УДК 519.7; 519.67; 004.021; 004.043

ИССЛЕДОВАНИЕ МОДЕЛЕЙ РЕАЛИЗАЦИИ ВОЛНОВОГО АЛГОРИТМА ДВИЖЕНИЯ РОБОТА ДЛЯ АРХИТЕКТУРЫ NVIDIA В ТЕХНОЛОГИИ CUDA[1]

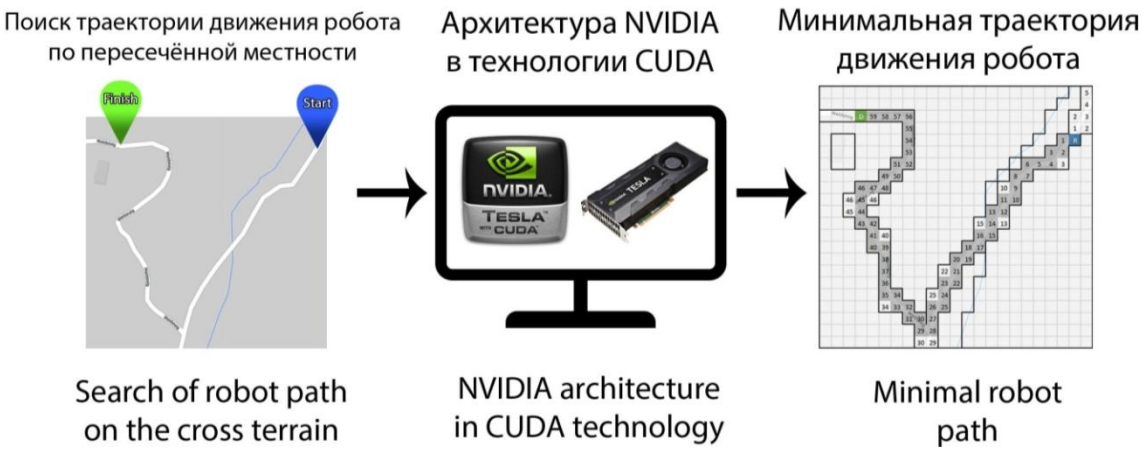
Статья поступила в редакцию 05.12.2018, в окончательном варианте – 26.12.2018.

Курочкин Михаил Александрович, Санкт-Петербургский политехнический университет Петра Великого (СПбПУ), 195251, Российская Федерация, г. Санкт-Петербург, ул. Политехническая, 29,
кандидат технических наук, доцент, e-mail: kurochkin.m@gmail.com
Крыштапович Виктор Сергеевич, Санкт-Петербургский политехнический университет Петра Великого (СПбПУ), 195251, Российская Федерация, г. Санкт-Петербург, ул. Политехническая, 29,
студент, e-mail: kry127@yandex.ru

В статье рассматриваются методы построения пути робота для архитектуры NVIDIA в технологии CUDA для трех моделей памяти (глобальная, локальная и текстурная). Построение пути реализовано волновым алгоритмом. Разработан метод распараллеливания волнового алгоритма и его программная реализация на суперкомпьютере «Политехнический» (Санкт-Петербургский политехнический университет). Приведены результаты моделирования эффективности использования каждой модели памяти для разной плотности расположения запретных зон для прохождения роботом. Плотность расположения запретных зон в имитационных моделях изменялась в диапазоне 5–50 %. Приведена технология и примеры массивно-параллельной реализации приложений в рамках использования технологии CUDA. Результаты исследований будут полезны специалистам, разрабатывающим приложения для планирования (управления) движением роботов по пересеченной местности содержащей непреодолимые для роботов препятствия.

Ключевые слова: технология CUDA, распараллеливание алгоритмов, траектория движения робота, волновой алгоритм, модели памяти

Графическая аннотация (Graphical annotation)



RESEARCH OF MODELS OF IMPLEMENTATION OF THE LEE ALGORITHM FOR A ROBOT MOTION BASED ON THE NVIDIA ARCHITECTURE IN THE CUDA TECHNOLOGY

The article was received by editorial board on 05.12.2018, in the final version – 26.12.2018.

Kurochkin Mikhail A., Peter the Great St.Petersburg Polytechnic University (SPbPU), 29 Politekhnicheskaya St., St. Petersburg, 195251, Russian Federation,
Cand. Sci. (Engineering), e-mail: kurochkin.m@gmail.com
Kryshtapovich Victor S., Peter the Great St.Petersburg Polytechnic University (SPbPU), 29 Politekhnicheskaya St., St. Petersburg, 195251, Russian Federation,
student, e-mail: kry127@yandex.ru

In this paper is described the methods of robot path construction for the NVIDIA GPU architecture by the CUDA technology. There are considered three main memory models of NVIDIA architecture: global, local and texture memory. The algorithm of the robot path construction is based on the Lee algorithm. The algorithm and software implementation executes

[1] Работа выполнена при поддержке гранта РФФИ 16-29-04319. Контекстно-ориентированные методы принятия решений и планирования операций смешанной группой мобильных роботов.

on the Polytechnic supercomputer (Peter the Great St.Petersburg Polytechnic University). The document resides results of modeling the efficiency of each memory model usage depending on different density of impassable terrain (degree of impassibility) by robot. The density of the impassable terrain in simulation models varies in the range of 5-50%. Paper contains examples of the massively parallel implementation of applications by the CUDA technology. The results will be useful in the applications for planning robots movement over rough terrain with obstacles.

Key words: CUDA technology, parallel algorithms, robot trajectory, wave algorithm, memory models

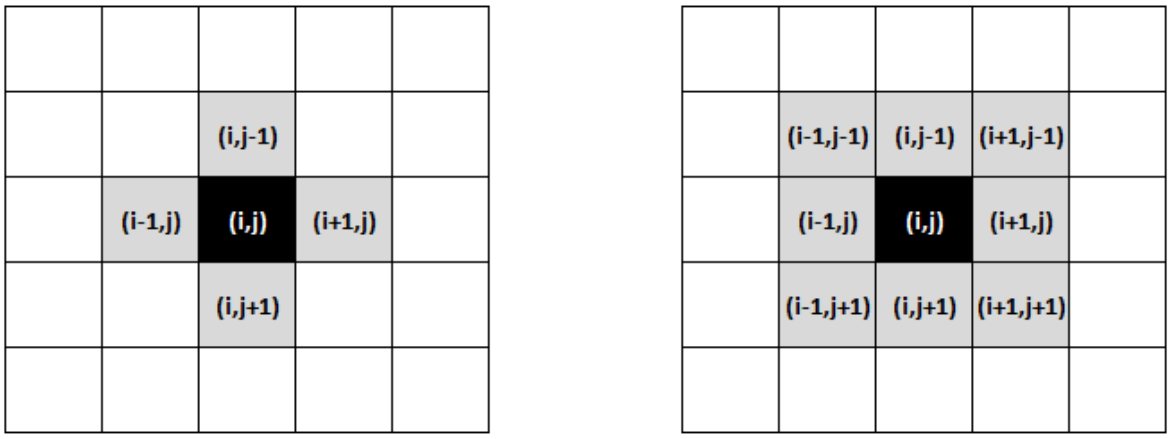
Введение. Задача построения кратчайшего пути движения робота является достаточно актуальной для планирования движения в условиях неопределенности, а также при наличии непреодолимых для роботов зон и / или границ между зонами. Управление движением группы интеллектуальных роботов сопряжено с рядом сложностей начиная с задачи целеполагания и заканчивая выработкой законов подачи управляющих воздействий на исполнительные механизмы каждого робота, обеспечивающие их движение. Алгоритмы планирования движения разрабатываются с учётом локальных (находящихся в непосредственной видимости) и глобальных (централизованно передаваемых в группе роботов) данных о динамической среде движения. При условии применения алгоритмов с учётом только локальных данных возникает пространственно-ситуационная неопределённость. Алгоритмическая сложность задачи построения пути составляет $O(n^2)$, где n – размер сетки полигона (количества ячеек в полигоне, соответствующем рассматриваемой области). Каждый цикл обновления данных глобальных или локальных данных требует корректировки траектории движения робота. Период обновления данных может составлять 1–3 с, и это обстоятельство накладывает сильные ограничения на время решения задачи. Получение решения в реальном масштабе времени с использованием одного процессора в большинстве случаев практически невозможно. В лучшем случае решением считается первый вариант построения пути. С ростом числа запретных зон (и/или границ между зонами), которые недоступны для движения робота, временны*е* затраты существенно возрастают. Одним из вариантов решения этой проблемы является использование механизма распараллеливания алгоритмов построения траектории движения робота, и выполнение их в программно-аппаратной среде обработки графических данных.

