УДК 519.67

**ОБОБЩЕННЫЙ БЛОЧНЫЙ АЛГОРИТМ ФЛОЙДА-УОРШЕЛЛА**

**Н. А. Лиходед, Д. С. Сипейко**

*Белорусский государственный университет, пр. Независимости, 4, 220030, г. Минск, Беларусь*

**Аннотация.** Одним из наиболее используемых на практике алгоритмовдля поиска кратчайших путей между всеми парами вершин во взвешенных графах является алгоритм Флойда-Уоршелла. Блочная версия алгоритма служит основой для получения эффективных параллельных алгоритмов для реализации на многоядерных CPU, на компьютерах с распределенной памятью, на графических процессорах. Увеличение зернистости вычислений в блочных версиях алгоритмов приводит к более эффективному использованию кэшей и к более эффективной организации параллельных вычислений. В этой работе предложено обобщение блочного алгоритма Флойда-Уоршелла. Порядок выполнения блоков вычислений реорганизован таким образом, чтобы элементы массива, участвующие в коммуникационных операциях как чтения, так и записи, реже вытеснялись из памяти с быстрым доступом. Тогда при реализации алгоритма на графическом процессоре реже, по сравнению с исходным блочным алгоритмом, используется медленная глобальная память.

**Ключевые слова:**параллельные алгоритмы, поиск кратчайших путей, графы, алгоритм Флойда-Уоршелла, блочный алгоритм, GPU.

**Благодарность.** Работа выполнена в рамках государственной программы научных исследований Республики Беларусь «Конвергенция-2020», подпрограмма «Методы математического моделирования сложных систем».

*Лиходед Николай Александрович* – доктор физико-математических наук, профессор кафедры вычислительной математики факультета прикладной математики и информатики БГУ, [likhoded@bsu.by](mailto:likhoded@bsu.by), +375 29 578 96 23.

*Сипейко Дмитрий Сергеевич* – магистрант факультета прикладной математики и информатики БГУ, mintaids@ya.ru.

**GENERALIZED BLOCKED FLOYD-WARSHALL ALGORITHM**

**N. A. Likhoded, D. S. Sipeyko**

*Belarusian State University, 4 Niezaliežnasci Avenue, Minsk 220030, Belarus*

**Abstract.** One of the most commonly used on practice all-pairs shortest paths algorithms on weighted graphs is Floyd-Warshall algorithm. Blocked version serves as a basis for obtaining effective parallel algorithms to be implemented on multicore CPU, on computers with distributed memory, on GPU. Increasing computation granularity in blocked versions of algorithm leads to a more efficient usage of caches and more efficient organization of parallel computations. In this paper we introduce generalization of blocked Floyd-Warshall algorithm. Computing blocks execution order was reorganized in such a way that array elements which participate in communication operations, both reading and writing, are pushed out of fast-access memory less often. This means that in GPU implementation slow global memory is used less often, compared with the original blocked algorithm.

**Keywords:**parallel algorithms, shortest paths, graphs, Floyd-Warshall algorithm, block algorithm, GPU.

**Acknowledgment.** The prepared report was sponsored by the government program of scientific research of the Republic of Belarus "Convergence-2020", subprogram "Methods of mathematical modeling of complex systems".

*Nikolai A. Likhoded*, D. Sc., Belarusian State University. likhoded@bsu.by

*Dmitry S. Sipeyko,* undergraduate, Belarusian State University. mintaids@ya.ru.

**Введение**

Поиск в графах играет важную роль в анализе больших наборов данных. Широкое применение в задачах маршрутизации и задачах логистики имеют алгоритмы поиска кратчайших путей. Одним из наиболее используемых на практике алгоритмовдля поиска кратчайших путей между всеми парами вершин во взвешенных графах является алгоритм Флойда-Уоршелла.

