УДК 591.68

РЕШЕНИЕ ЗАДАЧ НА ГРАФАХ С ПОМОЩЬЮ STAR-МАШИНЫ, РЕАЛИЗУЕМОЙ НА ГРАФИЧЕСКИХ УСКОРИТЕЛЯХ

Т. В. Снытникова, А. Ш. Непомнящая

Институт вычислительной математики и математической геофизики СО РАН, г. Новосибирск, Россия

Для решения многих задач на графах построены эффективные алгоритмы на ассоциативных параллельных процессорах. Но на данный момент нет широко используемых ассоциативных архитектур. Однако с развитием графических ускорителей появилась возможность реализовывать ассоциативные параллельные модели без существенной потери эффективности вычислений, что позволяет применять ассоциативные алгоритмы на практике. Представляется реализация абстрактной модели ассоциативной параллельной обработки данных (STAR-машина) на графических ускорителях с помощью технологии CUDA. Измеряется производительность реализации и показывается её эффективность для решения задач на графах на примере алгоритма Уоршалла нахождения транзитивного замыкания ориентированного графа. На графе с 5000 вершин последовательный алгоритм Уоршалла выполнялся за 884,622 с, ассоциативная параллельная версия — за 64,454 с (ускорение в 13 раз), а ассоциативная параллельная версия, адаптированная под GPU, — за 0,372 с (ускорение в 2 378 раз).

Ключевые слова: *ассоциативный параллельный процессор, вертикальная обработка данных, SIMD, GPU, ориентированный граф, транзитивное замыкание.*

DOI 10.17223/20710410/33/9

SOLUTION OF GRAPH PROBLEMS BY MEANS OF THE STAR-MACHINE BEING IMPLEMENTED ON GPUS

T. V. Snytnikova, A. Sh. Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia

E-mail: snytav@yahoo.com, anep@ssd.sscc.ru

Efficient algorithms have been developed for the solution of many graph problems. The solution is performed with associative parallel processors. At present there are no widely used associative architectures. Nevertheless, the recent development of the GPUs has made it possible to implement the associative parallel models with no significant efficiency loss. It enables to use the associative algorithms in practice. The implementation of the abstract model of the associative parallel data processor is given (the STAR-machine) with the GPUs by means of the CUDA technology. The performance is being analysed as well as the efficiency of the solution of the graph problems. The test case is the Warshall algorithm for finding the transitive closure of a directed graph. A graph with 5000 nodes was processed by the sequential Warshall algorithm for 884,622 s, by the associative parallel version for 64,454 s (speed-up is 13 times) and by the GPU-adopted associative parallel version for 0,372 s (speed-up is 2 378 times).

Keywords: *associative parallel processor, vertical data processing, SIMD, GPU, directed graph, adjacency matrix, transitive closure.*

Введение

Ассоциативные вычисления используют совершенно другой метод хранения, извлечения данных и их обработки по сравнению с общепринятой последовательной технологией. Ассоциативные (контекстно-адресуемые) параллельные процессоры (АПП) типа SIMD принадлежат к классу мелкозернистых параллельных систем с простейшими процессорными элементами и последовательно-поразрядной (вертикальной) обработкой информации. Такие системы называют ещё системами вертикальной обработки. В них файл входных данных физически загружается в матричную (двумерную) память, в которой каждая запись файла занимает отдельную строку и обрабатывается отдельным процессорным элементом. Основным преимуществом таких архитектур является реализация интеллектуальной памяти. Такие системы ориентированы, в первую очередь, на решение задач нечисловой обработки. Сюда относятся теория графов, реляционные базы данных, базы знаний, экспертные системы, обработка сейсмических данных и обработка изображений. Основными достоинствами АПП являются параллелизм по данным на базовом уровне, массовый параллельный поиск по содержимому памяти, двумерные таблицы в качестве основной структуры данных и обработка неупорядоченных данных [1]. Базовые операции поиска ($=$, $<$, $\leqslant$, $>$, $\geqslant$, min, max) и арифметические операции выполняются за время, пропорциональное числу битовых столбцов в таблице, а не числу её строк. К этому классу параллельных архитектур относятся хорошо известная система STARAN [2, 3] и современные компьютерные архитектуры Rutgers CAM, ASPRO, IXM2 [4].

На основе процессора типа STARAN разработаны некоторые модели ассоциативной параллельной обработки данных. В работе [5] сравниваются три модели ассоциативной обработки данных: STAR-машина [6], модель Поттера ASC [1, 7] и ортогональная машина. Отметим, что при разработке STAR-машины также учитывались основные свойства отечественного АПП ЕС-1020.

Заметим, что, с одной стороны, АПП позволяют строить эффективные алгоритмы для различных приложений. С другой стороны, пока нет широко используемых ассоциативных архитектур, позволяющих реализовывать эти алгоритмы. Однако в последнее время широко распространены относительно недорогие графические ускорители, которые используются для различных приложений. Они относятся к типу SIMD, поэтому хорошо подходят для реализации STAR-машины.

Наша цель — построить эффективную реализацию STAR-машины на графических ускорителях, используя технологию CUDA. Это позволит использовать на практике ассоциативные параллельные алгоритмы [8 – 22]. Для этого необходимо построить эффективную реализацию базовых операций языка Star на графическом ускорителе. Отметим, что библиотека стандартных процедур языка Star будет реализована как отдельный модуль. В работе [23] такой подход обсуждается для реализации ассоциативной параллельной модели вычислений MASC, которая является расширением модели ASC.

В п. 1 описана STAR-машина, приведены типы данных и базовые операции языка Star, перечислены базовые ассоциативные алгоритмы из библиотеки стандартных процедур и приведены группы ассоциативных алгоритмов (Star-алгоритмов) для решения задач на графах. Приводится ассоциативная параллельная версия алгоритма Уоршалла нахождения транзитивного замыкания ориентированного графа. В п. 2 описана ре-

ализация типов данных и базовых операций STAR-машины на языке CUDA C; показана адаптация Star-алгоритма, учитывающая архитектурные отличия STAR-машины и графического ускорителя. В п. 3 дан анализ производительности предложенной реализации STAR-машины на графических ускорителях. Сначала на примере выполнения Star-алгоритма Уоршалла оценивается время работы реализации базовых операций на графическом ускорителе, далее — размер данных, необходимых для хранения графа в форматах STAR-машины. Потом сравнивается время выполнения последовательного алгоритма Уоршалла со временем выполнения Star-алгоритмов, представленных в работе.