Цель настоящей работы – определение зависимости времени решения задачи для разных моделей памяти архитектуры _NVIDIA в технологии CUDA.

Общая характеристика волнового метода. Современные технологии распараллеливания классических алгоритмов класса GPGPU, таких как CUDA позволяют по-новому взглянуть на методы решения актуальных задач движения роботов в условиях неопределенности. К числу классических методов решения задачи построения пути, относится волновой алгоритм Ли [1, 3, 4].

Волновой алгоритм Ли. Волновой метод относится к классу методов «динамического программирования». Алгоритм состоит из двух этапов: «распространение волны» и «восстановление пути». Для осуществления первого этапа вводится матрица, в которой элементы сопоставляются некоторым целым числам для каждой клетки пространственной сетки, наложенной на рассматриваемую прямоугольную область. В клетку *назначения* записывается число «0». Остальные ячейки помечаются, как не пройденные волной. Ячейка «0» испускает волну первого поколения и записывает значение «1» в соседние доступные ячейки, которые ещё не были посещены. Каждое новое поколение волны: помечает ячейки как пройденные; порождает волну следующего поколения, добавляя в ячейки новой волны своё значение, увеличенное на «единицу». Процесс останавливается при выполнении одного из следующих условий: а) когда *текущее положение* (начало пути) было помечено как пройденное волной; б) когда все доступные ячейки были посещены. Если в процессе работы алгоритма начальная точка всё-таки была достигнута волной, то в массив сохраняется путь от *текущего положения* робота до клетки *назначения*.

Различают волновой алгоритм с ортогональным (алгоритм Ли) и ортогонально-диагональным распространением волны. В алгоритме Ли в качестве отношения соседства для ячейки выступают те ячейки, у которых имеется общее ребро с квадратной ячейкой, для которой ищется окрестность (окрестность фон Неймана). Во втором случае соседними ячейками являются те, которые имеют общие вершины с исходными (окрестность Мура). В наглядной форме такие окрестности представлены на рисунке 1, а на рисунке 2 приведены результаты работы волнового алгоритма.



Окрестность Неймана Окрестность Мура

Рисунок 1 – Разновидности двухмерных окрестностей в клеточных автоматах

Восстановление пути – это второй этап работы волнового алгоритма. Для его выполнения нужно поместить в *путевой список* начальную точку: *текущее положение* робота. Далее произвести несколько итераций: выбрать ячейку, соседнюю с последней в списке, т.е. такую, которая была бы помечена числом на единицу меньше, чем последняя в *путевом списке*. Продолжать процесс до тех пор, пока в *путевом списке* не окажется ячейка с пометкой «0», то есть ячейка *назначения* робота.

Левая часть рисунка 2 демонстрирует следующий приоритет выбора соседних ячеек для восстановления пути: «левая, верхняя, правая, нижняя». Правая часть рисунка 3 демонстрирует приоритет выбора соседних ячеек для восстановления пути: «верхняя, левая, нижняя, правая». Алгоритм восстановления пути представим в виде «жадного алгоритма».

Сложность вычислений волнового алгоритма в худшем случае составляет $O(n^2)$.

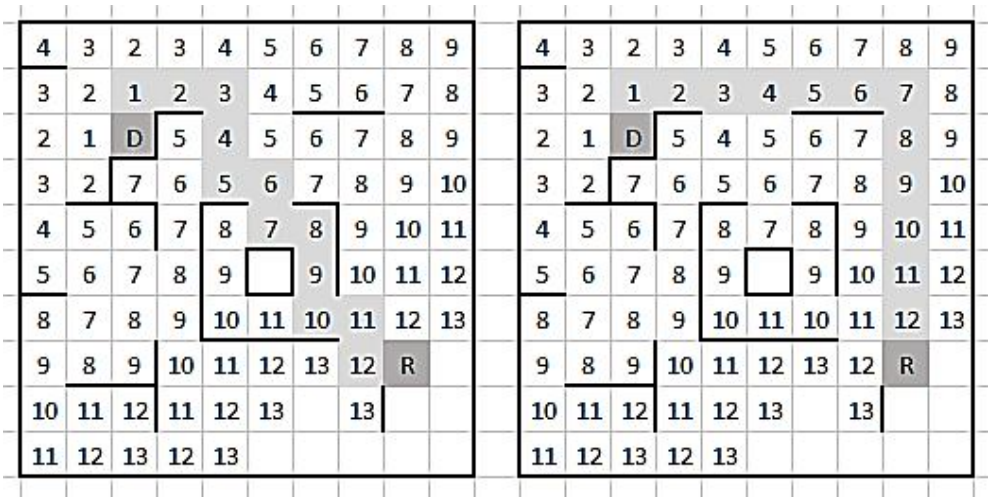


Рисунок 2 – Результат работы волнового алгоритма, толстыми жирными линиями показаны «границы» между зонами, через которые робот не может переместиться

Основные понятия технологии CUDA.

Grid в технологии CUDA – это массив размерности 1, 2 или 3, состоящий из блоков. Ограничения на количество блоков (она же размерность сетки ***gridDim***) зависит от конкретного вычислителя. Тем не менее, размер по оси *x* не превышает 2^{31}–1, а по осям *y* и *z* не превышает величину 65535.

Block в технологии CUDA – трёхмерный массив нитей. Каждый блок обладает уникальным трёхкомпонентным адресом ***blockIdx***. На размерность блока ***blockDim*** также распространяются ограничения общего правила: размер по осям *x* и *y* не может превышать 1024, а размер по оси *z* не может превышать 64. Более того, произведение размерностей блока равно количеству нитей в блоке, и данная величина не может превышать значения 2048.

