**ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ФЛОЙДА-УОРШЕЛЛА**

**ПОИСКА КРАТЧАЙШИХ ПУТЕЙ**

Точечный алгоритм Флойда-Уоршелла, зависимости алгоритма. Блочный алгоритм с двумерными блоками. Блочный алгоритм с трехмерными блоками. Реализации на графических ускорителях блочного алгоритма с 2D-блоками. Реализации на графических ускорителях блочного алгоритма с 3D-блоками. Реализации, использующие тайлинг второго уровня. Об улучшении локальности реализаций на графических ускорителях.

Проблема поиска кратчайших путей хорошо исследована в теории графов. Полученные алгоритмы имеют широкое применение в задачах маршрутизации и задачах логистики. Все алгоритмы решения задачи поиска кратчайших путей между всеми парами вершин графа имеют нелинейную трудоемкость, поэтому одним из путей решения задачи большого размера является разработка параллельных версий алгоритма. Одним из наиболее используемых на практике алгоритмовдля поиска кратчайших путей между всеми парами вершин во взвешенных графах является алгоритм Флойда-Уоршелла [1]. В этом материале рассмотрены блочные алгоритмы Флойда-Уоршелла и их параллельные версии для реализации на графических ускорителях (GPU).

**Точечный алгоритм. Зависимости алгоритма**

Пусть *G*(*V,E*) – 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) *V*1*=*{(*k,i,j*)Z3| 1≤*k*≤*n*, 1≤*i*≤*n*, 1≤*j*≤*n*} для оператора *S*1 имеет размерность 3.

**Замечание 0.** Алгоритм Флойда-Уоршелла в некотором сысле сходен с алгоритмом прямого хода метода Гаусса решения системы уравнений с матрицей *A*:

do *k=* 1*, n–*1

do *i= k+*1*, n*

do *j= k+*1*, n+*1

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

enddo

enddo

enddo

Область итераций алгоритма Флойда-Уоршелла проще, но анализ зависимостей сложнее.

**Замечание 1.** Непосредственно из записи алгоритма Флойда-Уоршелла следует, что данные *a*(*i,j*) не обновляются, если *i=k* или *j=k*. Например, если *i=*1, *j=k*, то *a*(1*,j*) не обновляется:

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

Все операции *S*1(*k,k,j*), *S*1(*k,i,k*)можно исключить из алгоритма (а можно, чтобы не усложнять код, и не исключать).

**Замечание 2.** Формально между операциями одного слоя *k* существуют зависимости. Эти зависимости связаны с обновлением или использованием элементов массивов *a*(*i,j*) при *i=k* или *j=k*. Например, если *k* зафиксировано, *i=*1, то

*S*1(*k,*1*,j*): *a*(1,*j*)*=*min(*a*(1*,j*)*,a*(1*,k*)*+a*(*k,j*)), *j=*1,2,…,*k–*1,

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

// использование *a*(1*,k*) порождает антизависимости

*S*1(*k*,1*,j*)→*S*1(*k,*1*,k*), *j=*1,2,…,*k–*1,

*S*1(*k,*1*,j*): *a*(1,*j*)*=*min(*a*(1*,j*)*,a*(1*,k*)*+a*(*k,j*)), *j=k+*1,*k+*2,…,*n*,

// использование *a*(1*,k*) порождает истинные зависимости

*S*1(*k*,1*,k*)→*S*1(*k,*1*,j*), *j=k+*1,*k+*2,…,*n*.

Устранить зависимости между операциями одного слоя *k* можно введением дополнительных массивов [2]. Но, учитывая замечание 1 (элементы массивов *a*(*i,j*) при *i=k* или *j=k* не обновляются), этого делать не стоит: просто в дальнейших исследованиях перейдем к алгоритму, в котором операции при *i=k* и операции при *j=k* не выполняются (или выполняются, но зависимости, возникающие между операциями одного слоя *k*, игнорируются).

Таким образом, для фиксированного *k* все операции фактически используют данные, вычисленные на предыдущем (*k*–1)-м шаге, и операции можно выполнять в произвольном порядке (можно считать, что все операции при фиксированном *k* не зависят друг от друга).

Рассмотрим зависимости и векторы зависимостей алгоритма с учетом сделанных замечаний. 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 (все зависимости являются истинными):

* *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).