1. Модель ассоциативной параллельной обработки данных

STAR-машина — абстрактная модель типа SIMD (Single Instruction Multiple Data) с вертикальной обработкой данных — была представлена в работах [6, 8]. STAR-машина состоит из следующих частей (рис. 1):

— последовательного устройства управления (ПУУ), в котором записаны программа и скалярные константы;
— матричной памяти;
— устройства ассоциативной обработки (УАО), состоящего из p одноразрядных процессорных элементов (ПЭ).

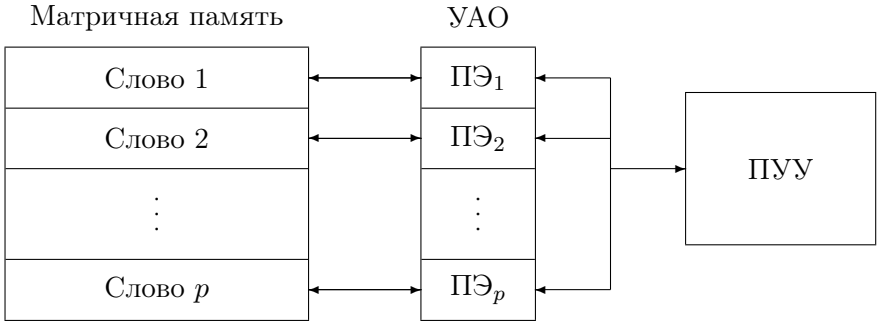

Рис. 1. Модель STAR-машины

Входные данные, записанные в двоичном коде, помещаются в матричную память в виде двумерных таблиц, причём каждая единица данных хранится в отдельной строке и обрабатывается отдельным ПЭ. Строки каждой таблицы нумеруются сверху вниз, а столбцы — слева направо (рис. 2). В матричную память можно загружать несколько таблиц.

Устройство ассоциативной обработки (рис. 3) представляется в виде совокупности h вертикальных одноразрядных регистров длины p. Вертикальный регистр можно представлять как массив, состоящий из одного столбца. Обработка информации происходит следующим образом: из определённой таблицы в определённом порядке извлекаются её одноразрядные столбцы и помещаются в вертикальные регистры устройства ассоциативной обработки, в котором выполняются побитовые операции. Результат выполнения любой операции записывается либо в некоторый регистр, либо в определённый столбец обрабатываемой таблицы, либо в матричную память.

Поясним содержательно работу STAR-машины. Каждый ПЭ может быть либо активным, либо пассивным в зависимости от своего содержимого. Перед выполнением любой программы задаются активные ПЭ, т. е. указываются позиции тех строк заданной таблицы, которые надо анализировать. Каждая команда программы одновременно

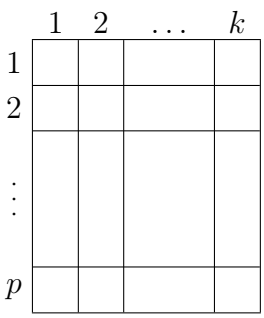

Рис. 2. Матричная память

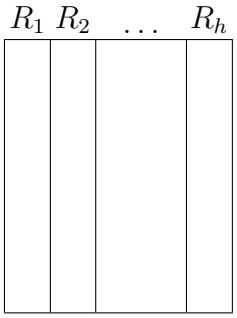

Рис. 3. Устройство ассоциативной обработки

подаётся на все ПЭ, но только активные ПЭ её выполняют. В процессе выполнения некоторых команд некоторые активные ПЭ могут стать пассивными. Результат выполнения программы определяется с помощью активных ПЭ. В качестве примера приведём идею ассоциативного параллельного алгоритма для нахождения строк заданной таблицы T, которые совпадают с заданным образцом v (рис. 4).

v

1	0	1	1

T			
1	0	1	0
0	0	1	1
1	0	1	1
1	0	1	1
1	0	1	1
1	0	1	0

X
1
0
1
0
1
1

Z
0
0
1
0
1
0

Рис. 4. Нахождение строк таблицы T, совпадающих с образцом v

Сначала с помощью управляющего битового столбца X указываются позиции анализируемых строк таблицы T или активных ПЭ. После считывания k-го бита образца v активными являются ПЭ, которые хранят строки, начинающиеся подцепочкой длины k образца v. После завершения программы активные ПЭ указывают позиции строк таблицы T, которые совпадают с образцом v. На рис. 4 после считывания последнего бита образца v первый и шестой процессорные элементы станут пассивными. Результат указывается с помощью битового столбца Z. В работе [8] приведена процедура MATCH, которая реализует ассоциативный алгоритм поиска строк, совпадающих с образцом.

Для STAR-машины в [6, 8] разработан язык высокого уровня Star, похожий на язык Pascal. Чтобы моделировать обработку информации в матричной памяти, в языке Star имеются три типа данных: `slice` (далее слайс), `word` (слово) и `table` (таблица). С помощью переменной типа `slice` моделируется доступ к таблице по столбцам, а с помощью переменной типа `word` — доступ по строкам. С каждой переменной типа `table` ассоциируется бинарная таблица из n строк и k столбцов, где $n \leqslant p$.

Приведём основные операции для переменной типа `slice`. Пусть X и Y — слайсы, i — целочисленная переменная. Тогда операции имеют следующий вид:

— $SET(Y)$ записывает в слайс Y все единицы;
— $CLR(Y)$ записывает в слайс Y все нули;
— $Y(i)$ выделяет i-ю компоненту в слайсе Y $(1 \leqslant i \leqslant n)$;

— $FND(Y)$ выдаёт порядковый номер i позиции первой (старшей) единицы в слайсе Y, где $i \geqslant 0$;
— $STEP(Y)$ выдаёт такой же результат, как и операция $FND(Y)$, и затем обнуляет старшую единицу;
— $NUMB(Y)$ выдаёт число компонент 1 в слайсе Y;
— $CONVERT(Y)$ выдаёт слово, i-й бит которого совпадает с $Y(i)$;
— $FRST(Y)$ сохраняет первую (старшую) единицу в слайсе Y и обнуляет остальные компоненты.

Общепринятым способом вводятся предикат $SOME(Y)$, а также следующие побитовые логические операции: $X \text{ and } Y$, $X \text{ or } Y$, $\text{not } X$, $X \text{ xor } Y$.

Замечание 1. Операция $STEP(Y)$ используется для построения неравномерных циклов.

Для переменных типа `word` выполняются те же операции, что и для переменных типа `slice`. Кроме этого, для слов определены следующие операции:

— $TRIM(i, j, w)$ возвращает подстроку слова w с i-го по j-й биты, где $1 \leqslant i < j \leqslant |w|$;
— $REP(i, j, v, w)$ замещает подстроку $w(i)w(i+1)\ldots w(j)$ слова w словом v, где $|v| = j - i + 1$ и $1 \leqslant i < j < |w|$.