Thread в технологии CUDA – элементарный параллельный процесс. Каждая нить обладает своим адресом, определяемым индексом нити в блоке ***ThreadIdx*** и индексом блока в сетке ***BlockIdx***.

Вышеперечисленные характеристики являются типичными для архитектуры Kepler, на которой и производился запуск испытаний [11].

Особенности реализации методов доступа к памяти технологии CUDA. Рассмотрим три типа памяти:

- глобальная – универсальная, ориентирована на решение любой задачи, но отличается медленной скоростью обращения;
- разделяемая – обеспечивает гарантированный доступ к данным за фиксированное количество тактов за счёт размещения в кэше L1 потокового мультипроцессора; расположена непосредственно на кристалле, что делает ее гораздо более быстрой, чем глобальная;
- текстурная – поддерживает только режим чтения данных. Физически текстурная память не отделена от глобальной – она позволяет увеличить производительность обращения к глобальной памяти за счет системы кэшей. Отличительной особенностью является оптимизация текстурного кэша для двумерной пространственной локальности (данные расположены рядом в двумерном пространстве). Варианты применения и адресации текстурной памяти приведены в [8, 13, 14].

Особенности используемой реализации волнового алгоритма. Введем понятие «доступность перехода». Переход между двумя смежными ячейками считается доступным, если разница высот этих ячеек меньше заданного порога, при этом переход считается недоступным в как одном, так и в другом направлении движения между этими ячейками. Это ограничение обусловлено характеристиками движущей платформы робота, хотя возможны и другие варианты интерпретации этого перехода. В этом случае они будут отражены в структуре данных и увеличат число возможных траекторий движения робота. В нашей постановке каждая ячейка граничит с четырьмя соседями. Поэтому для каждого из четырех переходов вычисляется свой признак доступности. В рассматриваемых примерах на рисунках 2–4 признак недоступности перехода между смежными ячейками помечен жирной линией. В постановке задачи доступность переходов относится к исходным данным.

Введём оценку карты. Она определяется количеством недоступных переходов по отношению к общему числу возможных переходов между ячейками. Для каждой ячейки имеется признак её доступности из четырёх возможных направлений (суммарно $4n^2$). При этом, так как периметр полигона всегда считается недоступным, то ячейки на краю карты не должны учитывать доступность со стороны (суммарно 4n). Таким образом, общее число анализируемых атрибутов доступности: $4n^2 – 4n$. Введём определение ***степени доступности*** карты местности.

$$\pi = \frac{A}{4n^2 - 4n}, \quad A \in [0, 4n^2 - 4n].$$

Степень недоступности μ – это отношение количества недоступных (за исключением периметра) переходов B к общему числу переходов.

$$\mu = \frac{B}{4n^2 - 4n}, \quad B \in [0, 4n^2 - 4n].$$

Так как $A + B = 4n^2 - 4n$, то общее число контролируемых признаков доступности, а связь между степенью доступности и недоступности определяется формулой $\pi + \mu = 1$.

Постановка задачи. Дано: квадратный полигон P(nxn). Координаты начального и конечного положения робота – Pst(x0, y0), Pfin(xf, yf).

Необходимо построить траекторию L(x0, y0; …..xk, yk;… xf, yf), соединяющую точки Pst(x0, y0) и Pfin(xf, yf), и исследовать время решения задачи на ПЭВМ для трех типов памяти: глобальной, разделяемой и текстурной.

Описание алгоритма построения пути

Входные данные

1. Размерность задачи

`n: Integer`

2. Текущее положение робота

`x0, y0: Integer`, $1 \le x_0, y_0 \le n$

3. Координаты назначения

`xf, yf: Integer`, $1 \le x, y \le n$

4. Признаки доступности, вычисленные на основе карты высот

`array [1..n] of array [1..n] of array[1..4] of Boolean`

5. Интервал обновления исходных данных (в секундах)

`deltaT = 0.4 : Real`

Необходимо найти не позднее, чем за время обновления данных deltaT последовательность вершин **L(x0, y0; …..xk, yk;… xf, yf)** , которая составляет кратчайший маршрут движения робота.

Пусть P – полигон размером $n \times n$, как определено в постановке задачи,

$W: P \to \mathbb{N}_0 \cup \{*\}$ – целочисленные метки, полученные в результате работы волнового алгоритма. Для маршрута $L = \langle (x_0,\ y_0), \dots, (x_k,\ y_k), \dots, (x_f, y_f) \rangle \subset P \times P \times \dots \times P$ рассмотрим набор условий:

1. $L_0(x_0,\ y_0) = P_{st}$ – маршрут начинается со стартовой точки, определённой входными данными;
2. $L_f(x_f,\ y_f) = P_{fin}$ – маршрут заканчивается финишной точкой, определённой входными данными;
3. $\forall k: 0 \le k < f\ \ \|L_k - L_{k+1}\|_1 = |x_k - x_{k+1}| + |y_k - y_{k+1}| = 1$ – соседние точки в упорядоченном списке должны являться соседними на плоскости. Иными словами, точка L_{k+1} должна лежать в выколотой окрестности точки L_k, и наоборот, как показано на рис. 1. Норму $\|\cdot\|_1$ принято именовать метрикой городских кварталов (city block);
4. $W\left(P_{fin} = (x_f, y_f)\right) = 0$ – финишная точка всегда помечается меткой 0;
5. $\forall k: 0 \le k < f\ \ W(L_{k+1}) = W(L_k) - 1$ – веса монотонно убывают с шагом 1.

Маршрут $L = \langle (x_0,\ y_0), \dots, (x_k,\ y_k), \dots, (x_f,\ y_f) \rangle$ является допустимым, если выполнены условия 1–3. При выполнении всех пяти условий можно гарантировать, что маршрут является оптимальным, и длина этого маршрута составляет $W(L_{st} = (x_{st},\ y_{st}))$. Стоит учесть, что допустимых маршрутов с минимальной длинной может быть несколько. Также, если начальная точка была помечена как $W(L_{st}) = *$, то оптимального маршрута (и вообще какого-либо маршрута) не существует.

Волновой алгоритм позволяет правильно сопоставить метки W для полигона P, а также способен отыскать оптимальный путь L из множества допустимых оптимальных путей L^*, если множество оптимальных путей не пусто: $L^* \neq \emptyset$.

Предположим, что робот находится в ячейке $P_{st} = R$ и ему необходимо добраться до точки $P_{fin} = D$, как показано на рисунке 3. Координатная сетка вводится следующим образом: левый верхний элемент имеет координаты (1, 1). Первая координата соответствует горизонтальному направлению, её ось направлена слева направо. Вторая координата соответствует вертикальному направлению, её ось направлена сверху вниз. Наряду с этим, жирной линией отмечено, что ячейки не доступны для перемещения в обоих направлениях.

Рассмотрим метод работы волнового алгоритма на приводимом ниже примере.