Изобразим схематично для операций *k*-го шага вторые и третьи из перечисленных зависимостей. Зависимости порождаются элементами *k*-й строки и *k*-го столбца массив *a* (вычисленными на (*k*–1)-м шаге):





*k*

**Блочный алгоритм с двумерными блоками**

Выделим блоки вычислений. Размеры блоков будем обозначать *r*1, *r*2, *r*3. Блоки вычислений рассматриваются как макрооперации, поэтому они должны выполняться атомарно, независимо друг от друга. С учетом зависимостей алгоритма, такие блоки проще всего задать, если положить *r*1*=*1, *r*2 и *r*3 – параметры. Всего будет *Q*1×*Q*2×*Q*3 двумерных (2D) блоков, где *Q*1*=n*, *Q*2*=*, *Q*3*=*. Основная часть алгоритма Флойда-Уоршелла с выделенными двумерными (2D) блоками имеет следующий вид:

do *k =* 1*, n*

do *igl =* 0, *Q*2–1 // *igl* – номера частей, на которые при формировании блоков

разбивается множество значений параметра *i*

do *jgl =* 0, *Q*3*–*1 // *jgl* – номера частей, на которые при формировании блоков

разбивается множество значений параметра *j*

// Начало блока Tile(*k*,*igl*,*jgl*)

do *i =* 1+*iglr*2, min((*igl+*1)*r*2, *n*)

do *j =* 1+*jglr*3, min((*jgl+*1)*r*3, *n*)

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

enddo

enddo

// Конец блока Tile(*k*,*igl*,*jgl*)

enddo

enddo

enddo

В алгоритме с 2D блоками сохраняется последовательное выполнение шагов *k*. На каждом шаге порядок выполнения операций изменился, но, как уже отмечалось, при фиксированном *k* операции можно выполнять в произвольном порядке.

**Блочный алгоритм с трехмерными блоками**

Блочный алгоритм Флойда-Уоршела с трехмерными (3D) блоками впервые предложен в работе [3]. Выделим *Q*×*Q*×*Q* трехмерных (3D) 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.

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.

Можно воспользоваться прежним рисунком, но считать, что выделена блочная итерация *kgl*.





*kgl*

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).

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).

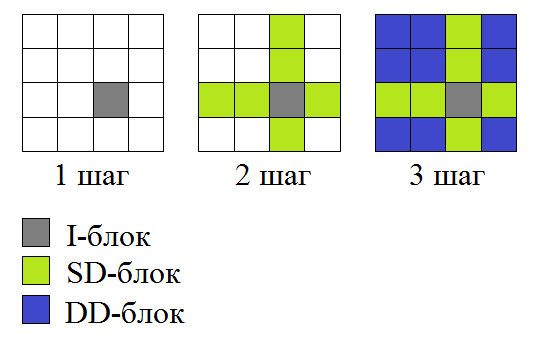
3. Остальные блоки Tile(*kgl*,*igl*,*jgl*), 0≤*igl*,*jgl*≤*Q*–1, *igl*≠*kgl*, *jgl*≠*kgl*, зависят от вычислений блоков ведущих строк и столбцов. Для вычислений этих блоков нужны их собственные элементы, а также элементы соответствующих блоков ведущих строки и столбца. Эти блоки вычисляются на блочном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

Опишем шаги блочной итерации *kgl* (см. также поясняющий рисунок ниже):

1. Производятся вычисления ведущего блока (I-блока). Фактически выполняется обычный точечный алгоритм Флойда-Уоршела; в итоге вычислений 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.

3. Производятся вычисления оставшихся блоков (DD-блоков). При обращении к элементам блоков ведущей строки и ведущего столбца происходит обращение к их последней версии.



Отметим, что результаты промежуточных вычислений точечного и блочного алгоритмов Флойда-Уоршела не совпадают.

Действительно, в точечном алгоритме Флойда-Уоршела производятся вычисления

*ak*(*i*,*j*) *=* min(*a k−*1(*i,j*)*, a k−*1(*i,k*) *+ a k−*1(*k,j*)),

где верхний индекс означает номер итерации, на которой вычислен элемент. В блочном алгоритме на итерации *kgl* в блоках ведущей строки и ведущего столбца производятся соответственно вычисления

*ak*(*i*,*j*) *=* min(*a k−*1(*i,j*)*, a k−*1(*i,k*) *+ a k*1(*k,j*)),

*ak*(*i*,*j*) *=* min(*a k−*1(*i,j*)*, a k*1(*i,k*) *+ a k−*1(*k,j*)),

где 1+*kglr*≤*k*≤(*kgl+*1)*r*, *k*1=(*kgl+*1)*r*. В DD-блоках производятся вычисления

*ak*(*i*,*j*) *=* min(*a k−*1(*i,j*)*, a k*1(*i,k*) *+ a k*1(*k,j*)).

Тем не менее, блочный алгоритм Флойда-Уоршелла приводит к корректному результату, т.е. полученная в итоге матрица будет представлять собой матрицу кратчайших расстояний между всеми парами вершин. Это утверждение верно, так как имеет место

**Теорема** [4]**.** Пусть *ak*(*i*,*j*) вычисляется следующим образом:

*ak*(*i*,*j*) *=* min(*a k−*1(*i,j*)*, a k*1(*i,k*) *+ a k*2(*k,j*)),

где *k*–1≤*k*1≤*n*, *k*–1≤*k*2≤*n*. Тогда алгоритм Флойда-Уоршелла правильно вычислит все пары кратчайших путей.