Следуя Дж. Поттеру [1], для двух переменных типа `word` определены следующие предикаты: $EQ(v, w)$, $NOTEQ(v, w)$, $LESS(v, w)$, $LESSEQ(v, w)$, $GREAT(v, w)$ и $GREATEQ(v, w)$, а также арифметические функции $ADD(v, w)$ и $SUBT(v, w)$.

Для переменных типа `table` определены следующие две операции:

— $COL(i, T)$ обеспечивает доступ к i-му столбцу таблицы T для выполнения чтения и записи;
— $ROW(i, T)$ обеспечивает доступ к i-й строке таблицы T для выполнения чтения и записи.

1.1. Базовые Star-алгоритмы

Для STAR-машины разработаны различные базовые ассоциативные алгоритмы как для нечисловой, так и для числовой обработки данных. В работе [8] они приведены в виде процедур на языке Star. Процедуры используют управляющий слайс, в котором единицами отмечены *позиции* обрабатываемых строк. К алгоритмам нечисловой обработки, в частности, относятся:

— поиск строк таблицы, совпадающих с образцом;
— сравнение строк с образцом (больше, меньше, равно);
— поиск минимальной строки;
— поиск максимальной строки;
— сравнение таблиц построчно.

К алгоритмам числовой обработки, в частности, относятся:

— прибавление слова w к строкам таблицы;
— построчное сложение таблиц;
— вычитание слова w из строк таблицы;
— построчное вычитание одной таблицы из другой;
— нахождение позиций совпадающих строк.

Эти алгоритмы выполняются на STAR-машине за время, пропорциональное количеству битовых столбцов таблицы T.

1.2. Star-алгоритмы на графах

Для STAR-машины построены как новые ассоциативные параллельные алгоритмы на графах, так и ассоциативные версии ряда известных последовательных алгоритмов. В работах [9, 10] приводятся специальные конструкции и методы ассоциативной обработки данных, которые были разработаны для построения ассоциативных параллельных алгоритмов на графах.

Для *ориентированных* графов разработаны ассоциативные версии алгоритма Уоршалла для нахождения транзитивного замыкания [11], алгоритма Флойда для нахождения кратчайших расстояний между любыми парами вершин [11], алгоритмов Дейкстры [12] и Беллмана — Форда [13] для нахождения кратчайших расстояний и кратчайших путей от заданной вершины до остальных вершин графа, алгоритма Эдмондса [14] для нахождения оптимального ветвления. Для *неориентированных* графов построены ассоциативные версии алгоритмов Краскала и Прима — Дейкстры для нахождения минимального остовного дерева (MST) [15] и алгоритма Габова [16] для нахождения наименьшего остовного дерева с ограничением на степень одной из вершин.

Перечислим группу *динамических алгоритмов* на графах, реализованных на STAR-машине. В [17] построены два ассоциативных параллельных алгоритма для динамической обработки MST после удаления и после добавления одного ребра к неориентированному графу. В [18, 19] построены ассоциативные параллельные алгоритмы для преобразования MST после удаления и после добавления к графу одной вершины вместе с инцидентными рёбрами. В [20] приведены ассоциативные версии алгоритмов Итальяно для динамической обработки транзитивного замыкания ориентированного графа после добавления и после удаления одной дуги. В [21, 22] построены ассоциативные версии алгоритмов Рамалингама для динамической обработки подграфа кратчайших путей с одним стоком после удаления из графа одной дуги и после добавления к графу новой дуги.

1.3. Star-версии алгоритма Уоршалла

Для оценки времени работы реализации базовых операций STAR-машины будем использовать ассоциативную версию алгоритма Уоршалла. При его реализации используются только базовые операции языка Star без привлечения базовых алгоритмов. Описание алгоритма, доказательство его корректности и оценка сложности приведены в [11].

Процедура получает на вход матрицу смежности ориентированного невзвешенного графа с n вершинами и возвращает матрицу путей для транзитивного замыкания графа (листинг 1).

```
 1 proc WARSHALL(n:integer; var P: table);
 2 /*Здесь n - число вершин графа.*/
 3 var X:slice(P); v,w: word; i,k: integer;
 4 begin for k:=1 to n do
 5    begin
 6       X:=COL(k,P);
 7       w:=ROW(k,P);
 8       while SOME(X) do
 9       begin
10         i:=STEP(X);
11         v:=ROW(i,P);
12         v=v or w;
```

```
13          ROW(i,P):=v;
14        end;
15    end;
16 end;
```

Листинг 1. Star-версия алгоритма Уоршалла

Модель STAR-машины предполагает организацию доступа как к строкам, так и к столбцам за единицу времени. Но при реализации STAR-машины на графических ускорителях доступ к столбцу осуществляется быстрее, чем к строке. Поэтому модифицируем процедуру WARSHALL так, чтобы минимизировать обращение к строкам (листинг 2).

```
1  proc WARSHALL-C(n:integer; var P: table);
2  var x:word; V, W: Slice; i,k: integer;
3  begin for k:=1 to n do
4     Begin
5        x:=ROW(k,P);
6        W:=COL(k,P);
7        i=STEP(x);
8        while i>0 do
9        Begin
10          V:=COL(i,P);
11          V=V or W;
12          COL(i,P):=V;
13          i:=STEP(x);
14       end;
15    end;
16 end;
```

Листинг 2. Модифицированная Star-версия алгоритма Уоршалла

Рис. 5 иллюстрирует выполнение одной итерации ($k = 2$) процедур WARSHALL и WARSHALL-C. В процедуре WARSHALL в слайс X записан второй столбец, а в слово w — вторая строка матрицы P^1. В цикле (8–14) строки (выделены курсивом), отмеченные единицами в слайсе X, объединяются со словом w. Значения элементов строк, выделенных только курсивом, не изменяются, потому что соответствующие им элементы слова w равны нулю. Поэтому изменить значения могли только те элементы строк, которые находятся в столбцах, отмеченных единицами в слове w. Такие элементы отмечены жирным шрифтом. Аналогично в процедуре WARSHALL-C в слайс W записан второй столбец, а в слово x — вторая строка матрицы P^1. В цикле (8–14) столбцы (выделены курсивом), отмеченные единицами в строке x, объединяются со слайсом W. На этой итерации могли измениться значения только тех элементов столбцов, которые находятся в строках, отмеченных единицами в слайсе W. Эти элементы отмечены жирным шрифтом и их позиции совпадают с позициями элементов, которые могли измениться в процедуре WARSHALL. В утверждении 1 приведено формальное доказательство.