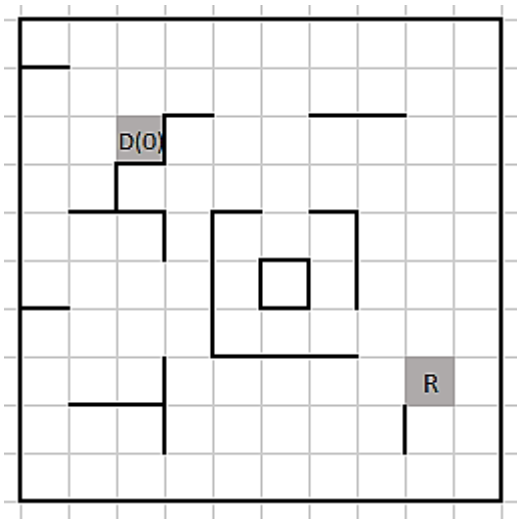


Рисунок 3 – Карта с начальными помеченными ячейками и с информацией о доступности

Распространение волны начинается с ячейки D с координатами (3, 3), в которую помещается значение 0. Это относится к этапу инициализации. Затем работает алгоритм распространения волны. После поиска третьего поколения волны, карта размечается значениями в соответствии с рисунком 4.

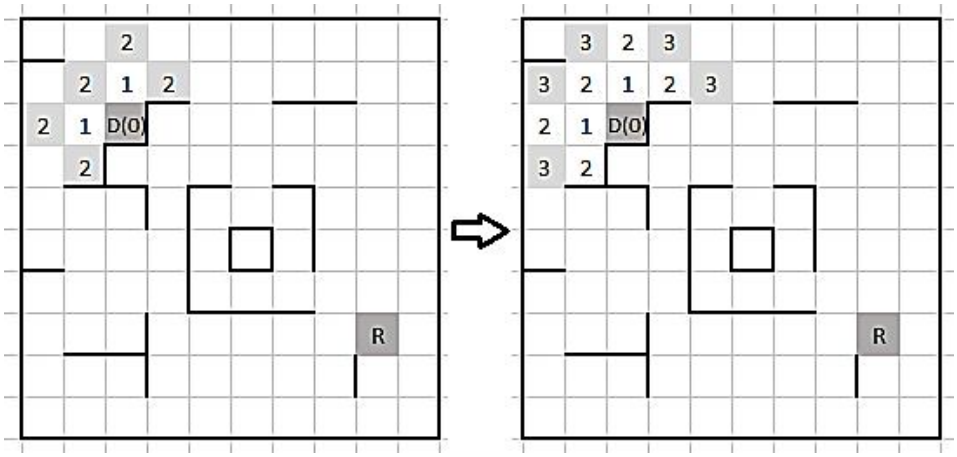


Рисунок 4 – Формирование третьего фронта волны

Легко понять, к какому поколению волны относится та или иная ячейка, помеченная как пройденная: в ней, по сути, содержится число с номером поколения волны. Как правило, когда непомеченная ячейка соседствует с двумя помеченными, то значение в этих соседних ячейках одинаково, так как алгоритм работает последовательно, создавая каждую волну целостно. Однако при распараллеливании вычислений возможен затруднённый обмен данными, и алгоритм работы придётся усложнить.

При полном распространении волны возникнет ситуация, показанная на рисунке 5 слева. Очевидно, что в однопоточной реализации работу алгоритма можно остановить на 13-той итерации распространения волны, так как фронт волны достиг точки положения робота, что позволяет нам начать этап восстановления пути. Тем не менее в параллельной реализации на CUDA рекомендуется осуществлять выполнение волнового алгоритма до полного окончания распространения всех волн – это самый надёжный критерий остановки массивно-параллельного волнового алгоритма. Выбор критерия моментальной остановки приведёт к приближённому, а не к точному решению. Также отметим, что в данной конфигурации карты присутствует недоступная клетка, помеченная символом *, до которой волна «не добралась».

На этапе восстановления волны (рисунок 5 справа) в список пути при инициализации помещается начальная точка движения робота: $R(8,2)$. Затем список последовательно заполняется координатами. На итерации 7 список приобретает следующий вид:

$$L_7 = \langle R = (9,8), (9,7), (8,7), (7,7), (7,6), (7,5), (6,5), (6,4) \rangle$$

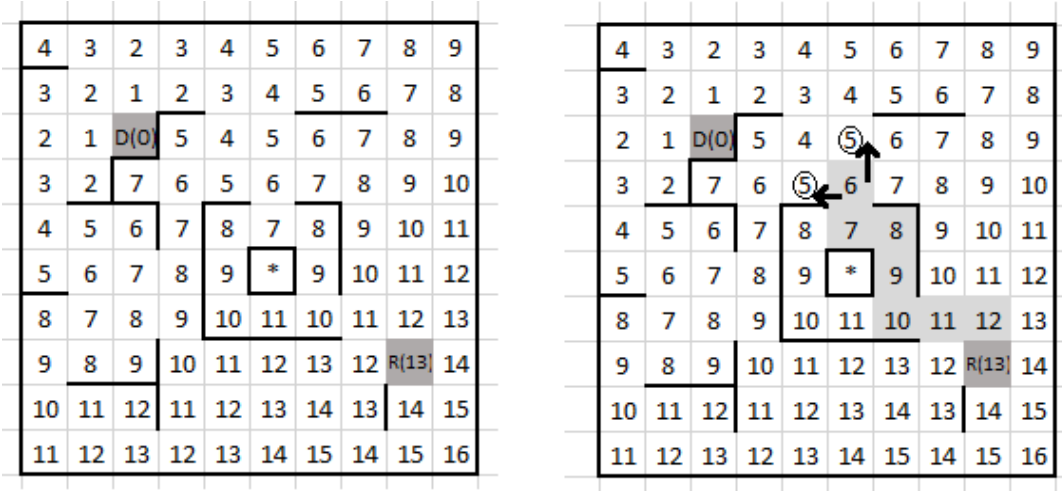


Рисунок 5 – Слева: последний возможный шаг распространения волны. Справа: этап восстановления пути

Так как на последней итерации не была достигнута точка $D(3,3)$, то берётся последняя точка из списка: $(6,4)$, берётся её метка: 6. Среди соседних ячеек мы ищем метку со значением $6-1=5$. Таких точек две – их координаты $(6,3)$ и $(5,4)$. Выбрать можно любую из них. Это обусловлено критерием оптимальности – в соответствии с ним длина пути должна быть минимальной. Оптимальное расстояние пути уже получено на этапе распространения волны, и равно метке в точке D. Соответственно, выбор любой из двух точек не повлияет на результирующую длину пути. Поэтому данные пути являются эквивалентно оптимальными по данному критерию. (см. п. 5 условий оптимальности пути).

Помещаем выбранную точку в список. Получаем массив следующей, 8-ой итерации:

$$L_8 = \langle R = (9,8), (9,7), (8,7), (7,7), (7,6), (7,5), (6,5), (6,4), (5,4) \rangle.$$

Процесс продолжается до тех пор, пока в путевой список не будет помещена точка D(3,3). Процесс всегда завершится, и вне зависимости от выбора точек длина пути составит $W\big(P_{st} = (9,8)\big) = 13$.

