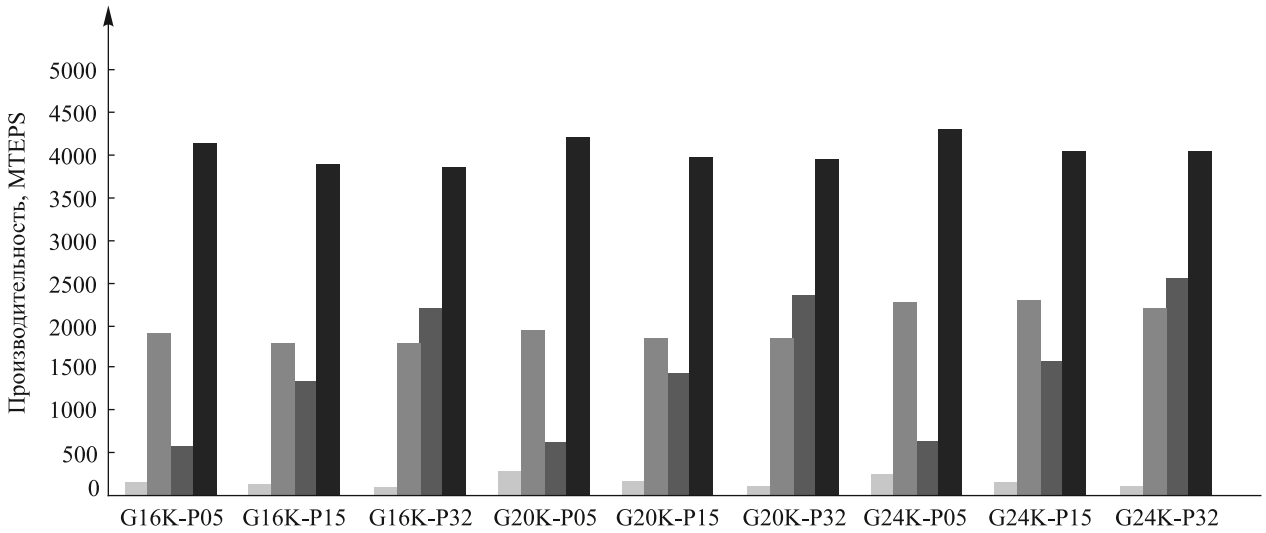

Рис. 1. Производительность алгоритмов. Здесь и далее:
– CPU_BASIC; – GPU_BASIC; – GPU_CSR; – GPU_CCAM

На рис. 2 приведены объемы памяти, используемые GPU при работе каждого из алгоритмов. Рассматривается только память, необходимая для хранения представления графа. Особый интерес вызывает сравнение объемов памяти, используемой алгоритмами GPU_CSR и GPU_CCAM, которое подтверждает высказанное в предыдущем разделе предположение.

Из рис. 3 следует, что при превышении порогового значения вероятности p время, затрачиваемое на копирование данных алгоритмом GPU_CSR, существенно возрастает относительно показателей GPU_CCAM. В худшем случае это приводит к тому, что время выполнения самого алгоритма может быть в 1,5 раза меньше, чем затрачиваемое на копирование данных. Таким образом, даже при условии, что производительность алгоритма GPU_CSR с увеличением вероятности p возрастает (см. рис. 1), общее время выполнения также увеличивается.