Утверждение 1. Процедуры WARSHALL и WARSHALL-C различаются порядком обхода, но вычисляют одну и ту же логическую функцию.

Доказательство. Для доказательства утверждения 1 восстановим вычисляемую логическую функцию для каждой процедуры.

X	$w=$	**1**	0	**1**	0	0	0
0		0	0	1	0	0	0
0		1	0	1	0	0	0
1		***1***	*1*	***1***	*0*	*0*	*1*
1		***1***	*1*	***1***	*0*	*0*	*0*
0		0	0	1	0	0	0
0		0	0	0	1	1	0

WARSHALL

W	$x=$	***1***	0	***1***	0	0	0
0		*0*	0	*1*	0	0	0
0		*1*	0	*1*	0	0	0
1		***1***	1	***1***	0	0	1
1		***1***	1	***1***	0	0	0
0		*0*	0	*1*	0	0	0
0		*0*	0	*0*	1	1	0

WARSHALL-C

Рис. 5. Результат выполнения итерации ($k = 2$) для Star-версий алгоритма Уоршалла. Обрабатываемые элементы матриц выделены курсивом, элементы, изменяющие свои значения, — жирным шрифтом

Заметим, что $COl(k, P)$ соответствует $\forall j\, P(j, k)$, а $ROW(k, P)$ — $\forall j\, P(k, j)$. Тогда в этих терминах для каждого фиксированного k тело цикла (5–15) процедуры WARSHALL записывается в следующем виде:

$$\forall j\, \forall i\, P^k(i,j) = \begin{cases} P^{k-1}(i,j), & \text{если } P^{k-1}(i,k) = 0, \\ P^{k-1}(i,j) \text{ or } P^{k-1}(k,j), & \text{если } P^{k-1}(i,k) = 1, \end{cases}$$

или

$$\forall j\, \forall i\, P^k(i,j) = P^{k-1}(i,j) \text{ or } \left(P^{k-1}(i,k) \text{ and } P^{k-1}(k,j)\right). \tag{1}$$

Для тела цикла (4–15) процедуры WARSHALL-C имеем

$$\forall j\, \forall i\, P^k(j,i) = \begin{cases} P^{k-1}(j,i), & \text{если } P^{k-1}(k,i) = 0, \\ P^{k-1}(j,i) \text{ or } P^{k-1}(j,k), & \text{если } P^{k-1}(k,i) = 1, \end{cases}$$

или

$$\forall j\, \forall i\, P^k(j,i) = P^{k-1}(j,i) \text{ or } \left(P^{k-1}(j,k) \text{ and } P^{k-1}(k,i)\right). \tag{2}$$

Формулы (1) и (2) совпадают на каждой итерации $k = 1, \ldots, n$. ■

Замечание 2. Теоретическая оценка обеих ассоциативных версий совпадает (не более $O(n^2)$, фактически $O(j)$, где j — количество ненулевых элементов в результирующей матрице, $n \leqslant j \leqslant n^2$). В процедуре WARSHALL операция COL вызывается n раз, а операция ROW — $(n+j)$ раз на чтение и j раз на запись. В процедуре WARSHALL-C операция ROW вызывается n раз на чтение, а операция COL — $(n + j)$ раз на чтение и j раз на запись.

2. Реализация базовых операций языка Star на графических ускорителях

2.1. Представление типов данных

Как отмечено выше, для доступа к матричной памяти используются типы данных `word`, `slice` и `table`.

Так как CUDA C является расширением языка C++, для реализации типов данных используются одноимённые классы. Заметим, что класс `Slice` используется и для типа `word`. Приведём описание этих классов.

```
1 class Slice{
2 // основная память CPU
3  /* length - длина слайса в битах;
```

```
4      N - количество 64-х разрядных элементов для хранения
      слайса */
5   unsigned int length,N;
6  // глобальная память GPU
7  /* *d_v хранит адрес первого элемента массива длины $N$ */
8   Unsigned long long int *d_v;
9  ...}
```

Здесь переменная `N` равна числу 64-разрядных переменных, необходимых для хранения битового столбца длины `length`. Битовый столбец хранится в массиве `d_v` из `N` переменных типа `unsigned long long int`, расположенном в глобальной памяти GPU.

Класс `Table` содержит массив из объектов класса `Slice` и несколько указателей.

```
1  class Table{
2  /*length - число строк, size - число столбцов */
3      unigned int length,size;
4      Slice table[size];
5      LongPointer *slice_device_pointer_table;
6  // глобальная память GPU
7      LongPointer *d_table, *d_slice_device_pointer_table;
8  ...}
```

Указатель на GPU `*d_table` хранит адрес первого элемента массива `length`·`N` элементов типа `unsigned long long int`, расположенного в глобальной памяти GPU. Указатели `*slice_device_pointer_table` на CPU и `*d_slice_device_pointer_table` на GPU хранят адреса первых элементов массивов из `size` указателей на столбцы `d_table`. С помощью такой системы указателей осуществляется доступ как целиком к таблице, так и к каждому столбцу в отдельности.

2.2. Базовые операции для типа `slice`

Базовые операции над переменными типа `slice` реализованы как одноименные методы класса `Slice`. В свою очередь, каждый метод запускает функцию на GPU, исполняющую операцию. Таким образом, все вычисления производятся на графическом ускорителе параллельно.

Ниже приводятся реализации базовых операций. Отметим, что при операции присваивания указателей на объект класса возможна неоднозначность. Чтобы её избежать, побитовые логические операции реализованы как совмещенные с присваиванием, аналогично подобным операциям языка C++, т. е. $Z := X \text{ and } Y$ записывается как $Z = X$; $Z.AND(Y)$. Операция and реализована следующим образом:

```
1  void Slice::AND(Slice *Y)
2  {
3      and_long_values<<<N,1>>>(d_v,Y->d_v);
4  }
5  __global__ void and_long_values( *d_v, *d_v1)
6  {
7      d_v[blockIdx.x] &= d_v1[blockIdx.x];
8  }
```

Операции $SET(X)$, $CLR(X)$ и остальные побитовые логические операции реализованы аналогично. Заметим, что каждый из N блоков вычисляет свой элемент массива

независимо от других. Поэтому время выполнения этих операций не зависит от длины битового столбца `slice`.

Операция $FND(X)$ выполняется в два этапа. На первом этапе с помощью глобальной процедуры `find` массив `d_v[N]` за несколько шагов сворачивается до одной переменной типа `unsigned long long int`. На каждом шаге массив сворачивается по следующему правилу:

— если `dv[level,i]`$\neq 0$, то в i-й бит элемента массива `dv[level+1]` записывается 1;
— если `dv[level,i]`$= 0$ или $i > N$, то в i-й бит элемента массива `dv[level+1]` записывается 0.

