МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
ОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Ф.М. ДОСТОЕВСКОГО

В. В. Коробицын, Ю. В. Фролова

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА CUDA

Учебно-методическое пособие

Омск

ИЗДАТЕЛЬСТВО
ОМСКОГО ГОСУДАРСТВЕННОГО
УНИВЕРСИТЕТА

2016

УДК 004.4’2
ББК 32.973.202-018.2я73
К681

Рекомендовано к изданию
редакционно-издательским советом ОмГУ

Рецензенты:
канд. техн. наук *С. А. Хрущев*,
канд. техн. наук *И. Б. Ларионов*

Коробицын, В. В.

К681 Основы программирования на CUDA : учебно-методическое пособие / В. В. Коробицын, Ю. В. Фролова. – Омск : Изд-во Ом. гос. ун-та, 2016. – 68 с.

ISBN 978-5-7779-2015-7

Издание включает главы с практическими заданиями, предназначенные для освоения основ программирования на CUDA. Шесть глав дают представление о базовых понятиях разработки параллельных программ вычислений общего назначения на графическом процессоре.

Для студентов вузов и начинающих программистов.

УДК 004.4’2
ББК 32.973.202-018.2я73

ISBN 978-5-7779-2015-7

ОГЛАВЛЕНИЕ

Предисловие

Программно-аппаратная архитектура CUDA, разработанная корпорацией NVIDIA в 2007 году, позволяет производить параллельные вычисления на графических процессорах NVIDIA. Средство разработки NVIDIA SDK позволяет создавать программы, обеспечивающие выполнение вычислений общего назначения на графических процессорах (GPGPU). Такие программы позволяют повысить производительность без привлечения дорогостоящих суперкомпьютеров. CUDA – это эффективный подход к построению высокоскоростных вычислительных систем.

Изучение CUDA позволяет решить несколько задач:

- знакомство с параллельным программированием;
- освоение принципов вычислений общего назначения на GPU;
- формирование навыков реализации прикладных алгоритмов на CUDA.

В данном издании предложено 6 глав, рассчитанных на получение базовых навыков разработки программ на CUDA. Каждая из них решает свою практическую задачу.

Первая глава посвящена знакомству с базовыми понятиями параллельного программирования: параллельное исполнение операций, оценка ускорения и эффективности параллельных реализаций. Эта работа выполняется без привлечения CUDA.

Вторая глава посвящена подготовке рабочего места программиста на CUDA. Основная задача заключается в установке

и настройке CUDA на компьютере. Эту работу можно проводить на разных операционных системах: Windows, Linux, MacOS.

Третья глава посвящена знакомству с принципами разработки функции-ядра на CUDA, изучению архитектуры GPU, получению первого опыта разработки программы на CUDA.

Четвертая глава дает представление о принципах масштабирования программ на CUDA. Создание универсальных программ для широкого круга GPU (от мобильных до серверных).

Пятая глава знакомит со способами интеграции CUDA с графической библиотекой OpenGL. Дает представление о решении практической задачи ускорения вычислений на задаче построения растрового фрактала.

Шестая глава знакомит с принципами построения параллельных алгоритмов на CUDA на примере реализации клеточного автомата. Также показан пример интеграции CUDA с Qt.

Представленные темы не могут охватить всего разнообразия решаемых с помощью CUDA задач, но подготавливают программиста к самостоятельному углублению знаний в области GPGPU. Надеемся, что разработанный материал поможет в подготовке квалифицированных программистов.

Авторы выражают глубокую признательность рецензентам за полезные замечания, которые позволили сделать пособие лучше.

Глава 1. ЗНАКОМСТВО С ПАРАЛЛЕЛЬНЫМИ АЛГОРИТМАМИ

1.1. Понятие параллельного алгоритма

Понятие *параллельного алгоритма* всегда противопоставляется традиционному *последовательному алгоритму*. Параллельный алгоритм может быть реализован по частям на множестве вычислительных устройств с последующим корректным объединением результата вычислений.

Некоторые алгоритмы довольно легко поддаются распараллеливанию (разбиению на независимо выполняемые фрагменты). Например, поиск заданного элемента в неупорядоченном массиве. Эта задача разбивается на подзадачи разделением массива на части, в каждой из которых поиск производится независимо.

Однако большая часть известных алгоритмов являются последовательными и довольно трудно распараллеливаются. Причиной возникающих сложностей является зависимость результата вычислений от предыдущих этапов.

Тем не менее, параллельные алгоритмы весьма важны, поскольку повышение производительности современных вычислительных систем достигается посредством увеличением параллельности исполнения программ, в частности за счет увеличения количества ядер процессоров.

Изучение параллельных алгоритмов лучше начинать с простых примеров. Например, как распараллелить процесс суммирования элементов массива.

1.2. Параллельное суммирование

Рассмотрим задачу нахождения частных сумм последовательности числовых значений

$$S_k = \sum_{i=1}^{k} x_i, \quad k = 1, 2, \ldots, n,$$

где n – количество суммируемых значений (данная задача известна также под названием *prefix sum problem*).

Последовательный алгоритм суммирования. Традиционный алгоритм для решения задачи суммирования состоит в последовательном суммировании элементов числового набора с сохранением в переменную S

$$S = x_1, \; S = S + x_2, \ldots$$

Вычислительная схема данного алгоритма может быть представлена следующим образом (рис. 1):

$$G_1 = (V_1, R_1),$$

где $V_1 = (v_{01}, ..., v_{0n}, v_{11}, ..., v_{1n})$ – множество операций суммирования (вершины $v_{01}, ..., v_{0n}$ обозначают операции ввода, каждая вершина $v_{1i}, i = 1, 2, \ldots, n$ соответствует прибавлению значения x_i к накапливаемой сумме S), а $R_1 = \{(v_{0i}, v_{1i}), (v_{1i}, v_{1,i+1}), i = 1, 2, \ldots, n-1\}$ – множество дуг, определяющих информационные зависимости операций.