Все алгоритмы решения задачи поиска кратчайших путей между всеми парами вершин графа имеют нелинейную трудоемкость, поэтому для решения задач большого размера требуются алгоритмы, хорошо приспособленные для реализации на современных вычислительных устройствах. Одним из таких алгоритмов является блочный алгоритм Флойда-Уоршелла [1, 2]. Он служит основой для получения эффективных параллельных версий алгоритма для реализации на многоядерных CPU, на компьютерах с распределенной памятью, на графических процессорах (GPU) [3–6]. Как и во многих других случаях использования блочных алгоритмов, увеличение зерна вычислений приводит к более эффективному использованию кэшей и к более эффективной организации параллельных вычислений.

Целью этой работы является построение модификации блочного алгоритма Флойда-Уоршелла, в которой порядок выполнения блоков вычислений реорганизован таким образом, чтобы реже вытеснялись из памяти с быстрым доступом (кэши, регистры) элементы массива, участвующие не только в коммуникационных операциях чтения, но еще и в операциях записи. Можно ожидать, что тогда при реализации алгоритма будет более эффективно, по сравнению с исходным блочным алгоритмом, использоваться память с быстрым доступом. Предлагаемая модификация блочного алгоритма Флойда-Уоршелла задает параметрическое семейство блочных алгоритмов, которое включает в себя и исходный алгоритм.

Степень использования памяти с быстрым доступом отражает вычислительное свойство алгоритма, называемое локальностью. При реализации алгоритмов на многопроцессорных вычислительных устройствах использование локальности играет важнейшую роль для достижения высокой производительности [7, 8]. В этой работе построенный обобщенный алгоритм Флойда-Уоршелла реализован на графическом процессоре. При вычислениях на GPU быстрым является процесс обращения к регистрам, разделяемой памяти мультипроцессора и кэшам, но не обращение к глобальной памяти GPU. Реализация на основе обобщенного алгоритма проводит к уменьшению числа обращений к глобальной памяти и, как показали вычислительные эксперименты, к уменьшению времени выполнения.

**Точечный алгоритм. Зависимости алгоритма**

Пусть *G*(*V,E*) – некоторый граф, *V* – множество вершин, *E* – множестворебер. Будем считать, что вершины графа занумерованы последовательными целыми числами, начиная от 1 и заканчивая *n*, а граф задан матрицей смежности *А* размера *n×n*.

Приведем основную часть последовательного точечного (т.е. не блочного) алгоритма Флойда-Уоршелла:

do *k =* 1*, n*

do *i =* 1*, n*

do *j =* 1*, n*

*S*1(*k,i,j*): *a*(*i*,*j*) *=* min(*a*(*i,j*)*, a*(*i,k*) *+ a*(*k,j*))

enddo

enddo

enddo

Перед началом выполнения алгоритма матрица расстояний *А* заполняется длинами рёбер графа (или заведомо большим числом, если ребра нет). На каждом шаге *k* алгоритм обновляет матрицу *А*. После выполнения алгоритма матрица *А* содержит длины кратчайших путей между всеми вершинами графа.

В гнезде циклов имеется один выполняемый оператор *S*1 и используется один массив *a* размерности 2. Область изменения параметров циклов (область итераций) для оператора *S*1 имеет размерность 3:

*V*1*=*{(*k,i,j*)Z3| 1≤*k*≤*n*, 1≤*i*≤*n*, 1≤*j*≤*n*}.

Формально между операциями одного слоя *k* существуют информационные зависимости. Эти зависимости связаны с обновлением или использованием элементов массивов *a*(*i,j*) при *i=k* или *j=k*. Устранить зависимости между операциями одного слоя *k* можно введением дополнительных массивов [5]. Но непосредственно из записи алгоритма Флойда-Уоршелла следует, что данные *a*(*i,j*) не обновляются, если *i=k* или *j=k*. Поэтому для фиксированного *k* все операции фактически используют данные, вычисленные на предыдущем (*k*–1)-м шаге. Можно допустить, что все операции при фиксированном *k* не зависят друг от друга и их можно выполнять в произвольном порядке.