Таким образом, за шаг свёртки длина массива `dv[level+1]` меньше длины массива `dv[level]` в 64 раза и `dv[level+1]` — битовая маска текущего массива `dv[level]`.

На втором этапе глобальная процедура `first_backward` вычисляет результат, разворачивая свёртку (листинг 3).

```
1  __global__ void first_backward(LongPointer *d_v, int *
       d_first_non_zero, int level)
2  {
3       int f[LEVELS];
4       unsigned long long int *dvl,u;
5       char lprt[100];
6       f[level+1] = 1;
7       while(level >= 0)
8       {
9           dvl = d_v[level];
10          int index = f[level+1]-1;
11          u=dvl[index];
12          f[level]=__ffsll(u)+index*SIZE_OF_LONG_INT;
13          level--;
14      }
15      *d_first_non_zero = f[0];
16 }
```

Листинг 3. Процедура `first_backward`

Время выполнения операции $FND(X)$ оценивается как $O(\log_{64}(N))$, где N — длина массива данных слайса.

Операция $STEP(X)$ и предикат $SOME(X)$ реализованы следующим образом. Пусть $i = FND(X)$ и $i > 0$. Тогда в операции $STEP(X)$ i-й бит обнуляется, а в качестве результата возвращается число i. Предикат $SOME(X)$ возвращает `true`. Операция $X = FRST(Y)$ реализуется следующим образом: $CLR(X)$; $i := FND(Y)$; $X(i) := 1$.

Операция $TRIM(i, j, w)$:

```
1  Slice::TRIM(int i, int j, Slice *w)
2  {
3     trim_long_values<<<N,1>>>(d_v,i,j,w->d_v);
4  }
5  __global__ void trim_long_values(*d_v, int i, int j, *d_v1)
6  {   unsigned long long int head, tail;
7      int k, n, index;
8      index=blockIdx.x;
```

```
 9 // сдвиг по битам
10     k = i % SIZE_OF_LONG_INT -1;
11 // сдвиг по элементам
12     n=(k==0)?(i/SIZE_OF_LONG_INT-1):(i/SIZE_OF_LONG_INT);
13     head = d_v1[n+index]>>k;
14     tail = d_v1[n+index+1]<<(SIZE_OF_LONG_INT-k);
15     head =head|tail;
16     d_v[index] = head;
```

Время выполнения операции $TRIM$ не зависит от длины битового столбца `slice`.

Отметим, что операция $CONVERT$ не нуждается в реализации, поскольку класс `Slice` используется для представления как слайсов, так и слов.

Замечание 3. Базовые операции языка Star над слайсами выполняются на графическом ускорителе или за константное время, или за время $\mathrm{O}(\log_{64}(N))$.

2.3. Базовые операции для типа `table`

Операции над типом `table` обеспечивают доступ к столбцам и строкам таблицы для выполнения чтения и записи.

Операция $COL(i, T)$ выполняется вызовом метода `T.col(i)`:

```
Slice * Table::col(int i) {return &(table[i-1]);}
```

Он обеспечивает доступ к столбцу как на чтение, так и на запись.

Для реализации операции $ROW(i, T)$ используются два метода класса `Table`. Метод `SetRow(Slice *s, int i)` записывает слово s в i-ю строку таблицы T. Метод `GetRow(Slice *s, int i)` записывает i-ю строку таблицы в слово s. Процедуры вычисляются на `s.N` блоках по 64 нити, где `s.N` — число битовых элементов в слове s. Поэтому каждый блок производит вычисления над своим элементом слова, а нити — над своим столбцом.

Напомним, что переменная T типа `table` представляется в глобальной памяти графического ускорителя массивом слайсов. Слайсы, в свою очередь, представляются массивами 64-разрядных элементов. Поэтому для доступа к j-му биту i-й строки необходимы следующие индексы:

- `n_col`$= \lceil i/64 \rceil$ — номер элемента массива в столбце;
- `b_col`$= i \bmod 64$ — номер бита в этом элементе;
- `n_row = blockIdx.x` — номер элемента в слове (равен номеру блока);
- `b_row = threadIdx.x` — номер бита в элементе слова (равен номеру нити в блоке);
- `k`$= 64 \cdot$`blockIdx.x + threadIdx.x` — номер столбца.

В качестве вспомогательных переменных используется массив `tmp[64]` в разделяемой памяти графического ускорителя (у каждого блока свой экземпляр массива, к которому имеют доступ все нити блока) и `col` — указатель на k-й столбец. Запись слова s в таблицу происходит следующим образом:

```
1 if (s(k)==1)
2 then
3 {
4    tmp[b_row]=1<<(b_col-1);
5    col[n_col]|=tmp[b_row];
6 }
7 else
8 {
```

```
 9     tmp[b_row]=~(1<<(b_col-1));
10     col[n_col]&=tmp[b_row]
11 }
```

При чтение строки таблицы элементы массива `tmp` для каждого блока вычисляются следующим образом:

```
1     bit=(col[n_col]>>(b_col-1))&1;
2     tmp[b_row]=bit<<(b_col-1);
3     s[n_row]=tmp[0]|...|tmp[63];
```

Для каждого блока вычисляются 64 элемента массива `tmp`. В элементе `tmp[bit_row-1]` бит с номером `b_row` совпадает с i-м битом k-го столбца, остальные биты равны 0. После этого все элементы массива `tmp` объединяются с помощью побитового ИЛИ в один элемент для каждого блока.

Замечание 4. Операции $COL(i,T)$ и $ROW(i,T)$ выполняются за константное время. Тем не менее $COL(i,T)$ выполняется быстрее $ROW(i,T)$.

2.4. Адаптация Star-алгорима Уоршалла к выполнению на графическом ускорителе

Модель STAR-машины предполагает параллельную обработку одного столбца или одной строки, но архитектура графических ускорителей даёт возможность одновременно обрабатывать несколько столбцов или строк. Поэтому некоторые Star-алгоритмы могут быть адаптированы к выполнению на графических ускорителях.