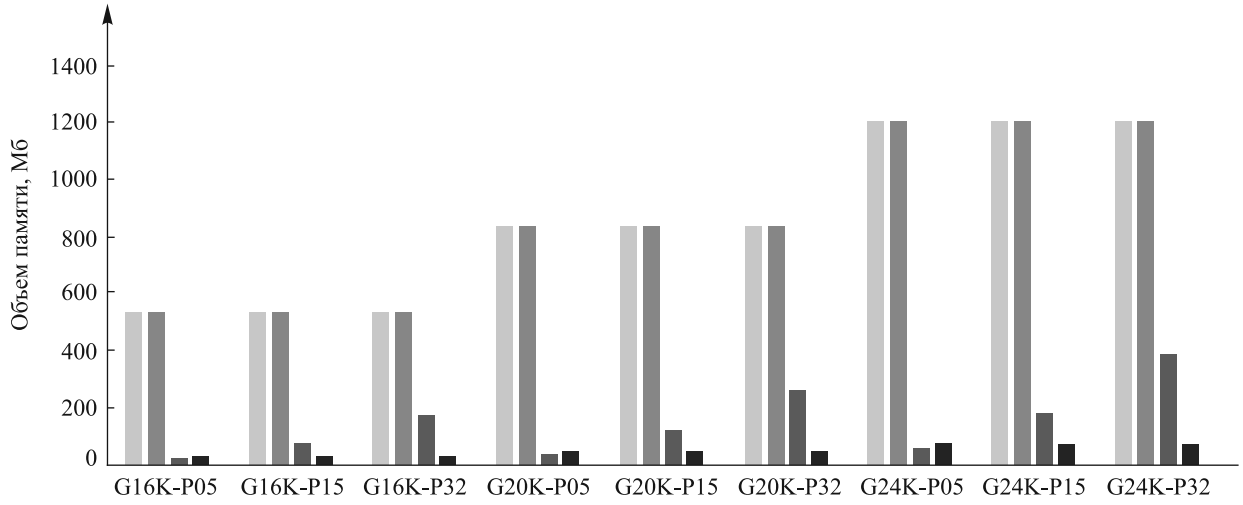

Рис. 2. Объемы используемой памяти

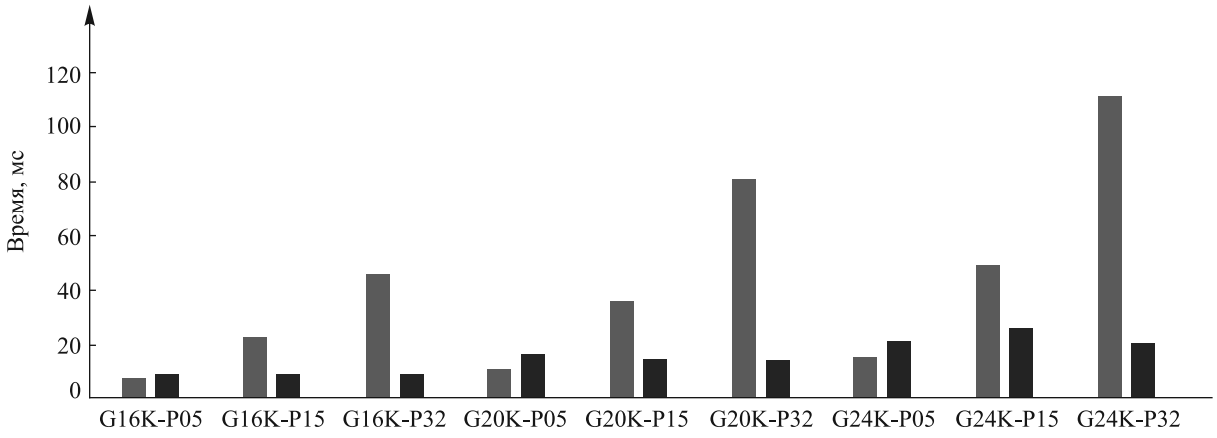

Рис. 3. Время, необходимое для загрузки данных на GPU

Кроме этого, в качестве подтверждения причин повышения производительности алгоритма GPU_CCAM приведем средние показатели загрузки канатов. Так, для GPU_BASIC это значение составляет 30,73 %, что крайне мало и приводит к существенному снижению его производительности. Для алгоритма GPU_CSR оно составляет 64,8 % и, как следует ожидать, возрастает при увеличении значения p. Алгоритм GPU_CCAM, как и предполагалось, использует 96,7 % ресурсов графического процессора, что существенно повышает его производительность при описанных выше условиях.

Таким образом, ускорение параллельного алгоритма поиска в ширину на графах при вычислениях на GPU достигается кодированием матрицы смежности, ориентированным на программно-аппаратную архитектуру CUDA. Использование подобного способа представления данных оправдано для случайных графов с вероятностью появления ребра $p > 0{,}062\,5$. Данный способ позволяет снизить накладные

расходы на операции чтения/записи из глобальной памяти, необходимый объем памяти для представления графа, а также существенно сократить время, затрачиваемое на загрузку данных, которое в других случаях может превышать время выполнения самого алгоритма. Несмотря на избыточность вычислений, при высоких значениях вероятности p полезная нагрузка на GPU остается на уровне 96,7 %.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Graph 500 Benchmark 1 («Search») [Electronic resource]. URL: http://www.graph500.org/specifications (date of access: 15.04.2015).

2. *Черноскутов М. А., Ермаков Д. Г.* Балансировка нагрузки в ГПУ-реализации поиска в ширину на графе // Вычисл. методы и программирование. 2013. Т. 14. С. 54–62.