Основная часть алгоритма Флойда-Уоршелла с выделенными 3D блоками имеет следующий вид [5]:

do *kgl =* 0, *Q*–1

// Начало вычисления I-блока Tile(*kgl*,*kgl*,*kgl*)

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

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

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

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

enddo

enddo

enddo

// Конец вычисления I-блока Tile(*kgl*,*kgl*,*kgl*)

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

do *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

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

do *i =* 1+*kglr*, min((*kgl+*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

enddo

// Конец вычисления SD-блоков Tile(*kgl*,*kgl*,*jgl*)

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

do *igl =* 0, *Q*–1 (*igl*≠*kgl*)

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

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

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

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

enddo

enddo

enddo

enddo

// Конец вычисления SD-блоков Tile(*kgl*,*igl*,*kgl*)

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

do *igl =* 0, *Q*–1 (*igl*≠*kgl*)

do *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

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

enddo

enddo

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

enddo(*kgl*)

**Замечание 3.** 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 тайл через TileDD(*kgl*,*igl*,*jgl*) и запишем его явный вид:

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

do *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

enddo

enddo

Запишем основную часть 3D блочного алгоритма Флойда-Уоршелла, используя только уровень блоков:

do *kgl =* 0, *Q*–1

Tile(*kgl*,*kgl*,*kgl*)

do *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

Tile(*kgl*,*kgl*,*jgl*)

enddo

do *igl =* 0, *Q*–1 (*igl*≠*kgl*)

Tile(*kgl*,*igl*,*kgl*)

enddo

do *igl =* 0, *Q*–1 (*igl*≠*kgl*)

do *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

TileDD(*kgl*,*igl*,*jgl*)

enddo

enddo

enddo(*kgl*)

**Реализация на GPU блочного алгоритма с 2D-блоками**

В рассмотренном ранее блочном алгоритме с 2D блоками сохраняется последовательное выполнение шагов *k*. Алгоритм обладает естественным параллелизмом в пределах одной итерации, 2D-блоки вычислений могут выполняться на GPU независимо:

do *k =* 1*, n*

запуск ядра// всего – *n* запусков

dopar *igl =* 0, *Q*2–1

dopar *jgl =* 0, *Q*3*–*1

Bl(*igl*,*jgl*) // вычисления блока Bl(*igl*,*jgl*) – это вычисления тайла Tile(*k*,*igl*,*jgl*)

enddopar

enddopar

enddo

Потоки в блоке вычислений проще всего организовать таким образом, что один поток вычисляет один элемент матрицы:

Bl(*igl*,*jgl*):

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

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

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

enddopar

enddopar

Так как единственное место GPU, где может храниться матрица, является глобальная память, то на каждой итерации *k* (*k=*1*,*2,…,*n*) необходима запись всех обновленных элементов матрицы в глобальную память. Из глобальной памяти считываются все нужные данные, подсчитанные на предыдущей итерации.

Впервые данная реализация предложена в работе [9].

**Реализации на GPU блочного алгоритма с 3D-блоками**

В работе [10] реализован для 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*.

Назовем эту реализацию алгоритма с 3D блоками базовой. На каждой блочной итерации *kgl* базовой реализации алгоритма требуется три запуска ядра:

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 назад 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.

Псевдокод описанной реализации:

do *kgl =* 0, *Q*–1

запуск ядра // вычисления I-блока

BlI(?) // блок BlI(?) – это тайл Tile(*kgl*,*kgl*,*kgl*)

запуск ядра // вычисления SD-блоков

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

BlSDR(*jgl*) // блок BlSDR(*jgl*) – это тайл Tile(*kgl*,*kgl*,*jgl*) ведущей строки

enddopar

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

BlSDC(*igl*) // блок BlSDC(*igl*) – это тайл Tile(*kgl*,*igl*,*kgl*) ведущего столбца

enddopar (3DФУ)

запуск ядра // вычисления DD-блоков

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

BlDD(*igl*,*jgl*) // блок BlDD(*igl*,*jgl*) – это тайл TileDD(*kgl*,*igl*,*jgl*)

enddopar

enddopar

enddo (*kgl*)

BlDD(*igl*,*jgl*):

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

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

Thr(*i*,*j*):

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

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

enddo

enddopar

enddopar

**Замечание 4.** При вычислении DD-блоков (блоков BlDD(*igl*,*jgl*)) в разделяемой памяти достаточно хранить только по две матрицы: потокам не понадобится обращаться к элементам блока, за которые отвечают другие потоки – нужен будет только свой элемент и элементы двух SD-блоков. Каждый поток может хранить элемент, за который он отвечает, в своей регистровой памяти. Назовем данную оптимизацию регистровой.