Так, в процедуре WARSHALL-C цикл (8–14) может выполняться параллельно по всем столбцам. Приведём листинг кода, используя в основном язык Star (листинг 4). Для записи процедуры потребуются из CUDA C++ переменная `blockIdx.x`, возвращающая номер исполняемого блока, и директивы `__global__` и `<<<n>>>`, указывающие, что процедура выполняется на n блоках.

```
1  __global__ Warshall_kernel(P,k)
2  Begin
3  // номер блока индексируется с 0, а номер столбца с 1
4      i:=blockIdx.X+1;
5      W:=COL(k,P);
6      V:=COL(i,P);
7      if(V(k)=1) then V=V or W;
8      COL(i,P):=V;
9  end;
10 proc WARSHALL-adapt(n:integer; var P: table);
11 var x:word; V, W: Slice; i,k: integer;
12 Begin
13   for k:=1 to n do
14     Warshall_kernel<<<n>>>(P,k);
15 end;
16 }
```

Листинг 4. Адаптированный Star-алгоритм Уоршалла

Такой подход позволяет снизить временну́ю сложность алгоритма до $O(n)$ и значительно уменьшить время его выполнения. Отметим, что не все блоки физически

выполняются одновременно, поэтому экспериментальное время счёта отличается от линейного. Но производители графических ускорителей постоянно наращивают количество ядер в них.

3. Анализ производительности

Приведём анализ времени работы базовых операций в зависимости от размера данных, оценим объём памяти, необходимый для представления графов в зависимости от числа вершин и ребер, и сравним время выполнения модифицированного Star-алгоритма Уоршалла и адаптированного Star-алгоритма со временем выполнения других реализаций этого алгоритма на графическом ускорителе. Все расчёты производились на графическом ускорителе NVidia Kepler k-40.

3.1. Время работы базовых операций

Для оценки времени работы базовых операций рассмотрим выполнение Star-алгоритма Уоршалла (см. листинг 1). Эта процедура использует следующие базовые операции языка Star: or, $SOME$, COL (на чтение), ROW (на чтение и на запись) и $STEP$. При этом вызываются следующие глобальные процедуры:

— `or_long_value` для выполнения операции or;
— `find` и `first_backward` для выполнения предиката $SOME$ и операции $STEP$ (вызывают операцию FND);
— `get_row` для выполнения операции ROW на чтение;
— `set_row` для выполнения операции ROW на запись.

Отметим, что операция COL передаёт указатель на столбец, поэтому ее время работы не измеряется.

Для определения времени работы операций Star-алгоритм Уоршалла выполнялся на графах с разным количеством вершин: 100, 1 000, 3 000 и 5 000 вершин. Результаты профилирования показаны в табл. 1. В первом столбце приведено название базовой операции, во втором — имя процедуры, выполняющейся на графическом ускорителе (реализация операции FND вызывает две глобальных процедуры). Отметим, что время выполнения процедуры включает в себя время запуска ядра, поэтому может существенно отличаться для значений порядка нескольких микросекунд. В табл. 1 приведено среднее время запуска; для 100 вершин в скобках показаны минимальное и максимальное время выполнения процедур.

Таблица 1

Профилирование базовых операций

Операция	Процедура	Время выполнения, мкс			
		100	1 000	3 000	5 000
ROW (чтение строки)	`get_row`	18,6 (18,5 – 19,7)	19,5	19,6	19,8
ROW (запись строки)	`set_row`	4,1 (2,9 – 5,6)	3,9	4,3	4,1
or (побитовое ИЛИ)	`or_long_value`	1,9 (1,9 – 3,0)	2,2	2,0	2,1
FND (номер старшей единицы в слайсе)	`find`	2,5 (2,3 – 2,8)	2,8	3,5	3,8
	`first_backward`	3,2 (3,1 – 4,0)	3,2	3,3	3,4

Замечание 5. Время выполнения базовых операций практически не зависит от размера данных, что соответствует теоретическим оценкам в п. 2: $O(\log_{64}(n))$ для реализации операций FND, $SOME$, $STEP$ и $FRST$ и константное время для реализации остальных операций.

3.2. Размер данных

Исследуем вопрос о необходимом размере памяти для хранения графа, который задаётся в разных форматах.

Для ассоциативных параллельных алгоритмов графы задаются в одном из двух форматов: матрицей весов (матрицей смежности, если граф невзвешенный) или списком рёбер и весов.

При использовании матрицы смежности для графа с n вершинами необходимо $8(n(\lceil n/64 \rceil + 1) + 1)$ байт глобальной памяти графического ускорителя. Соотношение между числом вершин графа и размером глобальной памяти, необходимой для его хранения, показано на рис. 6. Таким образом, для хранения матрицы смежности графа со 180 тысячами вершин необходимо около 3,78 Гбайт.

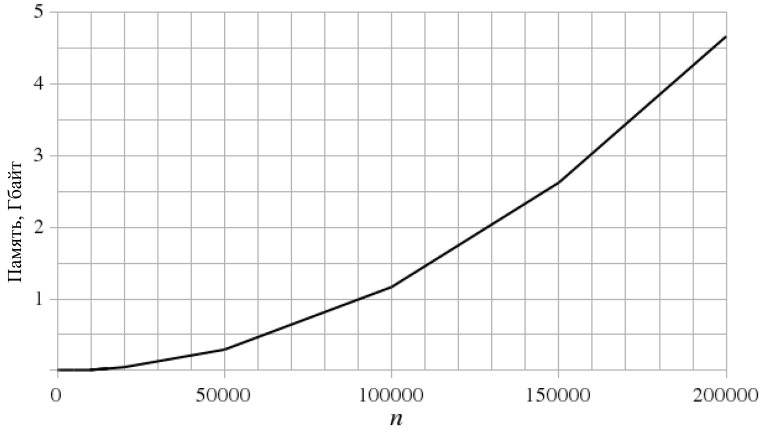

Рис. 6. Соотношение между числом вершин графа и размером глобальной памяти

Для хранения невзвешенного графа с n вершинами и m рёбрами, представленного в виде списка рёбер, необходимо $2(8(\log_2 n(\lceil m/64 \rceil + 1) + 1))$ байт. В этом случае потребуется около 3,34 Мбайт для хранения графа с 10^4 вершин и 10^6 рёбер. Для хранения дорожного графа США (около $24 \cdot 10^6$ вершин и $58 \cdot 10^6$ дуг) потребуется 348 Мбайт глобальной памяти. Файл данных этого графа расположен на сайте [24]. Для примера, графический ускоритель Kepler K40 оснащён 12 Гбайт памяти и 2 880 ядер CUDA [25].

3.3. Сравнение времени работы различных реализаций алгоритма Уоршалла

Для сравнения времени работы различных реализаций алгоритма Уоршалла на графических ускорителях использованы модифицированный и адаптированный Star-алгоритмы Уоршалла. Время работы в секундах последовательного алгоритма Уоршалла, модифицированного (листинг 2) и адаптированного (листинг 4) Star-алгоритмов приведено в табл. 2.