Рассмотрим зависимости и векторы зависимостей алгоритма с учетом сделанного допущения. Векторы зависимостей будем для наглядности помечать элементами матрицы, фигурирующими на порождающих зависимости вхождениях. Например, вектор *da*(*i,j*),*a*(*i,k*) порождается зависимостью между данными *a*(*i,j*) в левой части оператора *S*1, и данными *a*(*i,k*) в правой части оператора. Укажем итерации, порождающие зависимости, и векторы зависимостей:

* *S*1(*k–*1,*i,j*)→*S*1(*k,i,j*): данное *a*(*i,j*), вычисленное на итерации (*k–*1,*i,j*), является аргументом *a*(*i,j*) для вычислений на итерации (*k,i,j*); *da*(*i,j*),*a*(*i,j*)*=*(1,0,0);
* *S*1(*k–*1,*i,k*)→*S*1(*k,i,j*): *a*(*i,k*), вычисленное на итерации (*k–*1,*i,k*), является аргументомдля вычислений на итерациях (*k,i,j*); *da*(*i,j*),*a*(*i,k*)*=*(1,0*,j–k*);
* *S*1(*k–*1,*k,j*)→*S*1(*k,i,j*): *a*(*k,j*), вычисленное на итерации (*k–*1,*k,j*), является аргументомдля вычислений на итерациях (*k,i,j*); *da*(*i,j*),*a*(*k,j*)*=*(1,*i–k,*0).

**Блочный алгоритм**

Блочный алгоритм Флойда-Уоршела с трехмерными (3D) блоками впервые предложен в работе [1]. Выделим *Q*×*Q*×*Q* блоков размера *r*×*r*×*r*, где *Q=*, *r* – параметр, задающий размеры блоков. Отметим, что одинаковые размеры блока имеют существенное значение, а не выбраны для простоты. Пусть *kgl*, *igl*, *jgl* – номера частей, на которые при формировании блоков разбиваются области значений параметров *k, i, j* циклов, 0≤*kgl*,*igl*,*jgl*≤*Q*–1. Блоки вычислений называют также тайлами.

Блок Tile(*kgl*,*igl*,*jgl*) имеет следующий вид:

do *k =* 1+*kglr*, min((*kgl+*1)*r*, *n*)

do *i =* 1+*iglr*, min((*igl+*1)*r*, *n*)

do *j =* 1+*jglr*, min((*jgl+*1)*r*, *n*)

*a*(*i*,*j*) *=* min(*a*(*i,j*)*, a*(*i,k*) *+ a*(*k,j*))

enddo

enddo

enddo

Установим корректный порядок выполнения блоков и обоснуем корректный порядок выполнения вычислений в блоке.

Рассмотрим блоки некоторой блочной итерации *kgl* (некоторого блочного слоя *kgl*), т.е. блоки Tile(*kgl*,*igl*,*jgl*) при фиксированном *kgl*. Назовем:

* Tile(*kgl*,*kgl*,*kgl*) – ведущий блок,
* Tile(*kgl*,*kgl*,*jgl*), 0≤*jgl*≤*Q*–1, *jgl*≠*kgl*, – блоки ведущей строки,
* Tile(*kgl*,*igl*,*kgl*), 0≤*igl*≤*Q*–1, *igl*≠*kgl*, – блоки ведущего столбца.

Анализ зависимостей показывает:

1. Вычисления любого ведущего блока Tile(*kgl*,*kgl*,*kgl*) не зависят от вычислений других блоков блочного слоя; для вычисления элементов ведущего блока нужны только его элементы. Ведущий блок на блочном слое следует вычислять первым. Ведущие блоки назовем I-блоками (Independent blocks [4]).

2. Вычисления блоков ведущей строки и ведущего столбца зависят от вычислений ведущего блока блочного слоя. Для вычислений этих блоков необходимы их собственные элементы и уже подсчитанные элементы ведущего блока. Между собой эти блоки не конкурируют, поэтому последовательность их вычислений на блочном слое может быть произвольной. Назовем блоки ведущих строк и столбцов SD-блоками (Singly Dependent blocks).