Как можно заметить, данный «стандартный» алгоритм суммирования допускает только строго последовательное исполнение и не может быть распараллелен.

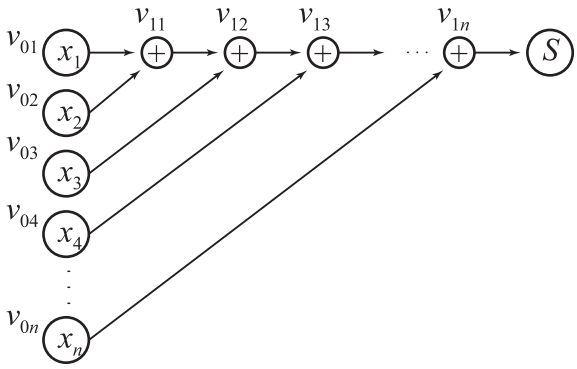

Рис. 1. *Последовательная вычислительная схема алгоритма суммирования*

Каскадная схема суммирования. Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Такой вариант суммирования (*каскадная схема*) состоит в следующем (рис. 2):

- на первой итерации все исходные данные разбиваются на пары, и для каждой пары вычисляется сумма двух значений;
- далее все полученные суммы пар разбиваются на пары и снова выполняется суммирование значений пар и т. д.

Будем считать, что $n = 2^k$, тогда вычислительная схема может быть определена как граф

$$G_2 = (V_2, R_2),$$

где $V_2 = \{(v_{i1}, \ldots, v_{il_i}), i = 0, 1, \ldots, k, l_i = 1, \ldots, n/2^i\}$ – вершины графа, $(v_{01}, \ldots, v_{0n})$ – операции ввода, $(v_{11}, \ldots, v_{1n/2})$ – операции первой итерации и т. д., а множество дуг графа

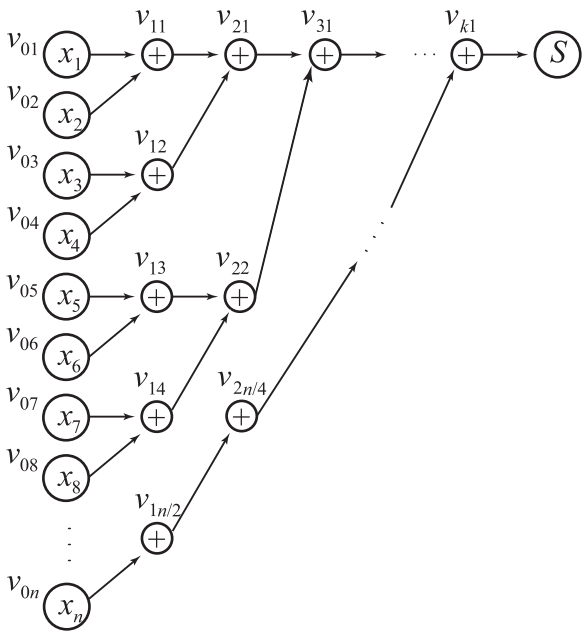

Рис. 2. *Каскадная схема алгоритма суммирования*

определяется соотношениями:

$$R_2 = \{(v_{i-1,2j-1}, v_{ij}), (v_{i-1,2j}, v_{ij}), i = 1, \ldots, k, j = 1, \ldots, n/2^i\}.$$

Количество итераций каскадной схемы оказывается равным величине $k = \log_2 n$, а общее количество операций суммирования

$$L_{\text{посл.}} = \frac{n}{2} + \frac{n}{4} + \cdots + 1 = n - 1$$

совпадает с количеством операций последовательного варианта алгоритма суммирования. При параллельном исполнении отдельных итераций каскадной схемы общее количество

параллельных операций суммирования

$$L_{\text{пар.}} = \log_2 n.$$

Предполагая, что скорость выполнения последовательных и параллельных операций одинаковы, получим показатели ускорения $\mathcal{S}_p$ и эффективности $\mathcal{E}_p$ каскадной схемы алгоритма суммирования

$$\mathcal{S}_p = \frac{T_1}{T_p} = \frac{L_{\text{посл.}}}{L_{\text{пар.}}} = \frac{n-1}{\log_2 n}, \quad \mathcal{E}_p = \frac{T_1}{pT_p} = \frac{n-1}{\frac{n}{2}\log_2 n},$$

где $p = \frac{n}{2}$ – необходимое для выполнения каскадной схемы количество параллельных потоков, T_1 – время выполнения вычислений в последовательном режиме, T_p – время выполнения вычислений в p потоках в параллельном режиме. Отметим, что эффективность использования параллельных потоков уменьшается при увеличении количества суммируемых значений $\lim \mathcal{E}_p \to 0$ при $n \to \infty$.