**Замечание 5.** Псевдокод реализации, в которой при запуске DD-блоков текущего блочного слоя производятся вычисления и ведущего блока (I-блока) следующего блочного слоя:

do *kgl =* 0, *Q*–1

if *kgl=*0, then запуск ядра // вычисления I-блока

BlI(0?) // блок BlI(0?) – это тайл Tile(0,0,0)

запуск ядра // вычисления SD-блоков

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

BlSDR(*jgl*) // блок BlSDR(*jgl*) – это тайл Tile(*kgl*,*kgl*,*jgl*) ведущей строки

enddopar

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

BlSDC(*igl*) // блок BlSDC(*igl*) – это тайл Tile(*kgl*,*igl*,*kgl*) ведущего столбца

enddopar

запуск ядра // вычисления DD-блоков

dopar *igl =* 0, *Q*–1 (*igl*≠*kgl*)

dopar *jgl =* 0, *Q*–1 (*jgl*≠*kgl*)

if *igl=kgl+*1, *jgl=kgl+*1 then BlDDI(*kgl*?) // блок BlDDI(*kgl*?) – это объединение

тайлов TileDD(*kgl*,*kgl+*1,*kgl+*1) и Tile(*kgl+*1,*kgl+*1,*kgl+*1)

else BlDD(*igl*,*jgl*) // блок BlDD(*igl*,*jgl*) – это тайл TileDD(*kgl*,*igl*,*jgl*)

enddopar

enddopar

enddo (*kgl*)

Здесь на каждом, кроме первого, блочном слое происходит два, а не три запуска ядра.

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; после выполнения каждой стадии необходима синхронизация потоков одного блока. Уменьшение размеров разделяемой памяти, необходимой блоку в каждый момент времени, позволяет получить заметный выигрыш в производительности, так как уменьшая размер разделяемой памяти, используемой в блоке, мы позволяем мультипроцессору запустить большее количество блоков одновременно.

В работе [11] предложена и реализована на графическом процессоре модификация алгоритма Флойда-Уоршелла с 3D блоками размера (*l+*1)*r*×*r*×*r*, где *l* – некоторый неотрицательный целочисленный параметр. 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*.

Напомним, реализация алгоритма (3DФУ) называется регистровой, если при вычислении блоков BlDD(*igl*,*jgl*)) в разделяемой памяти хранятся элементы *a*(*k,j*) блока ведущей строки и элементы блока ведущего столбца *a*(*i,k*), а элементы *a*(*i*,*j*) приватизируются потоками Thr(*i*,*j*) и хранятся в регистрах.

Приведем некоторые характеристики регистровой реализации ядра вычисления DD-блоков в предположении, что в каждом блоке используется максимально возможное число 32*×*32 потоков (*r=*32). Будем считать, что в одном мультипроцессоре могут выполняться 2048 потоков, мультипроцессор имеет 64К (65 536) регистров и 48КБ (49 152 байта) разделяемой памяти, может одновременно обрабатывать не более 8 блоков и не более 64 варпов. Будем рассматривать и лругой выриант (Тесла K80, например): мультипроцессор имеет 128К регистров, 128КБ (48КБ на блок) разделяемой памяти, может одновременно обрабатывать 16 блоков, 64 варпа.

Для вычислений потоков Thr(*i*,*j*) каждого из запускаемых блоков требуются три матрицы размера 32*×*32: «свои» (обновляемые на каждой итерации *k*) элементы *a*(*i*,*j*), элементы *a*(*k,j*) блока ведущей строки и элементы блока ведущего столбца *a*(*i,k*). Элементы *a*(*i*,*j*) приватизируются потоками Thr(*i*,*j*), поэтому их будем хранить в регистрах; всего на один блок требуется 1024 регистра (1 регистр на поток). Элементы *a*(*i,k*) и *a*(*k,j*) будем хранить в разделяемой памяти; всего на один блок требуется 2*×*1024*×*4*=*8КБ разделяемой памяти (2*×*1024 – два массива по 1024 элементов, *×*4 означает, что используется тип float – 4 байта на слово).

В блоке все 32 варпа могут выполняться (логически) параллельно, поэтому максимально возможное количество 64 варпа на мультипроцессор обеспечивается двумя одновременно выполняемыми блоками. Суммарно два блока требуют 2048 регистров и 16КБ разделяемой памяти. Можно обойтись без многостадийного выполнения; на момент написания работы [5] объем разделяемой памяти составлял 16КБ, поэтому, с учетом необходимости хранения необходимых параметров реализуемого алгоритма, требовалось многостадийное выполнение. (В работе по каким-то причинам рассматривается четыре стадии, хотя, вроде бы, достаточно двух. Возможно, в этом случае, точно всегда будут идти вычисления, не будет ситуации, что все вычисляемые на одном мультипроцессоре блоки заняты обменами.)