3. Остальные блоки Tile(*kgl*,*igl*,*jgl*), 0≤*igl*,*jgl*≤*Q*–1, *igl*≠*kgl*, *jgl*≠*kgl*, зависят от вычислений блоков ведущих строк и столбцов. Для вычислений этих блоков нужны их собственные элементы, а также элементы соответствующих блоков ведущей строки и ведущего столбца. Эти блоки вычисляются на блочном слое в произвольном порядке после вычисления блоков ведущей строки и столбца. Блоки вне ведущих строк и столбцов назовем DD-блоками (Doubly Dependent blocks).

Опишем шаги блочной итерации *kgl*:

1. Производятся вычисления ведущего блока (I-блока). Фактически выполняется обычный точечный алгоритм Флойда-Уоршела; в итоге сохраняется версия элементов подматрицы ведущего блока на итерации (*kgl+*1)*r* (или min((*kgl+*1)*r*, *n*) для последнего слоя).

2. Производятся вычисления блоков ведущей строки и ведущего столбца (SD-блоков). При обращении к элементам ведущего блока происходит обращение к их последней версии. Блоки могут вычисляться независимо, в произвольном порядке.

3. Производятся вычисления оставшихся блоков (DD-блоков). При обращении к элементам блоков ведущей строки и ведущего столбца происходит обращение к их последней версии. Блоки могут вычисляться независимо, в произвольном порядке.

**Замечание 1.** Результаты промежуточных вычислений точечного и блочного алгоритмов Флойда-Уоршела могут не совпадать. Тем не менее, блочный алгоритм Флойда-Уоршелла приводит к корректному результату [2]**.**

Основная часть алгоритма Флойда-Уоршелла с выделенными 3D блоками имеет следующий вид [4] (циклы, итерации которых заведомо можно выполнять независимо, запишем как dopar):

do *kgl =* 0, *Q*–1

Tile(*kgl*,*kgl*,*kgl*) // Вычисления I-блока

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

Tile(*kgl*,*kgl*,*jgl*) // Вычисления SD-блоков ведущей строки

enddopar

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

Tile(*kgl*,*igl*,*kgl*) // Вычисления SD-блоков ведущего столбца

enddopar

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

Tile(*kgl*,*igl*,*jgl*) // Вычисления DD-блоков

enddopar

enddopar

enddo (*kgl*)

**Замечание 2.** В DD-блоках вычисления всех *a*(*i*,*j*) происходят независимо друг от друга: *a*(*i,k*) и *a*(*k,j*) вычисляются вне DD-блока, между операциями блока остаются только зависимости, задаваемые вектором *da*(*i,j*),*a*(*i,j*)*=*(1,0*,*0). На практике в DD-блоках производят перестановку циклов так, чтобы цикл с параметром *k* стал самым внутренним. Обозначим такой блок через TileDD(*kgl*,*igl*,*jgl*) и запишем его явный вид:

dopar *i =* 1+*iglr*, min((*igl+*1)*r*, *n*)

dopar *j =* 1+*jglr*, min((*jgl+*1)*r*, *n*)

do *k =* 1+*kglr*, min((*kgl+*1)*r*, *n*)

*a*(*i*,*j*) *=* min(*a*(*i,j*)*, a*(*i,k*) *+ a*(*k,j*))

enddo

enddopar

enddopar

**Модифицированный блочный алгоритм**

В рассмотренном блочном алгоритме последовательно выполняются блочные итерации *kgl*. Для каждого фиксированного *kgl* сначала производятся вычисления I-блока Tile(*kgl*,*kgl*,*kgl*), затем (в произвольном порядке) вычисления 2(*Q*–1) SD-блоков Tile(*kgl*,*kgl*,*jgl*) и Tile(*kgl*,*igl*,*kgl*), затем (в произвольном порядке) вычисления (*Q*–1)×(*Q*–1) DD-блоков Tile(*kgl*,*igl*,*jgl*). Как уже отмечалось, размеры блока (*r*×*r*×*r* итераций) могут быть только одинаковые.