Модифицированная каскадная схема. Получение асимптотически ненулевой эффективности может быть обеспечено при использовании модифицированной каскадной схемы. В новом варианте каскадной схемы все проводимые вычисления подразделяются на два последовательно выполняемых этапа суммирования (рис. 3):

- на первом этапе вычислений все суммируемые значения подразделяются на $\frac{n}{\log_2 n}$ групп, в каждой из которых содержится $\log_2 n$ элементов. Далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования. Вычисления в каждой группе могут выполняться независимо друг от друга, т. е. параллельно. Для этого необходимо наличие не менее $\frac{n}{\log_2 n}$ процессоров;

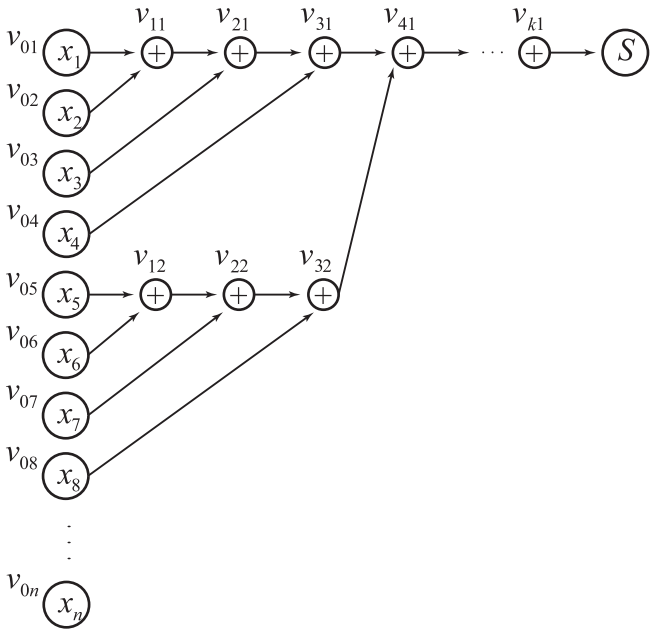

Рис. 3. *Модифицированная каскадная схема суммирования*

- на втором этапе для полученных $\frac{n}{\log_2 n}$ сумм отдельных групп применяется обычная каскадная схема.

Для выполнения первого этапа суммирования требуется выполнение $\log_2 n$ параллельных операций при использовании $p_1 = \frac{n}{\log_2 n}$ параллельных потоков.

Для выполнения второго этапа необходимо

$$\log_2 \left(\frac{n}{\log_2 n} \right) \leq \log_2 n$$

параллельных операций для $p_2 = \frac{n}{2 \log_2 n}$ потоков.

Как результат, данный способ суммирования характеризуется следующими показателями

$$T_p = 2\log_2 n, \quad p = \frac{n}{\log_2 n}.$$

С учетом полученных оценок показатели ускорения и эффективности модифицированной каскадной схемы определяются соотношениями

$$\mathcal{S}_p = \frac{n-1}{2\log_2 n}, \quad \mathcal{E}_p = \frac{n-1}{2(n/\log_2 n)\log_2 n} = \frac{n-1}{2n}.$$

Сравнивая данные оценки с показателями обычной каскадной схемы, можно отметить, что ускорение для предложенного параллельного алгоритма уменьшилось в 2 раза. Но для эффективности нового метода суммирования можно получить асимптотически ненулевую оценку снизу

$$\mathcal{E}_p = \frac{n-1}{2n} \geq \frac{1}{4}, \ \lim \mathcal{E}_p \to \frac{1}{2} \text{ при } n \to \infty.$$

Вычисление всех частных сумм. Вернемся к исходной задаче вычисления всех частных сумм последовательности значений и проведем анализ возможных способов последовательной и параллельной организации вычислений. Вычисление всех частных сумм в последовательном режиме получается в обычном последовательном алгоритме суммирования при том же количестве операций $T_1 = n - 1$. При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемому результату: достижение эффективного распараллеливания требует привлечения новых подходов (может даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения задач. Так, для рассматриваемой задачи нахождения всех

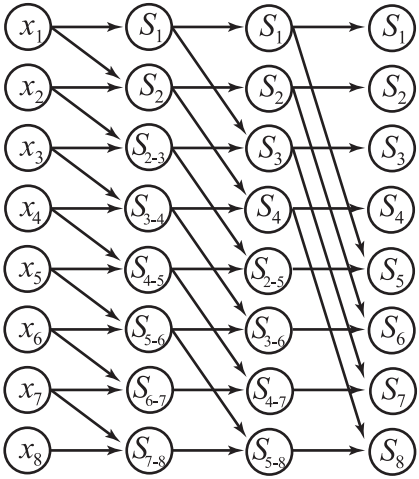

Рис. 4. *Схема параллельного алгоритма вычисления всех частных сумм (величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности)*

частных сумм алгоритм, обеспечивающий получение результатов за $\log_2 n$ параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (рис. 4):

- перед началом вычислений создается копия вектора суммируемых значений ($S = x$);
- на каждой итерации суммирования $i = 1, 2, \ldots, \log_2 n$, формируется вспомогательный вектор Q путем сдвига вправо вектора S на 2^{i-1} позиций (освобождающие при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов S и Q: $S \leftarrow S + Q$.

Параллельный алгоритм выполняется за $\log_2 n$ параллельных операций сложения. На каждой итерации алгоритма парал-

лельно выполняются n скалярных операций сложения, и таким образом общее количество выполняемых скалярных операций определяется величиной $L_{\text{пар.}} = n \log_2 n$ (параллельный алгоритм содержит большее количество операций по сравнению с последовательным способом суммирования). Необходимое количество параллельных потоков определяется количеством суммируемых значений ($p = n$). С учетом полученных соотношений, показатели ускорения и эффективности параллельного алгоритма вычисления всех частных сумм оцениваются следующим образом

$$\mathcal{S}_p = \frac{n}{\log_2 n}, \quad \mathcal{E}_p = \frac{n}{p \log_2 n} = \frac{1}{\log_2 n}.$$

Как следует из построенных оценок, эффективность алгоритма так же уменьшается при увеличении числа суммируемых значений. При необходимости повышения величины этого показателя может оказаться полезной модификация алгоритма, как в случае с обычной каскадной схемой.