Выходные данные

```
path: array of array[1..2] of Integer
```

path представляет собой упорядоченный список ячеек, задаваемых двумя координатами, соединяя которые получаем маршрут, являющийся решением поставленной задачи. В случае отсутствия оптимального пути массив остаётся пустым.

Характеристика реализации алгоритма. Работа алгоритма включает в себя три этапа: инициализацию, распространение волны и восстановление пути.

Инициализация. На этапе инициализации создаётся дискретная квадратная сетка размером *n* на *n*. Каждой ячейке сетки, соответствующей полю (карте) приписываются атрибуты доступности.

Распространение волны. Распространение волны рассматривается пошагово для всей карты в целом. Для каждой ячейки на одном шаге распространения волны выполняем следующие действия:

1. п. 1: Выбрать текущую ячейку поля $C(i,j)$.
2. п. 2: Выбрать очередного соседа N из окрестности ячейки *C:* $N \in \{(i-1,j), (i,j+1), (i,j-1), (i+1,j)\}$. Если все соседи рассмотрены, то перейти к п. 6, иначе перейти к п. 3.
3. п. 3: Проверить по признакам доступности возможность перехода из соседней ячейки в текущую:$\|N - C\|_1 = 1$. В случае отсутствия доступности завершить работу с соседней ячейкой N и перейти к п.2, иначе перейти к п. 4.
4. п. 4: Взять метку соседней ячейки $W(N)$ и прибавить к ней единицу $W(N)+1$. Если текущая ячейка C не помечена $W(C) = *$, либо помечена превосходящим значением $W(C) > W(N)+1$, то отметить текущую ячейку значением $W(C) := W(N)+1$.
5. п. 5: перейти к п. 2.
6. п. 6: завершение алгоритма.

Блок-схема алгоритма для одной итерации распространения волны представлена на рисунке 6. Метки полигона хранятся в целочисленном массиве P размерности $n \times n$. В массиве A хранятся признаки доступности. Для каждой ячейки $C(i,j)$ массив из четырёх элементов $A[C] = A[i][j]$ хранит атрибуты доступности в следующем порядке: доступность слева, доступность сверху, доступность снизу, доступность справа.

Стоит отметить, что алгоритм реализован таким образом, что учитывает особенности массивно-параллельного программирования. Тело двойного цикла может быть распараллелено по данным на n^2 потоков, где *n* – количество ячеек полигона по каждой оси.

Алгоритм выполняется до тех пор, пока распространение волны не прекратится. Признаком остановки распространения волны является сохранение значений меток при переходе к следующей итерации.

Восстановление пути. Начинаем движение из ячейки с меткой Pst(x0,y0) , в которой находится робот. Заносим эту ячейку начального положения робота в список ячеек оптимального маршрута. Далее производим следующие действия: для последней ячейки с меткой Pfin(xf,yf) в упорядоченном списке находим соседа с меткой Pfin(xf,yf) −1 и заносим его в конец списка. Продолжать до тех пор, пока последняя ячейка в планируемом пути робота не совпадёт с точкой назначения с меткой «0».

Описание программного обеспечения. Исходный код был реализован на языке программирования Си с поддержкой технологии CUDA. Сборка программы осуществлялась с помощью инструментальных средств, доступных на СКЦ «Политехнический» [5]. В том числе использовался компилятор nvcc [12], поставляемый компанией “Nvidia” для компиляции кода под устройства с архитектурой, поддерживающей технологию массивно-параллельных вычислений CUDA. В реализация программы используется несколько файлов (модулей) исходного кода:

- ”**program.cu”** – основной код программы (осуществляет чтение и обработку входных параметров);
- **”waveKernelGlobal.cu”** – реализация параллельных вычислений волнового алгоритма Ли с использованием глобальной памяти;
- **”waveKernelShared.cu”** – реализация параллельных вычислений волнового алгоритма Ли с использованием разделяемой памяти. Блок CUDA размером *m*×*n* копирует в локальную память свою часть карты размером $(m+2) \cdot (n+2)$, захватывая соседнюю область в размере одной клетки. Работа с динамическими метками карты производится сначала с локальной копией. Затем происходит глобальная синхронизация блоков, после чего процесс распространения волны повторяется.
- **”waveKernelTexture.cu”** – реализация параллельных вычислений волнового алгоритма Ли с использованием текстурной памяти. Константные признаки доступности размещаются в глобальной памяти ускорителя особым образом, который позволяет сослаться на неё текстурной ссылке. Обращение к данным посредством текстурной ссылки организует кеширование данных, обеспечивая вероятностный прирост производительности.