BlDD(*igl*,*jgl*):

dopar *igl2 =* 0, *r*–1

dopar *jgl2 =* 0, *r*–1

Thr(*igl2*,*jgl2*):

*i=*1+*iglr*+*igl2*

*j=*1+*jglr*+*jgl2*

if *i≤n*, *j≤n*

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

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

enddo

endif

enddopar(*jgl2*)

enddopar(*igl2*)

Далее такая запись блока BlDD(*igl*,*jgl*) будет названа (BlDDThr(1×1)). Эта запись блока вычислений эквивалентна записи (BlDDThr(*i*,*j*)), но нумерация координат узлов сетки потоков начинается с нуля. Заметим, что обозначения *igl2*, *jgl2* имеют здесь самостоятельное значение и напрямую не связаны с тайлингом второго уровня.

(По сути это очень близко к только что описанному алгоритму; проверяется, с учетом примера статьи по бродкасту и простр. лок., эффективно ли хранить *a*(*i,k*) в регистрах) Пусть каждый BlDD(*igl*,*jgl*) имеет размеры 32*×*32*×*32 (*r=*32). Для вычислений потоков Thr(*igl2*,*jgl2*), 0*≤igl2*,*jgl2≤*31, каждого из запускаемых блоков требуются три матрицы размера 32*×*32. Элементы *a*(*i*,*j*), *i=*1+32*igl*+*igl2*, *j=*1+32*jgl*+*jgl2*, обновляются на каждой итерации *k*; в одном потоке *i* и *j* фиксировано. Элементы массива *a*(*i*,*j*) приватизируются потоками, поэтому их будем хранить в регистрах; для этого на один блок требуется 1024 регистра (32 регистра на поток). Элементы *a*(*i,k*) также будем хранить в регистрах (не происходит ни приватизации, ни бродкаста, ни пространственной локальности; *a*(*i,k*) возможно, лучше хранить в разделяемой памяти, но возможны конфликты (например, *a*(1*,*1) и *a*(1*,*2) могут быть в одном банке)); на один блок требуется 32(регистра на поток)*×*1024(число потоков)*=*32К. Всего на один блок требуется 1К+32К*=*33К регистров (хватает, сейчас мультипроцессор имеет 64К (65 536) регистров; регистры делятся поровну между потоками; если в блоке используется 1024 потоков, то на один поток приходится 64 регистра). Элементы *a*(*k,j*) будем хранить в разделяемой памяти, так как происходит бродкаст (так как элементы массива только считываются, возможно, эффективна и константная память); всего на один блок требуется 1024*×*4*=*4КБ разделяемой памяти. В блоке всего 32 варпа. Одновременно могут обрабатываться 8 блоков, но в раccматриваемом случае одновременно может обрабатываться только 1 блок (он требует 33К из 64К регистров, а вот если 128К регистров, то ситуация другая). Максимальное возможное (и желательное?) количество варпов на мультипроцессор – 64; можно обрабатывать 1 (или 2 блока). Многостадийное выполнение не требуется.

То, что далее, для лекции не нужно.

?Нужны тайлы с приватизированными массивами (нумерация не глобальная, а в каждом блоке начинать с нуля или единицы)?

**Реализации, использующие тайлинг второго уровня**

**Тайлинг второго уровня** (случай блоков BlDD(*igl*,*jgl*))**.** В одном блоке вычислений может быть до 1024 потоков, поэтому рассмотренный ранее способ организации потоков вычислений годится только для *r≤*32. Для организации потоков вычислений в случае *r>*32 (например, *r=*64) произведем повторное разбиение (тайлинг уровня 2) итераций блоков вычислений BlDD(*igl*,*jgl*). Обозначим через *ri*,2, *rj*,2 и *rk*,2 размеры тайлов второго уровня.

Рассмотрим тайлинг, для которого цикл *k* блока вычислений является локальным не разбиваемым: *rk*,2=*r*. Циклы *i* и *j* разбиваются, числа *Qi*,2 и *Qj*,2 задают количество итераций в новых циклах с параметрами *igl2* и *jgl2* соответственно. Так как *rk*,2*=r*, то *Qk*,21, цикл do *kgl2=*0,*Qk*,2–1 с параметром *kgl2* является вырожденным, *kgl2=*0.

После разбиения и перестановки циклов получим следующее представление блока вычислений BlDD(*igl*,*jgl*):

dopar *igl2 =* 0, *Qi*,2–1

dopar *jgl2 =* 0, *Qj*,2–1

Thr(*igl2*,*jgl2*):

do *i =* 1+*iglr*+*igl2ri*,2, min(*iglr*+(*igl2+*1)*ri*,2, (*igl+*1)*r*, *n*)

do *j =* 1+*jglr*+*jgl2rj*,2, min(*jglr*+(*jgl2+*1)*rj*,2, (*jgl+*1)*r*, *n*)

do *k =* 1+*kglr*, min((*kgl+*1)*r*, *n*) (BlDDThr(*ri*,2×*rj*,2))

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

enddo

enddo

enddo

enddopar(*jgl2*)

enddopar(*igl2*)

Один поток Thr(*igl2*,*jgl2*) включает вычисления *ri*,2×*rj*,2 «своих» итераций *i* и *j*.