1.3. Практические аспекты исследования алгоритмов

Для оценки эффективности реализаций алгоритмов часто требуется реализовать некоторые вспомогательные инструменты: генератор последовательности случайных чисел или оценка скорости вычислений. Фрагменты таких программ приведены ниже.

Генерация последовательности случайных чисел. Для генерации последовательности случайных чисел можно воспользоваться стандартными функциями `rand` и `srand` из библиотеки `stdlib`. Пример программы приведен ниже.

```
// ctr_rand.c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void SimpleRandDemo(int n)
{  for (int i = 0; i < n; i++)
      printf("  %6d\n", rand());
}

void RangedRandDemo(int range_min, int range_max,
     int n)
{  for (int i=0; i<n; i++)
   {  int u=(double)rand()/(RAND_MAX+1)*(range_max
           -range_min)+range_min;
      printf("  %6d\n", u);
   }
}

int main(void)
{  srand((unsigned)time(NULL));
   SimpleRandDemo(10);
   printf("\n");
   RangedRandDemo(-100, 100, 10);
}
```

Программа демонстрирует способ инициализации генератора с помощью таймера (функция `time` из библиотеки `time.h`). Это обеспечивает получение различных последовательностей случайных чисел при разных запусках программы. Функция `SimpleRandDemo` генерирует последовательность из `n` целых чисел из стандартного диапазона и выводит их на терминал. Функция `RangedRandDemo` генерирует последовательность случайных чисел в заданном диапазоне [`range_min, range_max`].

Вычисление времени работы программы. Для определения времени работы программы можно воспользоваться функцией `difftime` из библиотеки `time`. Программа определяет затраченное машинное время на вычисление 500 миллионов операций умножения чисел с плавающей запятой.

```
// crt_difftime.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{ time_t start, finish;
  long loop;
  double result, elapsed_time;
  printf("Умножаем два вещественных числа 500 ");
  printf("миллионов раз...\n" );
  time(&start);
  for(loop=0; loop<500000000; loop++)
     result=234.346*512.2707;
  time(&finish);
  elapsed_time=difftime(finish, start);
  printf("\nВычисления потребовали %6.0f секунд.\n",
         elapsed_time);
}
```

Отметим, что данный код необходимо компилировать с выключенной оптимизацией. Иначе компилятор на этапе оптимизации может заменить операцию умножения двух вещественных констант на результат умножения, что исказит результат замеров времени исполнения программы.

1.4. Практическое задание

Цель работы – имитация работы параллельной программы на скалярном процессоре на примере задачи вычисления последовательности бинарных операций.

Ход работы:

1. Изучить теоретический и практический материал главы.
2. Написать программу на языке Си для исследования алгоритмов, выполняющей следующие шаги:
   - сформировать массив размерности N, содержащий случайно сгенерированные вещественные числа в диапазоне $[a; b]$;
   - вычислить значение последовательности бинарных операций элементов этого массива с помощью последовательного алгоритма. Определить количество выполненных бинарных операций. Замерить время работы программы;
   - реализовать соответствующий вариант параллельного алгоритма. Определить количество бинарных операций. Замерить время работы программы.
3. Используя полученные замеры времени, оценить эффективность параллельных алгоритмов, выполняемых в последовательном режиме.

Таблица 1

Исходные данные для практического задания по вариантам

	N	a	b	*Алгоритм*	*Функция*
1	2^{20}	-5	5	casc	sum
2	2^{24}	-1	1	mcas	mul
3	2^{18}	-100	100	part	max
4	2^{22}	-1000	1000	casc	min
5	2^{19}	-50	50	mcas	mean
6	2^{21}	-32	32	part	min
7	2^{23}	-500	500	casc	max
8	2^{25}	-200	200	mcas	sum
9	2^{16}	-100	300	part	min
10	2^{17}	0	10	casc	max
11	2^{20}	-50	50	part	sum
12	2^{24}	-10	10	mcas	min
13	2^{18}	-1	1	part	mul
14	2^{22}	-100	100	casc	max
15	2^{19}	0	5	mcas	sum
16	2^{21}	-320	320	part	min
17	2^{23}	-50	50	casc	mean
18	2^{25}	-20	20	mcas	max
19	2^{16}	-1	1	part	mul
20	2^{17}	-100	100	casc	sum
21	2^{20}	0	1000	mcas	sum
22	2^{24}	2	10	mcas	min
23	2^{18}	-10	10	part	mean
24	2^{22}	-100	100	casc	sum
25	2^{19}	0	500	mcas	max

В таблице указаны следующие алгоритмы: casc – каскадная схема, mcas – модифицированная каскадная схема, part – вычисление частичных сумм; функции: sum – суммирование, mul – произведение, max – максимум, min – минимум, mean – среднее значение

Контрольные вопросы

1. Сформулируйте основные отличия параллельного алгоритма от последовательного.

2. В чем суть задачи prefix sum problem?

3. Что такое показатель ускорения параллельного алгоритма? Как он определяется?

4. Что такое коэффициент эффективности параллельного алгоритма? Как он вычисляется?

5. Что обозначают узлы и дуги на графе параллельного алгоритма?

6. Почему каскадная модель суммирования неэффективна при большом количестве суммируемых элементов? Какая существует альтернатива этому методу?

7. Какова эффективность модифицированного каскадного алгоритма суммирования при неограниченном росте количества суммируемых элементов?

8. Во сколько раз потребуется больше простейших операций суммирования при реализации параллельного алгоритма получения всех частных сумм по сравнению с последовательным алгоритмом суммирования?