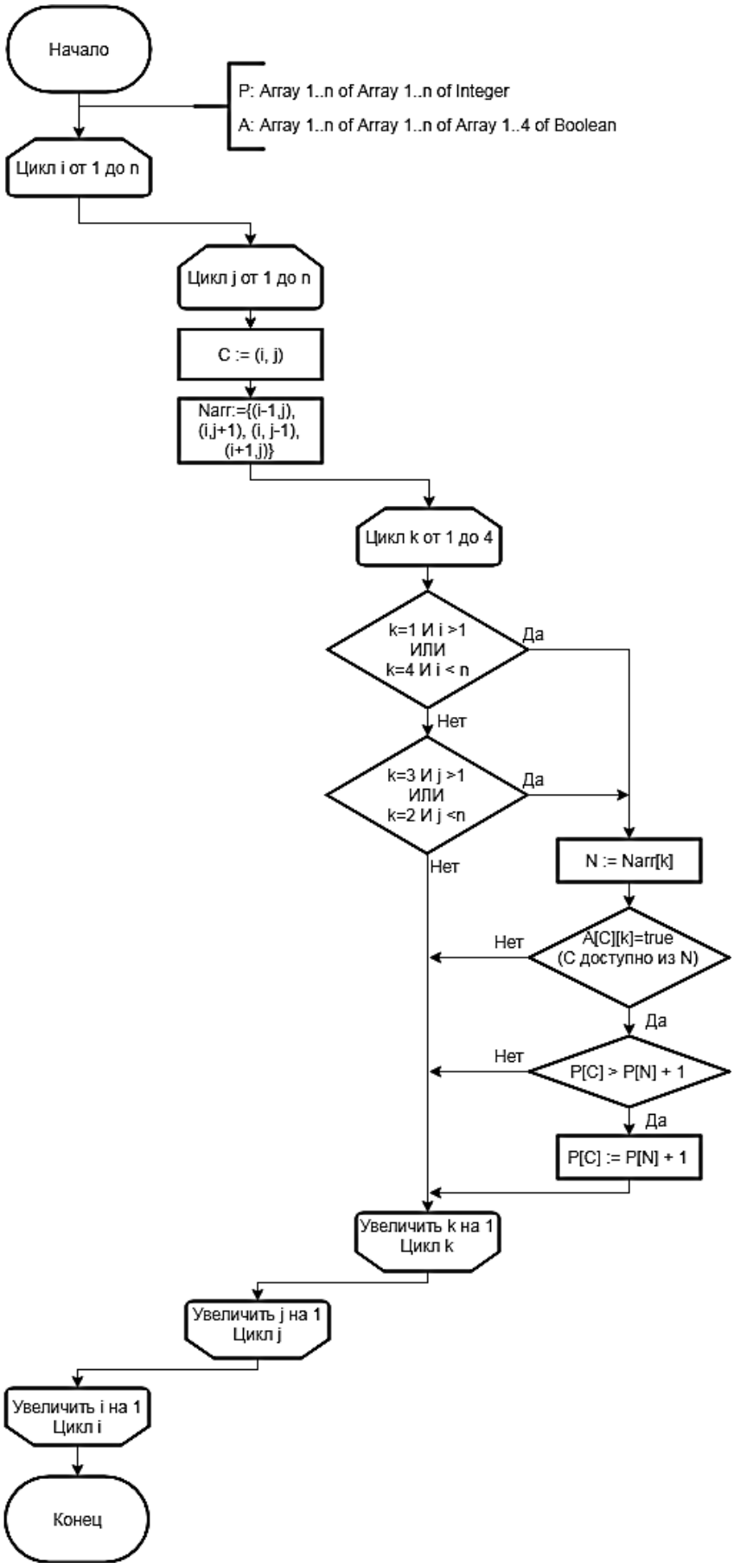


Рисунок 6 – Блок-схема алгоритма распространения волны (для следующего поколения)

По сути, в файлах ”waveKernelGlobal.cu”, ”waveKernelShared.cu” и ”waveKernelTexture.cu” представлен код, выполняющий один и тот же описанный алгоритм, но использующий разные парадигмы памяти CUDA.

Методика проведения вычислительных экспериментов. Для сравнения времени работы алгоритмов используем 11 классов недоступности ячеек полигона P(nxn). Разбиваем интервал $\mu \in [0\ \%, 100\ \%]$ на следующие интервалы: [0 %, 5 %), [5 %, 10 %), [10 %, 15 %), [15 %, 20 %), [20 %, 25 %), [25 %, 30 %), [30 %, 35 %), [35 %, 40 %), [40 %, 45 %), [45 %, 50 %), [50 %, 100 %].

В каждом интервале генерируем 30 вариантов полигона и вычисляем время решения задачи. По каждому интервалу проведём группировку данных и вычислим среднее значение времени работы алгоритма.

Выбирая некоторое n и манипулируя параметрами генерации карты для заполнения интервалов, получим столбчатую диаграмму, отображающую применимость алгоритма для карты с заданной степенью доступности. Размер полигона задаем значением $n = 10^4$. Эксперименты проводились на суперкомпьютере «Политехник – РСК Торнадо» [5]. Для работы на одном узле доступно два ускорителя вычислений CUDA с абсолютно одинаковыми характеристиками. Наиболее важные характеристики представлены ниже:

- Наименование ускорителя: «Tesla K40m»
- Объём памяти: около 12 Гб
- Compute Capability: 3.5

Графический ускоритель относится к архитектуре «Kepler».

- Максимальный размер разделяемой памяти на блок: 49152.
- Ограничение на максимальное количество нитей на блок осталось неизменным: $1024 = 32 \cdot 32$. Также размер dim3 (32, 32, 1).
- Число потоковых мультипроцессоров: 15, т.е. число независимо обрабатываемых блоков одновременно – 15 штук (один блок на мультипроцессор)
- Выравнивание текстурной памяти: 512.

Основной гипотезой является предположение о том, что наименьшее время решения задачи будет получено при работе с разделяемой памятью.

Результат обработки данных вычислительных экспериментов представлен на рисунке 6 в виде диаграммы зависимости времени выполнения программы для трёх методов работы с памятью в зависимости от степени недоступности карты.

Вывод 1: анализ полученных данных подтвердил выдвинутую гипотезу – целесообразно использование разделяемой памяти вместо глобальной и повышает скорость работы алгоритма в среднем в 10 раз.

Вывод 2: использование текстурной памяти даёт незначительное ускорение (в среднем в 1.2 раза) по сравнению с вариантом применения разделяемой памяти.

Вывод 3: при размере полигона $n = 10^4$ ни одна из реализаций не удовлетворяет требованиям реального времени, поставленным в условиях задачи. Максимальные значения времени решения задачи соответственно: $T_{global}^{\max} = 1925690, T_{shared}^{\max} = 161360,\ T_{texture}^{\max} = 144350$. Наихудший вариант решения задачи по скорости вычислений дает метод работы с глобальной памятью. Он отработал за 32 мин. на карте со степенью недоступности $\mu = 42{,}05\ \%$.

Приведенные результаты показывают, что с увеличением степени недоступности переходов от 5 до 40 % время решения задачи увеличивается. Это обусловлено просмотром бо́льшего числа возможных вариантов построения траектории. В то же время, при дальнейшем увеличении степени недоступности $\mu > 45\ \%$ время работы алгоритма резко падает – это объясняется невозможностью построения траектории перемещения в связи с полной изоляцией точек старта и финиша. В этих случаях (на интервале $\mu \in [50\ \%; 100\ \%]$) выдается диагностическое сообщение, «No path found» о невозможности построения пути и время работы алгоритма не превышает 400 мс при использовании локальной или текстурной памяти. При использовании глобальной памяти диагностика занимает от 1 до 3 с. Более точно:

$$\max_{\mu \in [50\ \%;\ 100\ \%]} T_{global} = 3340,\ \max_{\mu \in [50\ \%;\ 100\ \%]} T_{shared} = 400,\ \max_{\mu \in [50\ \%;\ 100\ \%]} T_{texture} = 360$$

$$\operatorname{avg}_{\mu \in [50\ \%;\ 100\ \%]} T_{global} = 1036,\ \operatorname{avg}_{\mu \in [50\ \%;\ 100\ \%]} T_{shared} = 261,\ \operatorname{avg}_{\mu \in [50\ \%;\ 100\ \%]} T_{texture} = 263$$

Попробуем найти значение n, для которых требования реального времени будут выполняться. Возьмём $n = 10^3$ и произведём те же самые измерения (вычисления).

В результате обработки полученных данных была получена зависимость, представленная на рисунке 7.