Реорганизуем порядок выполнения блоков вычислений таким образом, чтобы атомарно, как одна макрооперация, выполнялось некоторое количество тайлов с подряд идущими номерами первой блочной координаты и фиксированными номерами второй и третьей блочной координаты. Такие объединенные тайлы будем называть мультитайлами (вообще говоря, мультитайл может содержать только один тайл). В макрооперациях-мультитайлах выполнение большего числа подряд идущих итераций *k* приводит к более редкому вытеснению из памяти с быстрым доступом (кэши, регистры) элементов массива на вхождениях *a*(*i*,*j*). Вхождения *a*(*i*,*j*) требуют как операций чтения, так и операций записи; в то время как вхождения *a*(*i,k*) и *a*(*k,j*) требуют только операций чтения. Можно ожидать, что при реализации алгоритма будет более эффективно использоваться память с быстрым доступом.

Пусть κ – некоторое число в пределах от 1 до *Q*, а *l* – некоторое число в пределах от 0 доκ–1. Параметр κ задает число блочных итераций *kgl* (число блочных слоев), которые используются для получения мультитайлов. От параметра *l* зависит число блочных итераций *kgl* (число блочных слоев), которые объединяются для получения мультитайлов.

Введем в рассмотрение процедуры выполнения мультитайлов.

CalcLeadBlock(*kgl*,*l*) – процедура вычисления (при фиксированных параметрах *kgl*, *l* процедуры) мультитайла

, если *l=*0,

, если *l*≠0,

объединяющего *l* DD-блоков и один I-блок исходного блочного алгоритма. Мультитайл включает I-блок и не вычисленные DD-блоки (если таковые имеются, т.е. если *l>*0) с такими же, как у I-блока, номерами второй и третьей блочной координаты и с меньшими номерами первой блочной координаты.

CalcLeadRowAndColumn(*kgl*,*l*) – процедура вычисления *Q*–1 мультитайлов, каждый из которых включает один SD-блок (*kgl+l*)-й блочной строки исходного блочного алгоритма, и *Q*–1 мультитайлов, каждый из которых включает один SD-блок (*kgl+l*)-го блочного столбца исходного блочного алгоритма. Кроме того, каждый мультитайл включает DD-блоки (если таковые имеются) с такими же, как у SD-блока, номерами второй и третьей блочной координаты и с меньшими номерами первой блочной координаты.

Мультитайл с SD-блоком (*kgl+l*)-й блочной строки имеет вид

,

если *l=*0, *jgl =* 0, 1, ..., *Q*–1 (*jgl*≠*kgl*),

,

если *l*≠0, *jgl =* 0, 1, ..., *Q*–1 (*jgl*≠ *kgl*, ..., *kgl+l*),

,

если *l*≠0, *jgl =kgl*, ..., *kgl+l*–1.

Мультитайл с SD-блоком (*kgl+l*)-го блочного столбца имеет вид

,

если *l=*0, *igl =* 0, 1, ..., *Q*–1 (*igl*≠*kgl*),

,

если *l*≠0, *igl=* 0, 1, ..., *Q*–1 (*igl*≠ *kgl*, ..., *kgl+l*),

,

если *l*≠0, *igl =kgl*, ..., *kgl+l*–1.

CalcLeadRowAndColumnReverse(*kgl*,κ,*l*) – процедура вычисления (при фиксированных параметрах *kgl*, κ, *l* процедуры, *l<*κ–1) мультитайлов (*kgl+l*)-й блочной строки исходного блочного алгоритма и (*kgl+l*)-го блочного столбца исходного блочного алгоритма. Каждый мультитайл объединяет κ–*l*–1 DD-блоков с фиксированными номерами второй и третьей блочной координаты и с большими, чем *kgl+l*, номерами первой блочной координаты:

,

*jgl =* 0, 1, ..., *kgl+l*, *jgl=kgl+*κ, ..., *Q*–1,

,

*igl =* 0, 1, ..., *kgl+l*–1, *igl=kgl+*κ, ..., *Q*–1.