9. Как измерить время работы алгоритма, если количество операций мало? Например, длительность работы алгоритма меньше 1 мс.

10. Используя оценки времени генерации и суммирования массива чисел, оцените количество арифметических операций, необходимых для генерации одного случайного числа.

Глава 2. ПОДГОТОВКА СРЕДЫ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

2.1. Описание инструментария

CUDA (Compute Unified Device Architecture) – программно-аппаратная архитектура параллельных вычислений, позволяющая увеличить скорость вычислений благодаря использованию графических процессоров компании Nvidia.

CUDA SDK (Software Development Kit) основан на расширении языка Си и дает возможность реализовывать алгоритмы, выполняемые на графических процессорах. Архитектура CUDA позволяет организовать доступ к набору инструкций графического процессора и управления его памятью при организации параллельных вычислений.

Набор инструментов CUDA:

- компилятор NVCC;
- библиотеки FFT и BLAS для вычислений на GPU;
- профилировщик;
- отладчик gdb для GPU;
- CUDA runtime драйвер;
- руководство по программированию;
- CUDA Developer SDK (исходный код, утилиты и документация).

CUDA является кроссплатформенной и поддерживает операционные системы Windows, Mac OSX, Linux. Аппаратное обеспечение, поддерживаемое CUDA, составляют графические карты Nvidia GeForce, начиная с 8 серии, специализированные

графические карты Nvidia Quadro, специализированные вычислители Nvidia Tesla.

Любой исходный код на CUDA должен быть скомпилирован с помощью NVCC. Компилятор NVCC вызывает все необходимые инструменты, такие как cudacc, g++, cl и т. д. Результаты работы NVCC:

- C код (host CPU код), который должен быть собран с остальной частью приложения, используя стандартный компилятор среды разработки;
- PTX объектный код (PTX исходный код), интерпретируемый при выполнении программы в код программы для GPU.

Любой исполняемый код программы на CUDA требует для запуска две динамические библиотеки: «The CUDA runtime library» (cudart) и «The CUDA core library» (cuda). Эти библиотеки устанавливаются в систему при установке CUDA.

До версии 2.3 включительно разработчики CUDA поддерживали режим эмуляции устройства CUDA. Это позволяло компилировать, запускать и отлаживать CUDA-программы без использования видеокарты Nvidia. Однако, начиная с версии 3.0, этот режим не поддерживается. Разработчики решили, что желающие программировать на CUDA вполне могут приобрести недорогую совместимую видеокарту вместо эмулятора. Хотя в режиме эмуляции были определенные плюсы. Запуская программу в режиме эмуляции, можно было использовать инструменты стандартного отладчика, иметь доступ к специфическим данным device из host кода и наоборот, вызывать любую host функцию из device кода (например, printf); обнаружить ситуации зависания, вызванные неправильным использованием синхронизации. Однако режим эмуляции имел некоторые неприятные особенности. Эмулируемые потоки запускались последова-

тельно, что могло привести к получению результатов работы, отличных от результатов, получаемых при запуске той же программы, но в параллельном режиме на GPU. А также разыменование device указателей на host или host указателей на device обеспечивало верные результаты в режиме эмуляции, но генерировало ошибку в режиме реального исполнения. И все же это в прошлом, а сейчас разработаны инструменты для тестирования и отладки программ на GPU.

2.2. Установка CUDA под Linux

Разработчики CUDA стараются поддерживать актуальные версии операционной системы Linux (Fedora, RHEL, CentOS, OpenSUSE, SLES, Ubuntu) и различные типы центральных процессоров (x86_64, ARMv7, POWER8). При установке CUDA необходимо наличие соответствующего ядра операционной системы, компилятора GCC и библиотеки GLIBC.

При написании данной инструкции ориентировались на дистрибутив Linux Ubuntu. При установке на другие версии потребуется выполнить подобные шаги с корректировкой на дистрибутив. Помимо этого, не стоит забывать о правах доступа к файлам. Для установки необходимо обладать правами администратора (пользователь root).

До установки CUDA. Перед установкой необходимо убедиться, что на вашем компьютере имеется подходящая графическая карта с помощью команды

```
$ lspci | grep -i nvidia
```

Если у вас установлена видеокарта от Nvidia и она содержится в списке совместимых с CUDA, то можно продолжать установку.

Для проверки версии операционной системы наберите

```
$ uname -m && cat /etc/*release
```

Выведенная информация позволит определить, какой именно дистрибутив CUDA вам необходим.

Версию компилятора GCC можно получить командой

```
$ gcc --version
```

Перед установкой новой версии CUDA может потребоваться удаление предыдущей версии

```
$ sudo /usr/local/cuda-X.Y/bin/uninstall_cuda_X.Y.pl
```

Драйвер можно удалить командой

```
$ sudo /usr/bin/nvidia-uninstall
```

Установка из репозитория. Для установки из репозитория необходимо настроить прокси-сервер. Далее в команде указать имя нужного дистрибутива, как в примере

```
$ sudo dpkg -i cuda-repo-ubuntu1410-7-0-local_7.0-28
       _amd64.deb
```

Далее обновить кэш репозитория

```
$ sudo apt-get update
```

и установить CUDA

```
$ sudo apt-get install cuda
```

Установка RUN-файла. Нужно скачать актуальную версию пакета CUDA. Ее можно получить на сайте компании по адресу http://developer.nvidia.com/cuda-downloads. Файл будет иметь название вида «cuda_7.0.28_linux.run», для удобства (чтобы не набирать изначально длинное имя файла) полученный файл можно переименовать, например, в «cuda.run».