Таблица 2

Время работы последовательного алгоритма Уоршалла и его Star-версий

Количество вершин	1 000	2 000	3 000	4 000	5 000
Последовательный алгоритм	8,037	59,812	197,111	461,785	884,622
WARSHALL-C	4,827	11,161	23,363	42,661	64,454
WARSHALL-adapt	0,003	0,027	0,093	0,236	0,372

В работах [26 – 28] приводятся параллельные реализации алгоритма Уоршалла на GPU, но из-за различия используемых моделей графических ускорителей сравнить время их работы с временем работы Star-алгоритмов затруднительно.

Заключение

В работе приведена абстрактная модель ассоциативной параллельной обработки данных (STAR-машина). Обоснована актуальность реализации этой модели на современных архитектурах. Обоснован выбор графических ускорителей в качестве такой архитектуры. Приведена реализация этой модели на графических ускорителях с помощью технологии CUDA, которая позволяет получить на алгоритме Уоршалла ускорение в 13 раз на графе с 5 000 вершин.

Показано, что некоторые ассоциативные параллельные алгоритмы могут быть адаптированы для выполнения на графических ускорителях, если принимать во внимание архитектурные различия STAR-машины и графических ускорителей. Так, адаптация ассоциативной версии алгоритма Уоршалла под графические ускорители дала ускорение в 2 378 раз на графе с 5 000 вершин.

На данный момент Star-алгоритм записывается в виде процедуры на CPU и методы, реализующие базовые операции, запускают на графическом ускорителе функции, производящие вычисления. Исходя из этого, планируется продолжить работу над реализацией STAR-машины на графических ускорителях так, чтобы Star-алгоритм мог быть представлен в виде процедуры, запускаемой на GPU. Необходимо также сделать библиотеку базовых Star-процедур для выполнения на графических ускорителях. Заметим, что все базовые процедуры могут быть реализованы в виде процедур, запускаемых на GPU, чтобы свести к минимуму количество запусков ядер.

ЛИТЕРАТУРА

1. *Potter J. L.* Associative Computing: a Programming Paradigm for Massively Parallel Computers. Boston: Perseus Publishing, 1991. 304 p.
2. *Rudolph J. A.* A production implementation of an associative array processor — STARAN // AFIPS'72. N. Y.: ACM, 1972. P. 229–241.
3. *Foster C. C.* Content Addressable Parallel Processors. N. Y.: John Wiley & Sons, 1976. 233 p.
4. *Higuchi T., Furuya T., Handa K., et al.* IXM2: a parallel associative processor // ISCA'91. Proc. 18th Annual Intern. Symp. Computer Architecture. N. Y.: ACM, 1991. P. 22–31.
5. *Непомнящая А. Ш., Владыко М. А.* Сравнение моделей ассоциативного вычисления // Программирование. 1997. № 6. С. 41–50.
6. *Nepomniaschaya A. Sh.* Language STAR for associative and parallel computation with vertical data processing // Parallel Computing Technologies. Singapore: World Scientific, 1991. P. 258–265.
7. *Potter J., Baker J., Scott S., et al.* ASC: An Associative-Computing Paradigm // Computer. 1994. No. 27(11). P. 19–25.
8. *Nepomniaschaya A. Sh.* Basic associative parallel algorithms for vertical processing systems // Bulletin of the Novosibirsk Computing Center. 2009. Ser. Comp. Science. No. 9. P. 63–77.
9. *Nepomniaschaya A. Sh.* Constructions used in associative parallel algorithms for undirected graphs. Part 1 // Bulletin of the Novosibirsk Computing Center. 2013. Ser. Comp. Science. No. 35. P. 69–83.
10. *Nepomniaschaya A. Sh.* Constructions used in associative parallel algorithms for directed graphs // LNCS. 2015. V. 9251. P. 201–209.

11. *Nepomniaschaya A. Sh.* Solution of path problems using associative parallel processors // Intern. Conf. Parallel and Distributed Systems. Seoul: IEEE Computer Society Press, 1997. P. 610–617.
12. *Nepomniaschaya A. Sh. and Dvoskina M. A.* A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Informaticae. IOS Press, 2000. V. 43. P. 227–243.
13. *Nepomniaschaya A. Sh.* An associative version of the Bellman — Ford algorithm for finding the shortest paths in directed graphs // LNCS. 2001. V. 2127. P. 285–292.
14. *Nepomniaschaya A. Sh.* Efficient implementation of Edmonds' algorithm for finding optimum branchings on associative parallel processors // Proc. 8th Intern. Conf. Parallel and Distributed Systems (ICPADS'01). KyongJu City: IEEE Computer Society Press, 2001. P. 3–8.
15. *Непомнящая А. Ш.* Сравнение алгоритмов Прима — Дейкстры и Краскала с помощью ассоциативного параллельного процессора // Кибернетика и системный анализ. 2000. № 2. С. 19–27.
16. *Nepomniaschaya A. Sh.* Representation of the Gabow algorithm for finding smallest spanning trees with a degree constraint on associative parallel processors // LNCS. 1996. V. 1123. P. 813–817.
17. *Nepomniaschaya A. Sh.* Associative parallel algorithms for dynamic edge update of minimum spanning trees // LNCS. 2003. V. 2763. P. 141–150.
18. *Nepomniaschaya A. Sh.* Associative parallel algorithm for dynamic reconstructing a minimum spanning tree after deletion of a vertex // LNCS. 2005. V. 3606. P. 151–173.
19. *Непомнящая А. Ш.* Ассоциативный параллельный алгоритм для динамической обработки минимального каркаса после добавления к графу новой вершины // Кибернетика и системный анализ. 2006. № 1. С. 19–31.
20. *Nepomniaschaya A. Sh.* Efficient implementation of the Italiano algorithms for updating the transitive closure on associative parallel processors // Fundamenta Informaticae. IOS Press, 2008. V. 89. No. 2–3. P. 313–329.
21. *Nepomniaschaya A. Sh.* Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest paths subgraph // LNCS. 2009. V. 5698. P. 257–268.
22. *Непомнящая А. Ш.* Ассоциативная версия алгоритма Рамалингама для динамической обработки подграфа кратчайших путей после добавления к графу новой дуги // Кибернетика и системный анализ. 2012. № 3. С. 45–57.
23. *Mingxian J.* Associative operations from MASC to GPU // PDPTA-15 — The 21st Intern. Conf. on Parallel and Distributed Processing Techniques and Applications. Las Vegas: CSREA Press, 2015. P. 388–393.
24. `http://www.dis.uniroma1.it/challenge9/download.s` — 9th DIMACS Implementation Challenge — Shortest Paths. 2010.
25. `http://www.nvidia.ru/object/tesla-server-gpus-ru.html` — Высокопроизводительные вычисления для серверов| Графические процессоры Tesla|NVIDIA. 2016.
26. *Harish P. and Narayanan P.* Accelerating large graph algorithms on the GPU using CUDA // LNCS. 2007. V. 4873. P. 197–208.
27. *Katz G. J. and Kider Jr. T.* All-pairs shortest-paths for large graphs on the GPU // Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware. Sarajevo: Eurographics Association, 2008. P. 47–55.
28. *Lund B. and Smith J. W.* A Multi-Stage CUDA Kernel for Floyd — Warshall. arXiv preprint arXiv:1001.4108