Рассмотрим реализацию блоков вычислений BlDD(*igl*,*jgl*) для случая *r=*64, *ri*,2*=rj*,2*=*2. Как и ранее, будем считать, что в одном мультипроцессоре могут выполняться 2048 потоков, мультипроцессор имеет 64К (65 536) регистров и 48КБ (49 152 байта) разделяемой памяти, может одновременно обрабатывать не более 8 блоков и не более 64 варпов. Другой выриант: мультипроцессор имеет 128К регистров, 128КБ (48КБ на блок) разделяемой памяти, может одновременно обрабатывать 16 блоков, 64 варпа.

Для вычислений потоков Thr(*igl2*,*jgl2*), 0*≤igl*,*jgl22≤*31, каждого из запускаемых блоков требуются три матрицы размера 64*×*64: «свои» (обновляемые на каждой итерации *k*) элементы *a*(*i*,*j*), элементы *a*(*k,j*) блока ведущей строки и элементы блока ведущего столбца *a*(*i,k*). Элементы *a*(*i*,*j*) приватизируются потоками Thr(*i*,*j*), поэтому их будем хранить в регистрах (4 регистра на поток); всего на один блок требуется 4096*=*4К регистров. Элементы *a*(*i,k*) и *a*(*k,j*) будем хранить в разделяемой памяти; всего на один блок требуется 2*×*4096*×*4*=*32КБ разделяемой памяти (2*×*4096– два массива по 4096 элементов, *×*4 означает, что используется тип float – 4 байта на слово). В блоке 1024 потока – это 32 варпа. В блоке все 32 варпа могут выполняться (логически) параллельно, поэтому максимально возможное количество 64 варпа на мультипроцессор обеспечивается двумя одновременно выполняемыми блоками. Суммарно два блока требуют 8К регистров и 64КБ разделяемой памяти. Требуется двухстадийное выполнение. Если мультипроцессор имеет 128К регистров и 128КБ разделяемой памяти, то можно обойтись без многостадийного выполнения.

Отметим преимущества и недостатки случая блоков итераций 64×64×64 (*r=*64) перед случаем блоков итераций 32×32×32 (*r=*32). Преимущества: вдвое меньше блочных слоёв (запусков ядра), а значит, (вдвое?) меньше обменов посредством обращения к глобальной памяти. Недостатки: требуется двухстадийное выполнение, есть накладные расходы на организацию тайлов второго уровня.

Пусть в (BlDDThr(*ri*,2×*rj*,2)) *ri*,2*=*1 и *rj*,2*=*1 (тогда *Qi*,2*=r*, *Qj*,2*=r*):

dopar *igl2 =* 0, *r*–1

dopar *jgl2 =* 0, *r*–1

Thr(*igl2*,*jgl2*):

do *i =* 1+*iglr*+*igl2*, min(*iglr*+(*igl2+*1), (*igl+*1)*r*, *n*)

do *j =* 1+*jglr*+*jgl2*, min(*jglr*+(*jgl2+*1), (*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

enddo

enddo

enddopar(*jgl2*)

enddopar(*igl2*)

**Замечание 6.** Если *r* нацело делится на *ri*,2, то в алгоритме (BlDDThr(*ri*,2×*rj*,2)) проверка выхода итераций по *i* за пределы блока не требуется, т.е. под знак min в этом случае не включается выражение (*igl+*1)*r*. Аналогично и для итераций по *j*. Если *n* нацело делится на *r*, то в алгоритме (BlDDThr(*ri*,2×*rj*,2)) проверка выхода итераций по *i* и по *j* за пределы области вычислений не требуется, т.е. под знак min в этом случае не включается *n*.

Так как циклы с параметрами *i* и *j* содержат не более одной итерации, то, с учетом замечания 6, получим следующее представление блока вычислений BlDD(*igl*,*jgl*) в случае *ri*,2*=*1, *rj*,2*=*1:

dopar *igl2 =* 0, *r*–1

dopar *jgl2 =* 0, *r*–1

Thr(*igl2*,*jgl2*):

*i=*1+*iglr*+*igl2*

*j=*1+*jglr*+*jgl2*

if *i≤n*, *j≤n*

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

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

enddo

endif

enddopar(*jgl2*)

enddopar(*igl2*)

Заметим, что эта запись блока вычислений эквивалентна записи (BlDDThr(*i*,*j*)), но нумерация координат узлов сетки потоков начинается с нуля.

При проведении экспериментов вычисления надо оптимизировать. Например: *i=*1+*iglr*+*igl2* вычислить один раз, а не вычислять многократно в цикле; *n* должно нацело делиться на *r*, *ri*,2, *rj*,2 (тогда, в частности, не потребуется проверка if *i≤*min((*igl+*1)*r*, *n*), *j≤*min((*jgl+*1)*r*, *n*)).