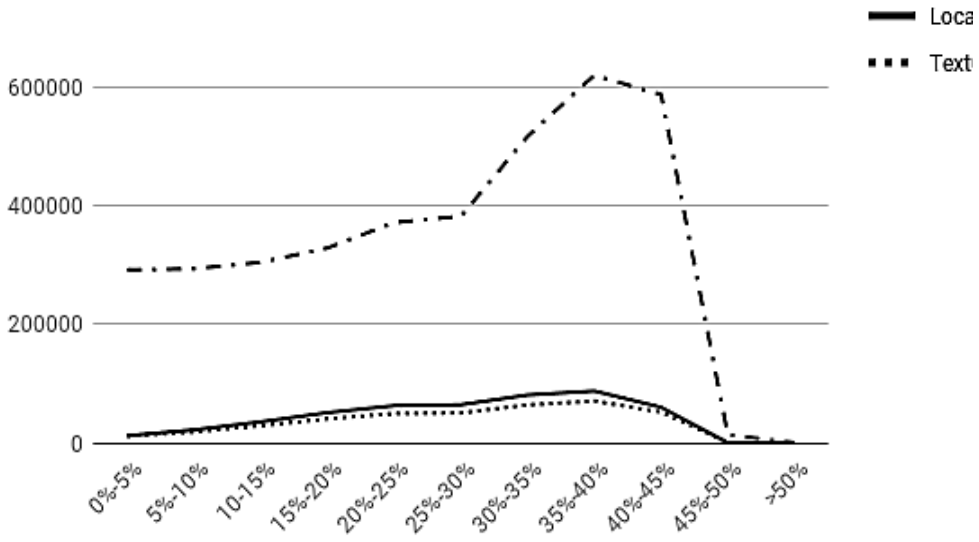


Рисунок 7 – Результаты измерения времени работы алгоритма Ли для размерности задачи n = 10^4. Усреднение по 30 измерениям на каждом интервале группировки. *Горизонтальная ось*: степень недоступности карты μ в процентах; *Вертикальная ось*: время в микросекундах; *Штрихпунктирная линия*: глобальная память, сплошная линия: локальная память, *пунктирная линия*: текстурная память

Данные экспериментов этой серии, представленные на рисунке 7, показывают следующее.

1. Наблюдается снижение эффективности применения текстурной памяти на картах с низкой степенью недоступности 0–30 %. Это обусловлено тем, что для небольшого объёма данных и меньшего количества обращений к текстуре эффективность кеширования снижается. Однако при более сложном распространении волны на картах с недоступностью 30% и более наблюдается прирост производительности за счёт кеширования более частого обращения к одинаковым данным, значения которых хранятся в кэше.

2. Все алгоритмы в среднем удовлетворяют требованиям реального времени, решения чи: $T^{\max}_{global} = 1390$, $T^{\max}_{shared} = 120$, $T^{\max}_{texture} = 190$. Как и в предыдущей серии, для размерности $n = 10^3$ использование разделяемой памятью является предпочтительным.

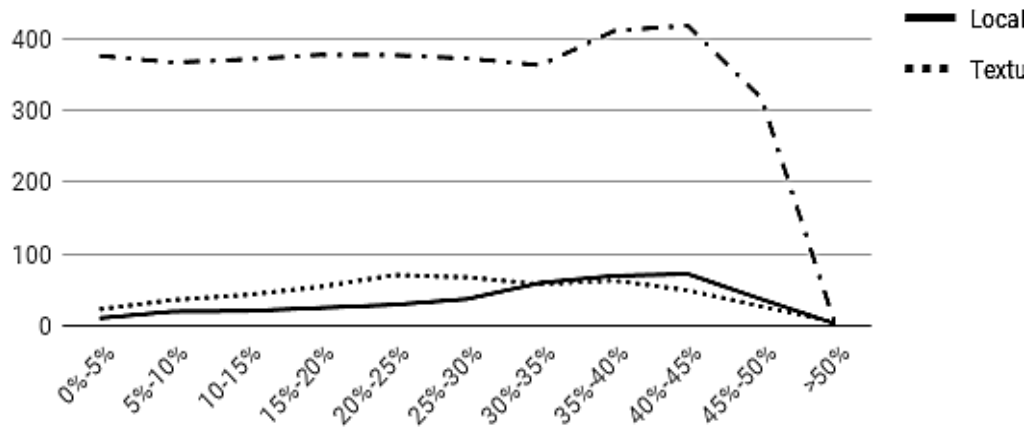


Рисунок 8 – Результаты измерения времени работы алгоритма Ли для размерности задачи n = 10^3. Усреднение по 30 измерениям на каждом интервале группировки. *Горизонтальная ось*: степень недоступности карты μ в процентах. Вертикальная ось: время в микросекундах. *Штрихпунктирная линия*: глобальная память, сплошная линия: локальная память, пунктирная линия: текстурная память

Выводы. Проведение вычислительных экспериментов позволило сравнить работу одного и того же алгоритма с разными парадигмами памяти, которые используются в технологии NVIDIA CUDA. Из трёх моделей памяти (глобальная, локальная и текстурная) наименее предпочтительно использование глобальной памяти в качестве рабочей области управления данными в процессе работы алгоритма. Значительно лучшие результаты дает использование локальной памяти: ядро программы копирует данные

из глобальной памяти для локального взаимодействия нескольких нитей в рамках одного блока, используя кэш уровня L1 потокового мультипроцессора. Это существенно повышает быстродействие.

Использование текстурной памяти должно быть обосновано спецификой задачи: если происходит частое чтение одинаковых данных набором соседних нитей, то текстурная память дает увеличение производительности. Это заключение обусловлено тем, что текстурная память всего лишь разметка глобальной памяти с возможностью кеширования данных в другие модели памяти потокового мультипроцессора. Прямые обращения к глобальной памяти по умолчанию не кэшируются.

Именно по этой причине при $n = 10^3$ использование текстурной памяти даёт проигрыш в производительности из-за дополнительных накладных расходов на управление текстурной памятью. Однако с ростом сложности карты ($\mu > 30$ %) наблюдается положительный эффект от использования текстурной памяти. Он обусловлен повышением числа обращений к данным из одних и тех же ячеек – вместо медленного обращения к глобальной памяти используется обращение к локальной памяти.

С повышением размерности задачи до $n = 10^4$ текстурная память даёт постоянный прирост в производительности вне зависимости от сложности карты местности. Тем не менее, программная реализация алгоритма Ли работает крайне медленно на такой размерности, в отличие от размерности $n = 10^3$, на которой работа программы удовлетворяет критериям реального времени. Таким образом, использование текстурной памяти можно рекомендовать для задач с очень большим числом чтения данных из глобальной памяти, которые кэшируются в память потокового мультипроцессора.

Библиографический список

1. Апанасик А. / Обход препятствий: волновой алгоритм (Алгоритм Ли) // Suvitruf's Blog URL: http://suvitruf.ru/2012/05/13/1176/volnovoj-algoritm-algoritm-li/ (дата обращения: 30 ноября 2018).

2. Климов М.А., Воротников С.А. Выборнов Н.А. / Способ калибровки систем локальной навигации мобильных роботов // Прикаспийский журнал: управление и высокие технологии. -2017 .- № 1 (37) .- С. 106 - 115

3. Козадаев А.С., Дубовицкий Е.В. / Реализация волнового алгоритма для определения кратчайшего маршрута на плоскости при моделировании трасс с препятствиями // Журнал «Вестник Томского государственного университета». - 2010. - т.15, вып.6.

4. Куратник Д. / Реализация волнового алгоритма нахождения кратчайшего пути к динамически движущимся объектам в unity3d на C# в 2d игре // Habr URL: https://habr.com/post/264189/ (дата обращения: 30 ноября 2018).

5. Лукашин А.А. / Руководства // Суперкомпьютерный центр «Политехнический» URL: http://scc.spbstu.ru/index.php/for-users/manuals-and-user-guides (дата обращения: 03.12.2018).