CalcRestBlocks(*kgl*,κ), κ≠*Q*, – процедура вычисления (*Q*–κ)×(*Q*–κ) мультитайлов

,

*igl =* 0, 1, ..., *Q*–1 (*igl*≠ *kgl*, *kgl+*1, ..., *kgl+*κ–1),

*jgl =* 0, 1, ..., *Q*–1 (*jgl*≠ *kgl*, *kgl+*1, ..., *kgl+*κ–1),

каждый из которых объединяет κ DD-блоков (вне ведущих блочных строк и столбцов) исходного блочного алгоритма.

Отметим, что во всех процедурах вычисление мультитайлов при фиксированных *kgl*, κ, *l* можно выполнять независимо (кроме процедуры CalcLeadBlock(*kgl*,*l*), вычисляющей только один мультитайл).

Основную часть обобщенного блочного алгоритма Флойда-Уоршелла можно представить следующим образом:

do *kgl =* 0, *Q*–1, κ // вычисления с шагом κ

do *l =* 0, κ–1

CalcLeadBlock(*kgl*,*l*)

CalcLeadRowAndColumn(*kgl*,*l*)

enddo

do *l =* κ–2, 0, –1 // вычисления с шагом –1

CalcLeadRowAndColumnReverse(*kgl*,κ,*l*)

enddo

CalcRestBlocks(*kgl*,κ)

enddo (*kgl*)

Если κ*=*1, то получим известный блочный алгоритм Флойда-Уоршелла (цикл do *l=*κ–2,0,–1 отсутствует).

Если κ*=*2, то получим случай, рассмотренный в магистерская диссертации Сычевой О.И. «Разработка и программная реализация новых параллельных версий алгоритма Флойда-Уоршелла» (БГУ, 2016).

Если κ*=Q*, то внешний цикл do *kgl=*0,*Q*–1,κ вырождается в одну итерацию *kgl=*0, циклы с параметром *l* охватывают вычисление всех блоков, функция CalcRestBlocks(*kgl*,κ) отсутствует.

**Реализация на графическом ускорителе**

Графический процессор (GPU) выполняет множество параллельных потоков вычислений. Потоки объединяются в блоки вычислений, каждый блок потоков выполняется атомарно на одном из мультипроцессоров графического процессора. Должны быть указаны блоки, которые могут выполняться мультипроцессорами одновременно и независимо друг от друга.

Точечный алгоритм Флойда-Уоршелла обладает естественным параллелизмом в пределах одной итерации. Поэтому на каждой итерации *k* (*k=*1*,*2,…,*n*) можно выделить двумерные (2D) блоки вычислений, которые могут выполняться независимо друг от друга. GPU-реализация алгоритма Флойда-Уоршелла, основанная на 2D-блоках вычислений, предложена в работе [9]. Матрица *A* хранится в глобальной памяти GPU, поэтому на каждой итерации *k* необходима запись всех обновленных элементов матрицы в глобальную память. Из глобальной памяти считываются все нужные данные, подсчитанные на предыдущей итерации.

При вычислениях на GPU быстрым является процесс обращения к разделяемой памяти мультипроцессора и к кэшам, но не обращение к глобальной памяти GPU. В работе [10] реализован на GPU алгоритм Флойда-Уоршелла с 3D блоками. Использование блочного алгоритма с 3D блоками позволило существенно уменьшить время выполнения алгоритма. Время реализации уменьшилось главным образом за счет того, что запись обновленных элементов матрицы в глобальную память производится не на каждой итерации *k*, а на каждой *r*-й итерации *k*. На каждой блочной итерации *kgl* требуется три так называемых запуска ядра (т.е. три процедуры выполнения блоков вычислений):

1. Запускаются вычисления ведущего блока (I-блока). Используется *r×r* потоков – один поток вычисляет один элемент матрицы. Все потоки в одном блоке можно запустить, если *r≤*32 (в одном блоке может быть до 1024 потоков). Для каждого потока нужно один элемент скопировать из глобальной памяти в разделяемую, обновить его на *r* слоях и вернуть новое значение в глобальную память.