Первым делом удаляется драйвер Nvidia

```
sudo apt-get --purge remove nvidia*
```

Затем нужно добавить в черный список драйвера Nouveau, которые могут конфликтовать с драйверами, прилагаемыми в пакете. Создать файл /etc/modprobe.d/blacklist-nouveau.conf со следующими строками

```
blacklist nouveau
options nouveau modeset=0
```

Обновить ядро initramfs

```
$ sudo update-initramfs -u
```

Затем нужно установить ряд пакетов, необходимых для работы некоторых образцов из SDK

```
sudo apt-get update
sudo apt-get install build-essential freeglut3-dev
     libxi-dev libxmu-dev mpi-default-dev libx11-dev
     libgl1-mesa-glx libglu1-mesa libglu1-mesa-dev
```

В файл /.bash_profile нужно добавить следующую строку

```
export PATH=/usr/local/cuda-5.0/bin:$PATH
```

А также эту (для 32-разрядной версии)

```
export LD_LIBRARY_PATH=/usr/local/cuda-5.0/lib
       :$LD_LIBRARY_PATH
```

И для 64-разрядной версии

```
export LD_LIBRARY_PATH=/usr/local/cuda-5.0/lib
       :/usr/local/cuda-5.0/lib64:$LD_LIBRARY_PATH
```

Теперь можно приступить к установке CUDA. Перейдите в консольный режим (например, Ctrl-Alt-F1) и отключите менеджер экрана

```
sudo service lightdm stop
```

Перейдите в папку, где расположен пакет CUDA и выполните следующую команду (где «cuda.run» – имя пакета)

```
sudo sh cuda.run
```

После завершения установки необходимо удостовериться, что установка всех трех частей пакета (драйвера, инструментария SDK и примеров) прошла успешно. Для этого нужно просмотреть сообщения, оставленные после завершения данного процесса. Так, в случае неуспешной установки какого-то из компонентов напротив него будет выведено сообщение «Installation Failed».

Для проверки работоспособности пакета соберите и запустите `deviceQuery` (находится в SDK, каталог `samples/1_Utilities/deviceQuery`). В случае корректной установки будет выдана информация об устройствах, поддерживающих CUDA, в случае ошибки будет написано «no CUDA-capable device is detected».

Запустите менеджер экрана

```
sudo service lightdm --full-restart
```

После этого перезагрузите систему.

2.3. Практическое задание

Цель работы – подготовить компьютер для работы в среде разработки параллельных программ.

Ход работы:

1. Изучить теоретический и практический материал главы.
2. Установить среду разработки.
3. Установить драйвер CUDA, пакет CUDA Toolkit, средства разработки CUDA SDK.
4. Настроить среду разработки для работы с CUDA.
5. Скомпилировать и запустить не менее трех проектов, входящих в состав CUDA SDK (в том числе, `device_Query`). Полученную информацию об устройствах CUDA сохранить в отчет.

Контрольные вопросы

1. Расшифруйте аббревиатуру CUDA.
2. Кто является разработчиком CUDA?
3. Что входит в состав инструментария CUDA?
4. Какие операционные системы поддерживаются разработчиками CUDA?
5. Что означают понятия host-код и device-код?
6. Что должно быть установлено на компьютере, чтобы можно было запустить откомпилированную CUDA-программу?
7. Какие среды разработки поддерживают CUDA?
8. Какие аппаратные средства поддерживает CUDA?
9. Какими правами должен обладать пользователь, чтобы установить CUDA на компьютер?
10. Какую информацию позволяет получить программа `deviceQuery`?

Глава 3. РЕАЛИЗАЦИЯ ПРОСТЕЙШЕЙ ПРОГРАММЫ НА CUDA

Знакомство с принципами программирования на CUDA начнем с реализации алгоритма умножения двух квадратных матриц.

3.1. Реализация умножения двух квадратных матриц

С математической точки зрения умножение двух квадратных матриц выглядит как операция попарного скалярного умножения строк и столбцов матриц

$$P = M \cdot N,$$

где P, M, N – квадратные матрицы чисел p_{ij}, m_{ij}, n_{ij}; $i, j = 1, 2, ..., W$. В координатной записи результат умножения будет

$$p_{ij} = m_{i1}n_{1j} + m_{i2}n_{2j} + ... + m_{iW}n_{Wj}, \text{ для всех } i, j = 1, 2, ..., W.$$

Реализация алгоритма умножения двух квадратных матриц на языке C++ для CPU

```
void MatrixMulOnHost(float** M, float** N, float** P,
                     int Width)
{
  for (int i = 0; i < Width; i++)
    for (int j = 0; j < Width; j++)
    {
      double sum = 0.0;
```

```
      for (int k = 0; k < Width; k++)
        sum += M[i][k] * N[k][j];
      P[i][j] = sum;
    }
}
```

В этой функции используются три вложенных цикла. Два используются для перебора всех элементов матрицы P (индексы i, j), и один – для вычисления суммы в скалярном произведении (индекс k). Необходимо отметить, что вычисление элементов матрицы P могут производиться независимо, то есть значение элемента p_{ij} не зависит от вычисленных ранее значений матрицы P. Это позволяет надеяться на успешное распараллеливание алгоритма по индексам i и j. Вычисление элементов матрицы P можно производить одновременно в разных вычислительных потоках. Отметим, что распараллеливание цикла по индексу k также возможно, используя параллельное суммирование, принципы которого были рассмотрены в главе 1. Однако мы рассмотрим параллельные реализации без разворачивания цикла по k.