6. Моторин Д. Е., Попов С. Г. Алгоритм многокритериального поиска траекторий движения робота на многослойной карте// Информационно-управляющие системы. 2018. № 3. С. 45–53. doi:10.15217/issn1684-8853.2018.3.45

7. Dmitrii Motorin, Serge Popov, Vladimir Muliukha, “Collision-Free Path Planning Algorithm for Group of Robots in Spatio-Situational Uncertainty”, PROCEEDING OF THE 21ST CONFERENCE OF FRUCT ASSOCIATION, Helsinki, Finland, pp. 456-461, ISSN 2305-7254

8. Textures & Surfaces. CUDA Workshop for FSI. Gernot Ziegler, Developer Technology (Compute). // GPU Technology Conference URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/texture_webinar_aug_2011.pdf (дата обращения: 03.12.2018).

9. Jeff Larkin / GPU Fundamentals, November 14, 2016 // Innovative Computing Laboratory URL: http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf (дата обращения: 03.12.2018).

10. Travis Archer / Procedurally Generating Terrain // Morningside College. Sioux City y, Iowa 51106: 2011.

11. CUDA C Programming Guide // NVIDIA Developer URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability (дата обращения: 03.12.2018).

12. CUDA COMPILER DRIVER NVCC Reference Guide (TRM-06721-001_v10.0) // NVIDIA Developer URL: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf (дата обращения: 03.12.2018).

13. How and when should I use pitched pointer with the cuda API? // Stack Overflow URL: https://stackoverflow.com/questions/16119943/how-and-when-should-i-use-pitched-pointer-with-the-cuda-api (дата обращения: 03.12.2018).

14. 2D Texture from 2D array CUDA // Stack Overflow URL: https://stackoverflow.com/questions/11924537/2d-texture-from-2d-array-cuda (дата обращения: 03.12.2018).

15. Can one index CUDA texture with integers // Stack Overflow URL: https://stackoverflow.com/questions/7233579/can-one-index-cuda-texture-with-integers (дата обращения: 03.12.2018).

16. The different addressing modes of CUDA textures // Stack Overflow URL: https://stackoverflow.com/questions/19020963/the-different-addressing-modes-of-cuda-textures (дата обращения: 03.12.2018).

References

1. Apasnik A. / Obhod prepyatstviy: volnovoy algoritm (Algoritm Li) // Suvitruf's Blog URL: http://suvitruf.ru/2012/05/13/1176/volnovoj-algoritm-algoritm-li/ (data obrashcheniya: 30 noyabrya 2018). [Apanasik Andrey. Obstacle avoidance: Lee algorithm. Suvitruf's Blog. Web. 30 Nov. 2018. http://suvitruf.ru/2012/05/13/1176/volnovoj-algoritm-algoritm-li/]

2. Klimov Matvey.., Vorotnikov Sergey. Vybornov Nikolay. / Sposob kalibrovki sistem lokal'noy navigatsii mobilnykh robotov // Prikaspiyskiy zhurnal: upravleniye i vysokie tekhnologii. -2017 .- № 1 (37) .- С. 106 – 115 [Klimov M.A.., Vorotnikov S.A. Vybornov N.A. / Method of calibration of local navigation system of mobile robots. // Caspian journal: control and high technonogy. -2017 .- № 1 (37) .- С. 106 – 115]

3. Kozadayev A.S., Dubovitskiy Ye.V. / Realizatsiya volnovogo algoritma dlya opredeleniya kratchayshego marshruta na ploskosti pri modelirovanii trass s prepyatstviyami // Zhurnal «Vestnik Tomskovo gosudarstvennogo universiteta». - 2010. - t.15, vyp.6. [Kozadayev A.S., Dubovitskiy Ye.V. Realization of wave algorithm for determination of shortest route on platitude at modeling of tracks with obstacles]

4. Kuratnik D. / Realizatsiya volnovogo algoritma nahozhdeniya kratchayshego puti k dinamicheski dvizhushchimsya obyektam v unity3d na C# v 2d igre // Habr URL: https://habr.com/post/264189/ (data obrashcheniya: 30 noyabrya 2018). [Kuratnik Dmitriy. Lee algorithm C# implementation of finding shortest path for the dynamically moving objects for Unity3D two-dimensional game application. Habr. Web. 30 Nov. 2018. https://habr.com/post/264189/]

5. Lukashin A.A. / Rukovodstva // Supercompjuterniy Tsentr «Politechnicheskiy» URL: http://scc.spbstu.ru/index.php/for-users/manuals-and-user-guides (data obrashcheniya: 03.12.2018). [Lukashin A.A. Manuals. Supercomputer Center «Polytechnic» Web. 03 Dec. 2018. http://scc.spbstu.ru/index.php/for-users/manuals-and-user-guides (data obrashcheniya: 03.12.2018).]

6. Motorin D. E., Popov S. G. Algoritm mnogokriterial'nogo poiska traektorij dvizheniya robota na mnogoslojnoj karte// Informacionno-upravlyayushchie sistemy. 2018. № 3. S. 45–53. doi:10.15217/issn1684-8853.2018.3.45 [Motorin D. E., Popov S. G. Multi-Criteria Path Planning Algorithm for a Robot on a Multilayer Map. Informatsionnoupravliaiushchie sistemy [Information and Control Systems], 2018, no. 3, pp. 45–53 (In Russian). doi:10.15217/issn1684-8853.2018.3.45]

7. Dmitrii Motorin, Serge Popov, Vladimir Muliukha, "Collision-Free Path Planning Algorithm for Group of Robots in Spatio-Situational Uncertainty", PROCEEDING OF THE 21ST CONFERENCE OF FRUCT ASSOCIATION, Helsinki, Finland, pp. 456-461, ISSN 2305-7254

8. Textures & Surfaces. CUDA Workshop for FSI. Gernot Ziegler, Developer Technology (Compute). // GPU Technology Conference URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/texture_webinar_aug_2011.pdf (access date: 03.12.2018).

9. Jeff Larkin / GPU Fundamentals, November 14, 2016 // Innovative Computing Laboratory URL: http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf (access date: 03.12.2018).

10. Travis Archer / Procedurally Generating Terrain // Morningside College. Sioux City y, Iowa 51106: 2011.

11. CUDA C Programming Guide // NVIDIA Developer URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability (access date: 03.12.2018).

12. CUDA COMPILER DRIVER NVCC Reference Guide (TRM-06721-001_v10.0) // NVIDIA Developer URL: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf (access date: 03.12.2018).

13. How and when should I use pitched pointer with the cuda API? // Stack Overflow URL: https://stackoverflow.com/questions/16119943/how-and-when-should-i-use-pitched-pointer-with-the-cuda-api (access date: 03.12.2018).

14. 2D Texture from 2D array CUDA // Stack Overflow URL: https://stackoverflow.com/questions/11924537/2d-texture-from-2d-array-cuda (access date 03.12.2018).

15. Can one index CUDA texture with integers // Stack Overflow URL: https://stackoverflow.com/questions/7233579/can-one-index-cuda-texture-with-integers (дата обращения: 03.12.2018).

16. The different addressing modes of CUDA textures // Stack Overflow URL: https://stackoverflow.com/questions/19020963/the-different-addressing-modes-of-cuda-textures ((access date: 03.12.2018).