Если *ri*,2*=*1 или *rj*,2*=*1, то соответственно получим:

dopar *igl2 =* 0, *r*–1

dopar *jgl2 =* 0, *Qj*,2–1

Thr(*igl2*,*jgl2*):

*i=*1+*iglr*+*igl2*

if *i≤ n*

do *j =* 1+*jglr*+*jgl2rj*,2, min(*jglr*+(*jgl2+*1)*rj*,2, (*jgl+*1)*r*, *n*)

do *k =* 1+*kglr*, min((*kgl+*1)*r*, *n*) (BlDDThr(1×*rj*,2))

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

enddo

enddo

endif

enddopar(*jgl2*)

enddopar(*igl2*)

dopar *igl2 =* 0, *Qi*,2–1

dopar *jgl2 =* 0, *r* –1

Thr(*igl2*,*jgl2*):

*j=*1+*jglr*+*jgl2*

if *j≤n*)

do *i =* 1+*iglr*+*igl2ri*,2, min(*iglr*+(*igl2+*1)*ri*,2, (*igl+*1)*r*, *n*)

do *k =* 1+*kglr*, min((*kgl+*1)*r*, *n*) (BlDDThr(*ri*,2×1))

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

enddo

enddo

endif

enddopar(*jgl2*)

enddopar(*igl2*)

Если *ri*,2*=*1, *rj*,2*=r* (тогда *jgl2 =* 0), то получим

dopar *igl2 =* 0, *r*–1

Thr(*igl2*): // Thr(*igl2*) – это Thr(*igl2*,0)

*i=*1+*iglr*+*igl2*

if *i≤n*

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

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

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

enddo

enddo

endif

enddopar(*igl2*)

Если *ri*,2*=*1, *rj*,2*=r*, *n* нацело делится на *r*, то получим следующее представление блока вычислений BlDD(*igl*,*jgl*):

dopar *igl2 =* 0, *r*–1

Thr(*igl2*): // Thr(*igl2*) – это Thr(*igl2*,0)

*i=*1+*iglr*+*igl2*

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

do *k =* 1+*kglr*, (*kgl+*1)*r* (BlDDThrZ(1×*r*))

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

enddo

enddo

enddopar(*igl2*)

Рассмотрим реализацию этих блоков вычислений для двух случаев: *r=*32 и *r=*64.

Как и ранее, будем считать, что в одном мультипроцессоре могут выполняться 2048 потоков, мультипроцессор имеет 64К (65 536) регистров и 48КБ (49 152 байта) разделяемой памяти, может одновременно обрабатывать не более 8 блоков и не более 64 варпов. Другой выриант: мультипроцессор имеет 128К регистров, 128КБ (48КБ на блок) разделяемой памяти, может одновременно обрабатывать 16 блоков, 64 варпа.

Пусть каждый BlDD(*igl*,*jgl*) имеет размеры 32*×*32*×*32 (*r=*32). Для вычислений потоков Thr(*igl2*), 0*≤igl2≤*31, каждого из запускаемых блоков требуются три матрицы размера 32*×*32. Элементы *a*(*i*,*j*), *i=*1+32*igl*+*igl2*, обновляются на каждой итерации *k*; в одном потоке *i* фиксировано, *j* принимает 32 значения. Элементы массива *a*(*i*,*j*) приватизируются потоками, поэтому их будем хранить в регистрах; для этого на один блок требуется 1024 регистра (32 регистра на поток). Элементы *a*(*i,k*) также приватизируются потоками, их будем хранить в регистрах; на один блок требуется 1024 регистра (32 регистра на поток). Всего на один блок требуется 2*×*1024*=*2К регистров. Элементы *a*(*k,j*) будем хранить в разделяемой памяти; всего на один блок требуется 1024*×*4*=*4КБ разделяемой памяти. В блоке всего 1 варп. Если одновременно могут обрабатываться 8 блоков, то выполняется 8 варпов, если одновременно могут обрабатываться 16 блоков, то выполняется 16 варпов. Максимальное возможное (и желательное) количество варпов на мультипроцессор – 64. Суммарно 8 блоков требуют 8*×*2К*=*16К регистров и 8*×*4КБ*=*32КБ, многостадийное выполнение не требуется. Суммарно 16 блоков требуют 16*×*2К*=*32К регистров и 16*×*4КБ*=*64КБ. Если мультипроцессор имеет 64КБ разделяемой памяти, то требуется двухстадийное (с учетом необходимости хранения необходимых параметров реализуемого алгоритма) выполнение. Если мультипроцессор имеет 128КБ разделяемой памяти, то многостадийное выполнение не требуется.