3. *Lijuan Luo, Martin Wong, Wen-mei Hwu.* An effective GPU implementation of breadth-first search // Proc. of the 47th Design Automation Conf. (Anaheim, 13–18 June, 2010). New York, 2010. P. 52–55.

4. *Takaaki Hiragushi, Daisuke Takahashi.* Efficient Hybrid Breadth-First Search on GPUs. Algorithms and Architectures for Parallel Processing // Lecture Notes in Computer Science. 2013. Vol. 8286. P. 40–50.

5. *Koji Ueno, Toyotaro Suzumura.* Parallel distributed breadth first search on gpu // IEEE Intern. Conf. on High Performance Computing (Bangalore, 18–21 Dec., 2013). Bangalore, 2013. P. 314–323.

6. *Chun-Yuan Lin.* Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme // IEEE Transactions on Computers. 2003. Vol. 52, № 12. P. 1640–1646.

7. *Harish P., Narayanan P. J.* Accelerating large graph algorithms on the GPU using CUDA // Proc. of the 14th Intern. Conf. on High Performance Computing (Cidade de Goa, India, 18–21 Dec., 2006). Berlin ; Heidelberg, 2007. P. 197–208.

8. *Merrill D., Garland M., Grimshaw A.* Scalable GPU graph traversal // Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New Orleans, 25–29 Febr., 2012). New York, 2012. P. 117–128.

9. *Singla G.* New approach for graph algorithms on GPU using CUDA // Intern. J. of Computer Applications. 2013. Vol. 72, № 18. P. 38–42.

10. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths // IEEE 28th Intern. Parallel and Distributed Processing Symposium (Phoenix, 19–23 May, 2014). Washington, 2014. P. 349–359.

11. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Santa Clara, 2009.

12. *Nicholas Wilt.* The CUDA Handbook. Addison-Wesley, 2013.

13. Accelerating CUDA graph algorithms at maximum warp / Hong Sungpack [et al.] // Proc. of the 16th ACM Symposium on Principles and Practice of Parallel Programming (San Antonio, 12–16 Febr., 2011). New York, 2011. P. 267–276.

Поступила в редакцию 05.11.2014.

Андрей Андреевич Козик – аспирант кафедры технологий программирования факультета прикладной математики и информатики БГУ. Научный руководитель – А. П. Побегайло.

Александр Павлович Побегайло – кандидат технических наук, доцент кафедры технологий программирования факультета прикладной математики и информатики БГУ.

УДК 681.327.12.001.362

В. В. КРАСНОПРОШИН, А. И. КУЗЬМИЧ

МОДЕЛЬ СЦЕНЫ И АДАПТИВНЫЙ АЛГОРИТМ МОНИТОРИНГА ТРАНСПОРТНЫХ СРЕДСТВ В УСЛОВИЯХ НЕОПРЕДЕЛЕННОСТИ

Решается задача унификации процессов мониторинга транспортных средств, используемых в логистических проектах, с целью минимизации расходов на их эксплуатацию. Предложены модель сцены реализации проекта, включающая акторов (участников) с естественным и искусственным интеллектом, и алгоритм оценки ее состояния. Модель ориентирована на замену людей системами с элементами искусственного интеллекта без изменения общей структуры и типов коммуникаций. Разработан адаптивный алгоритм оценки транспортных средств, обеспечивающий синтез релевантного решения актора, компетентность которого соответствует уровню неопределенности возникшей проблемной ситуации. Алгоритм инвариантен количеству диагностических параметров, что исключает проблемы «размерности» и «узости диапазона наблюдения», характерные для существующих систем мониторинга. Работоспособность алгоритмов показана на примерах решения практических задач.

Ключевые слова: мониторинг транспортных средств; принятие решений; распознавание образов; имитационное моделирование.

Vehicles monitoring process unification problem in order to minimize their operation cost in logistics projects is being solved. The project implementation model which includes actors with natural and artificial intelligence and ability to assess its condition is suggested. The model is originally designed to replace human with systems with artificial intelligence elements without changing the overall structure and types of communications. An adaptive algorithm for estimating the vehicle, providing relevant decisions synthesis by the actor is being developed. Its competence is appropriate to the level of problem situation uncertainty. The algorithm is invariant to the number of diagnostic parameters, which eliminates «dimension» and «narrow observation range» problems that are typical for existing monitoring systems. The algorithm efficiency is demonstrated by examples of solving practical problems.

Key words: vehicle monitoring; decision-making; pattern recognition; simulation.