2. Запускаются вычисления блоков ведущей строки и ведущего столбца (SD-блоков); в каждом блоке используется *r×r* (*r≤*32) потоков. Напомним, для вычислений этих блоков необходимы их собственные элементы и уже подсчитанные элементы ведущего блока. Для каждого из запускаемых блоков в разделяемой памяти хранятся две матрицы размером *r×r*: помимо своих элементов потокам суммарно требуются *r×r* элементов I-блока.

3. Запускаются вычисления DD-блоков; в каждом блоке используется *r×r* потоков (*r≤*32). Для каждого из запускаемых блоков в разделяемой памяти хранятся три матрицы размера *r×r*: помимо своих элементов потокам суммарно требуются *r×r* элементов блока ведущей строки и *r×r* элементов блока ведущего столбца.

**Замечание 3.** При вычислении DD-блоков в разделяемой памяти достаточно хранить только матрицы, связанные с вхождениями *a*(*i,k*) и *a*(*k,j*): потокам не понадобится обращаться к элементам *a*(*i,,j*), которые пересчитывают другие потоки – нужен будет только свой элемент и элементы двух SD-блоков. Каждый поток может хранить элемент, который он пересчитывает, в своей регистровой памяти.

Дальнейшая оптимизация алгоритма путем уменьшения объема используемой разделяемой памяти в блочном алгоритме Флойда-Уоршелла предложена в работе [5]. Главная идея подхода – многостадийное чтение SD-блоков при вычислении DD-блоков. Уменьшение размеров разделяемой памяти, необходимой блоку в каждый момент времени, позволяет получить заметный выигрыш в производительности, так как уменьшение объема разделяемой памяти, используемой в блоке, позволяет мультипроцессору выполнять большее количество блоков одновременно.

Построенный обобщенный алгоритм Флойда-Уоршелла позволяет реализовать на графическом процессоре 3D блоки размера (*l+*1)*r*×*r*×*r*, где, напомним, *l* изменяется в пределах от 0 доκ–1. Запись обновленных элементов матрицы в глобальную память производится на (*l+*1)*r*-й итерации *k*, а не *r*-й итерации, как в случаях блоков размера *r*×*r*×*r*.

Опишем вычислительные эксперименты. В экспериментах использовалось многостадийное чтение SD-блоков, число потоков в одном блоке равно 1024 (*r*×*r=*32×32).

Эксперименты проводились на графическом процессоре NVIDIA GeForce GTX 670. Приведем некоторые характеристики этого GPU: мультипроцессоров – 7, ядер в устройстве – 1344, объем глобальной памяти – 2 Гб, объем разделяемой памяти – 48 Кб на каждый мультипроцессор, количество 32-битных регистров в мультипроцессоре – 65 536, используемая архитектура – Kepler .

На рис. 1 продемонстрирована зависимость времени вычислений реализации алгоритма от параметра κ, влияющего на число записей в глобальную память.

Рис. 1. Зависимость времени вычислений (сек) реализации обобщенного блочного алгоритма (*n=*8192, *r=*32) от параметра κ

Fig. 1. Dependency of computation time (sec) of generalized blocked algorithm implementation (*n*=8192, *r*=32) on the parameter κ

График показывает, что при нескольких значениях параметра κ обобщенного блочного алгоритма достигается меньшее время вычислений по сравнению со случаем κ*=*1 (классический блочный алгоритм). В этом примере число вершин графа равно 8192. Аналогичная картина наблюдается и в примерах с другим числом вершин.

**Замечание 4.** При обращении к функции CalcLeadBlock выполняется только один мультитайл, поэтому активен только один (из нескольких) мультипроцессор GPU. Можно организовать вычисление функции CalcLeadBlock одновременно на всех мультипроцессорах. Вычислительные экспериментах показали, что общее время реализации алгоритма при этом уменьшается очень незначительно, так как почти все временные затраты приходятся на вычисление других функций.