Схема программы умножения матриц на CUDA для GPU содержит 3 этапа, которые можно представить в виде следующего фрагмента кода

```
void MatrixMulOnDevice(float* M, float* N, float* P,
                       int Width)
{
   int size = Width * Width * sizeof(float);
   float *Md, *Nd, *Pd;

// 1. Выделяем память для матриц M, N в памяти GPU
// и копируем туда данные
   cudaMalloc(&Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc(&Nd, size);
```

```
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

// Выделяем место под матрицу P на GPU
    cudaMalloc(&Pd, size);

// 2. Запускаем ядро, будет показано ниже
    ...

// 3. Копируем результат вычислений в P
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

// Освобождаем память
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Приведенная функция демонстрирует общий подход работы с GPU как с вычислительным сопроцессором. Взаимодействие осуществляется в три этапа: подготовка и пересылка данных в память на GPU, запуск вычислительного ядра, копирование результатов вычислений в оперативную память компьютера. Отметим, что это общий принцип, и иногда происходят отклонения от него. Например, запускается не одно ядро, а последовательно несколько разных ядер. Причем данные для следующего запускаемого ядра хранятся на GPU, а не копируются вновь. Также результаты вычислений могут не возвращаться на CPU, если они используются для вывода на экран. Тогда данные, хранимые на GPU, могут быть обработаны и выведены без копирования на CPU средствами самого GPU, используя для этого, например, библиотеку OpenGL.

Функция вычислительного ядра должна отражать суть выполняемых параллельных вычислений. Для чего используются средства CUDA, обеспечивающие идентификацию вычислительного потока. Приведем пример ядра, выполняющего скалярное умножение строки матрицы M на столбец матрицы N.

Результат сохраняется как элемент матрицы P.

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
           float* Pd, int Width)
{
    int i = threadIdx.y, j = threadIdx.x;
    float mik, nkj, pij = 0;
    for (int k = 0; k < Width; k++)
    {
       mik = Md[i * Width + k];
       nkj = Nd[k * Width + j];
       pij += mik * nkj;
    }
    Pd[i * Width + j] = pij;
}
```

Ключевое слово `__global__` используется для обозначения функции ядра. Такая функция может быть запущена только с CPU и выполняется только на GPU. Идентификация потоков осуществляется с помощью встроенной в CUDA переменной `threadIdx`, имеющей два поля `x` и `y`. Эти поля содержат значения номеров потока в блоке параллельно исполняемых потоков. Двумерный индекс служит для обеспечения удобства обращения к элементам матрицы. При запуске ядра будет сформировано столько параллельных потоков, сколько элементов в матрице P. Соответствующая конфигурация сетки потоков и запуск ядра представлены ниже.

```
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
  Width);
```

Данная конфигурация показывает, что в сетке (Grid) будет один блок (Block) размера `Width` $\times$ `Width` потоков. Отметим, что используемый тип `dim3` содержит три поля `x, y, z`. Однако для конфигурации сетки могут использоваться только два

из них, а для блока – все три. Причем, если требуется только одномерный индекс, то значения полей `y` и `z` должны быть равны 1, но не нулю.

Оператор запуска ядра состоит из трех полей: имя ядра, конфигурация сетки, передаваемые параметры. Имя ядра соответствует имени описанной ранее функции ядра. Конфигурация сетки задается в специальных тройных угловых скобках. Передаваемые параметры перечисляются в круглых скобках через запятую. Важно отметить, что параметрами ядра могут быть указатели на ячейки памяти GPU, но они не должны быть указателями на ячейки оперативной памяти. Такая ошибка не может быть выявлена компилятором, но обязательно приведет к ошибке исполнения программы.

К примеру, НЕЛЬЗЯ обычным способом, принятым в C++, определить и заполнить двухмерный массив, а затем его скопировать в память GPU. Поскольку при копировании будут скопированы значения указателей, а не значения элементов двухмерного массива. По этой причине используемые в примере массивы одномерные. В этом случае элемент матрицы с индексами i, j будет представлен в одномерном массиве индексом $k = i \cdot W + j$.

3.2. Практическое задание

Цель работы – реализовать на CUDA и запустить программу, вычисляющую матричную функцию.

Ход работы:

1. Изучить теоретический материал главы.
2. Реализовать программу вычисления матрицы D, заданной матричной функцией, согласно варианту из таблицы 2. Размер матриц $W \times W$. Программа реализуется на CUDA в параллельном исполнении одного потока на

один элемент результирующей матрицы. Матрицы A и B должны заполняться случайными числами в интервале $[a; b]$.

3. Реализовать функцию вычисления матрицы D на CPU. Сравнить результат вычисления на CUDA с результатом работы программы на CPU. Оценить отклонение результатов.
4. Определить максимальный размер W матриц, которые можно перемножить с помощью разработанной программы на вашем устройстве. Обосновать полученный результат.

Контрольные вопросы

1. Поясните назначение функций `cudaMalloc`, `cudaMemcpy`, `cudaFree`.

2. Что означает директива `__global__`?

3. Что такое функция-ядро в CUDA?

4. Как обозначаются функции, запускаемые и исполняемые на устройстве?

5. Как задается конфигурация запуска функции-ядра?

6. Какие типы параметров можно передавать функции-ядра?

7. Как передать массив в ядро?

8. Как вернуть результат вычислений из ядра?

9. Что такое область видимости переменной? Как разграничивается область видимости в CUDA?

10. Какие виды памяти есть в CUDA? Перечислите и дайте краткую характеристику каждому виду.

11. К каким видам памяти имеется доступ из функции-ядра?

12. В какие виды памяти можно записывать данные с хоста?