Пусть теперь каждый BlDD(*igl*,*jgl*) имеет размеры 64*×*64*×*64 (*r=*64). Для вычислений потоков Thr(*igl2*), 0*≤igl2≤*63, каждого из запускаемых блоков требуются три матрицы размера 64*×*64. В блоке 64 потока – это 2 варпа. Элементы *a*(*i*,*j*), *i=*1+64*igl*+*igl2*, обновляются на каждой итерации *k*; в одном потоке *i* фиксировано, *j* принимает 64 значения. Элементы массива *a*(*i*,*j*) приватизируются потоками, поэтому их будем хранить в регистрах; для этого на один блок требуется 4096 регистров (64 регистра на поток). Элементы *a*(*i,k*) также приватизируются потоками, их будем хранить в регистрах; на один блок требуется 4096 регистров (64 регистра на поток). Всего на один блок требуется 2*×*4096*=*8К регистров. Элементы *a*(*k,j*) будем хранить в разделяемой памяти; всего на один блок требуется 4096*×*4*=*16КБ разделяемой памяти. Суммарно 8 блоков требуют 8*×*8К*=*64К регистров и 8*×*16КБ*=*128КБ разделяемой памяти. Суммарно 16 блоков требуют 16*×*8К*=*128К регистров и 16*×*16КБ*=*256КБ. Если мультипроцессор имеет 64К регистров и 48КБ разделяемой памяти, то одновременно могут обрабатываться 7 блоков – это 7*×*8К*=*56К регистров, 7*×*16КБ*=*112КБ разделяемой памяти, одновременное выполнение 7*×*2*=*14 варпов (8 блоков – это 8*×*8К*=*64К регистров, предел, но регистры требуются не только для хранения элементов массивов). Требуется трех(?четырех)стадийное выполнение. Если мультипроцессор имеет 128К регистров и 128КБ разделяемой памяти, то одновременно могут обрабатываться 15 блоков – это 15*×*8К*=*120К регистров, 15*×*16КБ*=*240КБ разделяемой памяти, одновременное выполнение 15*×*2*=*30 варпов. Требуется двухстадийное выполнение. Максимальное возможное (и желательное) количество варпов на мультипроцессор – 64.

**Об улучшении локальности реализаций**

Приведем сведения, которые могут быть полезны для исследования локальности вычислений блока (BlDDThr(1×1)) (см. также файл «GPGPU»).

Как уже отмечалось, внутри пула варп потоки выполняются синхронно, поэтому при наличии временной (broadcast) или пространственной локальности никакой синхронизации потоков организовывать не надо.

Пусть потоки Thr(*i*,*j*) блока имеют следующий вид:

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

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

enddo

Если в потоках Thr(*i*,*j*) пула варп фиксировано *i*, а *j* близко расположены, то при синхронном выполнении потоков имеем:

на вхождении *a*(*i*,*j*) потоки полуварпа запрашивают одновременно близко расположенные в памяти элементы *a*(*i*,*j*), один поток запрашивает на всех итерациях *k* один тот же элемент,

на вхождении *a*(*i,k*) потоки полуварпа запрашивают одновременно один тот же элемент (broadcast), один поток запрашивает на соседних итерациях *k* соседние элементы одной строки;

на вхождении *a*(*k,j*) потоки полуварпа запрашивают одновременно близко расположенные в памяти элементы (если элементы массива *a*(*k,j*) расположены в разделяемой памяти, то обращение в один банк памяти, а значит конфликты?), один поток запрашивает на соседних итерациях *k* элементы разных строк.

**Список использованных источников**

* 1. [Котов В.М., Соболевская Е.П., Толстиков А.А. Алгоритмы и](http://www.fpmi.bsu.by/ru/?guid=13433) структуры данных. Учебное пособие. Минск: БГУ, 2011. 267 с.
  2. Mullapudi R.T., Bondhugula U. Tiling for dynamic scheduling // Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, Vienna, Austria, 2014.
  3. Venkataraman G., Sahni S., Mukhopadhyaya S. A blocked all-pairs shortest-paths algorithm // J. Exp. Algorithmics. 2003. Vol. 8. P. 2.2.
  4. Park J., Penner M., Prasanna V.K. Optimizing graph algorithms for improved cache herformance // IEEE Transactions on Parallel and Distributed Systems. – September. 2004. Vol. 15, № 9. P. 769–782.
  5. Lund B.D. Smith J.W. A multi-stage cuda kernel for floyd-warshall // CoRR abs/1001.4108. 2010.
  6. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2019. 232 с.
  7. [Электрон. ресурс – <https://edufpmi.bsu.by/> Параллельные и распределенные вычисления/ Лекции], лекция «GPGPU».
  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.
  11. Лиходед Н.А., Сипейко Д.С. Обобщенный блочный алгоритм Флойда-Уоршелла // **Журнал Белорусского государственного университета. Математика. Информатика.** 2019, № 3. С. 84–92.