Таким образом, в работе построено параметрическое семейство блочных алгоритмов Флойда-Уоршелла, которое включает в себя и классический блочный алгоритм, рассмотрена реализация предложенного алгоритма на графическом ускорителе.

**Библиографические ссылки**

1. Venkataraman G., Sahni S., Mukhopadhyaya S. A blocked all-pairs shortest-paths algorithm // J. Exp. Algorithmics. 2003. Vol. 8. P. 2.2.
2. Park J., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache performance // IEEE Transactions on Parallel and Distributed Systems. 2004. Vol. 15, № 9. P. 769–782.
3. Srinivasan T., Balakrishnan R., Gangadharan S. A., Hayawardh V. A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment // Proceedings of the 13th International Conference on Parallel and Distributed Systems, Washington, 2007.
4. Lund B.D. Smith J.W. A multi-stage cuda kernel for floyd-warshall // CoRR abs/1001.4108. 2010.
5. Mullapudi R.T., Bondhugula U. Tiling for dynamic scheduling // Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, Vienna, Austria, 2014.
6. Прихожий А.А., Карасик О.Н. Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа // Системный анализ и прикладная информатика. 2017. № 3. С. 68–75.  <https://doi.org/10.21122/2309-4923-2017-3-68-75>
7. Воеводин Вл.В., Воеводин Вад.В. Спасительная локальность суперкомпьютеров // Открытые системы. 2013. № 9. С. 12–15.
8. Buluc A., Gilberta J.R., Budak C. Solving path problems on the GPU // Parallel Computing. 2010. Vol. 36. № 5-6. P. 241–253.
9. Harish P., Narayanan P. Accelerating large graph algorithms on the GPU using CUDA // Lecture Notes in Computer Science. 2007. Vol. 4873. P. 197.
10. Katz G.J., Kider J. All-pairs shortest-paths for large graphs on the GPU // Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. Sarajevo, Bosnia and Herzegovina: Eurographics Association. 2008. P. 47–55.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

**ДЛЯ**

**МЕЖДУНАРОДНЫХ БАЗ ДАННЫХ**

1. Venkataraman G., Sahni S., Mukhopadhyaya S. A blocked all-pairs shortest-paths algorithm // J. Exp. Algorithmics. 2003. Vol. 8. pp. 857–874.
2. Park J., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache performance // IEEE Transactions on Parallel and Distributed Systems. 2004. Vol. 15, No. 9. pp.  769–782.
3. Srinivasan T., Balakrishnan R., Gangadharan S. A., Hayawardh V. A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment // Proceedings of the 13th International Conference on Parallel and Distributed Systems, Washington, 2007.
4. Lund B.D. Smith J.W. A multi-stage cuda kernel for floyd-warshall // CoRR abs/1001.4108. 2010.
5. Mullapudi R.T., Bondhugula U. Tiling for dynamic scheduling // Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, Vienna, Austria, 2014.
6. Prihozhy A.A., Karasik O.N. Heterogenious blocked all-pairs shortest paths algorithm // System analysis and applied information science. 2017. No. 3. pp.  68–75. (In Russ.)  <https://doi.org/10.21122/2309-4923-2017-3-68-75>
7. Voevodin Vl.V., Voevodin Vad.V. [**The fortunate locality of supercomputers**](http://www.osp.ru/os/2013/09/13038278/) // Open Systems. 2013. № 9. P. 12–15. (in Russ.)
8. Buluc A., Gilberta J.R., Budak C. Solving path problems on the GPU // Parallel Computing. 2010. Vol. 36. No. 5–6. pp. 241–253.
9. Harish P., Narayanan P. Accelerating large graph algorithms on the GPU using CUDA // Lecture Notes in Computer Science. 2007. Vol. 4873. pp. 197.
10. Katz G.J., Kider J. All-pairs shortest-paths for large graphs on the GPU // Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. Sarajevo, Bosnia and Herzegovina: Eurographics Association. 2008. pp. 47–55.