REFERENCES

1. *Potter J. L.* Associative Computing: a Programming Paradigm for Massively Parallel Computers. Boston, Perseus Publishing, 1991. 304 p.
2. *Rudolph J. A.* A production implementation of an associative array processor — STARAN. AFIPS'72, N. Y., ACM, 1972, pp. 229–241.
3. *Foster C. C.* Content Addressable Parallel Processors. N. Y., John Wiley & Sons, 1976. 233 p.
4. *Higuchi T., Furuya T., Handa K., et al.* IXM2: a parallel associative processor. ISCA'91. Proc. 18th Annual Intern. Symp. Computer Architecture. N. Y., ACM, 1991, pp. 22–31.
5. *Nepomnyashchaya A. Sh. and Vladyko M. A.* Sravnenie modeley assotsiativnogo vychisleniya [Compare of associative computation models]. Programmirovanie, 1997, no. 6, pp. 41–50. (in Russian)
6. *Nepomniaschaya A. Sh.* Language STAR for associative and parallel computation with vertical data processing. Parallel Computing Technologies. Singapore, World Scientific, 1991, pp. 258–265.
7. *Potter J., Baker J., Scott S., et al.* ASC: An Associative-Computing Paradigm. Computer, 1994, no. 27(11), pp. 19–25.
8. *Nepomniaschaya A. Sh.* Basic associative parallel algorithms for vertical processing systems. Bulletin of the Novosibirsk Computing Center, 2009, ser. Comp. Science, no. 9, pp. 63–77.
9. *Nepomniaschaya A. Sh.* Constructions used in associative parallel algorithms for undirected graphs. Part 1. Bulletin of the Novosibirsk Computing Center, 2013, ser. Comp. Science, no. 35, pp. 69–83.
10. *Nepomniaschaya A. Sh.* Constructions used in associative parallel algorithms for directed graphs. LNCS, 2015, vol. 9251, pp. 201–209.
11. *Nepomniaschaya A. Sh.* Solution of path problems using associative parallel processors. Intern. Conf. Parallel and Distributed Systems, Seoul, IEEE Computer Society Press, 1997, pp. 610–617.
12. *Nepomniaschaya A. Sh. and Dvoskina M. A.* A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors. Fundamenta Informaticae, IOS Press, 2000, vol. 43, pp. 227–243.
13. *Nepomniaschaya A. Sh.* An associative version of the Bellman — Ford algorithm for finding the shortest paths in directed graphs. LNCS, 2001, vol. 2127, pp. 285–292.
14. *Nepomniaschaya A. Sh.* Efficient implementation of Edmonds' algorithm for finding optimum branchings on associative parallel processors. Proc. 8th Intern. Conf. Parallel and Distributed Systems (ICPADS'01), KyongJu City, IEEE Computer Society Press, 2001, pp. 3–8.
15. *Nepomnyashchaya A. Sh.* Sravnenie algoritmov Prima — Deykstry i Kraskala s pomoshch'yu assotsiativnogo parallel'nogo protsessora [Compare of Prima — Dijkstra and Kruskal algorithms using associative parallel processor]. Kibernetika i sistemnyy analiz, 2000, no 2, pp. 19–27. (in Russian)
16. *Nepomniaschaya A. Sh.* Representation of the Gabow algorithm for finding smallest spanning trees with a degree constraint on associative parallel processors. LNCS, 1996, vol. 1123, pp. 813–817.
17. *Nepomniaschaya A. Sh.* Associative parallel algorithms for dynamic edge update of minimum spanning trees. LNCS, 2003, vol. 2763, pp. 141–150.
18. *Nepomniaschaya A. Sh.* Associative parallel algorithm for dynamic reconstructing a minimum spanning tree after deletion of a vertex. LNCS, 2005, vol. 3606, pp. 151–173.
19. *Nepomnyashchaya A. Sh.* Assotsiativnyy parallel'nyy algoritm dlya dinamicheskoy obrabotki minimal'nogo karkasa posle dobavleniya k grafu novoy vershiny [An associative parallel

algorithm for dynamic update of a minimum spanning tree after adding a new vertex to a graph]. Kibernetika i sistemnyy analiz, 2006, no. 1, pp. 19–31. (in Russian)

20. *Nepomniaschaya A. Sh.* Efficient implementation of the Italiano algorithms for updating the transitive closure on associative parallel processors. Fundamenta Informaticae, IOS Press, 2008, vol. 89, no. 2–3, pp. 313–329.
21. *Nepomniaschaya A. Sh.* Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest paths subgraph. LNCS, 2009, vol. 5698, pp. 257–268.
22. *Nepomnyashchaya A. Sh.* Assotsiativnaya versiya algoritma Ramalingama dlya dinamicheskoy obrabotki podgrafa kratchayshikh putey posle dobavleniya k grafu novoy dugi [Associative version of the ramalingam algorithm for dynamic update of the shortest-path subgraph after insertion of a new edge to the graph]. Kibernetika i sistemnyy analiz, 2012, no. 3, pp. 45–57. (in Russian)
23. *Mingxian J.* Associative operations from MASC to GPU. PDPTA-15 — The 21st Intern. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas, CSREA Press, 2015, pp. 388–393.
24. `http://www.dis.uniroma1.it/challenge9/download.s` — 9th DIMACS Implementation Challenge — Shortest Paths, 2010.
25. `http://www.nvidia.ru/object/tesla-server-gpus-ru.html`
26. *Harish P. and Narayanan P.* Accelerating large graph algorithms on the GPU using CUDA. LNCS, 2007, vol. 4873, pp. 197–208.
27. *Katz G. J. and Kider Jr. T.* All-pairs shortest-paths for large graphs on the GPU. Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware, Sarajevo, Eurographics Association, 2008, pp. 47–55.
28. *Lund B. and Smith J. W.* A Multi-Stage CUDA Kernel for Floyd — Warshall. arXiv preprint arXiv:1001.4108