Таблица 2

Варианты заданий

№	*Матричная функция*	$[a; b]$
1	$D = 2(A^2 + B)(2B - A)$	$[-1; 1]$
2	$D = 3A - (A + 2B)B^2$	$[-10; 10]$
3	$D = 3A^2 - (A + 2B)B$	$[0; 1]$
4	$D = (A - 2B^2)(2A + B^3)$	$[-10; 0]$
5	$D = 2(A - B)(A^2 + B)$	$[-100; 100]$
6	$D = (A - B)^2 A + 2B$	$[-1; 1]$
7	$D = (A^2 - B^2)(A + B^2)$	$[-10; 5]$
8	$D = 2(A - B)(A^2 + B)$	$[-3; 3]$
9	$D = 2A - (A^2 + B)B$	$[-2; 2]$
10	$D = 2(A - 2B) + A^3 B$	$[-1000; 1000]$
11	$D = (A - B)A^2 + 3B$	$[-30; 10]$
12	$D = 3(A^2 + B^2) - 2AB$	$[-10; 30]$
13	$D = 2A^3 + 3B(AB - 2A)$	$[-10; 50]$
14	$D = A(A^2 - B) - 2(B + A)$	$[0; 50]$
15	$D = (2A - B)(3A + B) - 2A^2 B$	$[0; 10]$
16	$D = A(3A - B^2) + 2B$	$[-10; 10]$
17	$D = (2A - B^3) - 4AB$	$[-0,1; 0,1]$
18	$D = (3A + AB) - 2A^2$	$[-0,01; 0,01]$
19	$D = 3A^2 B - 4A$	$[-10; 40]$
20	$D = 2(A - B)A^2 - B$	$[-40; 40]$
21	$D = (3A - B^3) - A^2 B$	$[-7; 7]$
22	$D = (2A - B)(3A + B^2)$	$[-8; 8]$
23	$D = (A - 3B^2) - AB$	$[-5; 11]$
24	$D = (AB - BA)^2$	$[-1,8; 3,5]$
25	$D = (A - 3B)^2 - AB^2$	$[-1,4; 1,8]$

Глава 4. БЛОЧНАЯ ИЕРАРХИЯ ПРОГРАММ НА CUDA

Выполняя задание из предыдущей главы, вы столкнулись с ограничениями количества исполняемых потоков в одном блоке. Это ограничение существенно сокращает круг реализуемых задач и снижает эффективность реализации параллельных вычислений. В этой главе мы познакомимся с принципами организации потоков в CUDA, снимающими ограничение на количество параллельных потоков.

4.1. Разбиение потоков на блоки

Поскольку количество потоков в одном блоке на CUDA ограничено определенным числом, в частности в версии 2.3 это число равно 512, то максимальный размер матрицы, вычисляемой с помощью программы из главы 3 также ограничен числом 22 ($22 \times 22 = 484 < 512, 23 \times 23 = 529 > 512$). Одним из возможных вариантов решения этой проблемы является применение иерархии потоков CUDA. Суть сводится к тому, что потоки группируются в блоки (не более чем по 512 потоков в одном блоке), а блоки объединяются в сетку блоков. Количество блоков в одной сетке ограничено весьма большим числом ($65536 \times 65536 = 4,29$ млрд блоков).

Различие между потоками в одном блоке и в разных существенно. Потоки в одном блоке могут обмениваться информацией через разделяемую память, доступную только для блоков

одного потока. Кроме того, возможно осуществление барьерной синхронизации для выполнения потоков одного блока. Все это не применимо для потоков в разных блоках. Однако разбиение на блоки потоков позволяет реализовать практически неограниченную масштабируемость программ. Это выражается в оптимизации выполнения программы в пределах одного блока, где количество потоков известно заранее, а также в независимости реализации от количества блоков в сетке. Для задачи умножения матриц можно оптимизировать код программы для умножения фрагментов матриц размера 16×16 в одном блоке и решать задачу для произвольной размерности матрицы.

Возьмем за основу код функции `MatrixMulKernel` из главы 3 и модифицируем его для реализации выполнения на множестве блоков потоков. Для этого потребуется реализовать более сложную систему адресации к элементам массивов `Md` и `Nd`. Теперь индекс элемента в массиве будет зависеть не только от номера потока `threadIdx.x` и `threadIdx.y`, но и от номера блока `blockIdx.x` и `blockIdx.y`. Индекс столбца элемента будет `blockDim.x*blockIdx.x+threadIdx.x`, а индекс строки – `(blockDim.y*blockIdx.y+threadIdx.y)*width`. Занесем эти значения в переменные `bx` и `by` соответственно. Добавим цикл по фрагментам матрицы (переменная `i`) и введем дополнительные переменные `ix=i*blockDim.x` и `iy=i*blockDim.y*width`, показывающие смещение по столбцам и строкам согласно выбранному фрагменту `i`. Тогда окончательный индекс элемента в матрице `Md` будет равен `by+ix+k`, в матрице `Nd` – `bx+iy+k*width`. Текст функции приведен ниже.

```
__global__ void MatrixBlockMulKernel(float* Md,
           float* Nd, float* Pd, int width)
{
  int bx=blockDim.x*blockIdx.x+threadIdx.x;
```

```
  int by=(blockDim.y*blockIdx.y+threadIdx.y)*width;
  float pij=0;
  for (int i=0; i<gridDim.x; i++)
  {
    int ix=i*blockDim.x;
    int iy=i*blockDim.y*width;
    for (int k=0; k<blockDim.x; k++)
    {
      float mik = Md[by+ix+k];
      float nkj = Nd[bx+iy+k*width];
      pij += mik * nkj;
    }
   }
   Pd[bx+by] = pij;
}
```

4.2. Использование разделяемой памяти

Для того чтобы сократить количество обращений к глобальной памяти, где хранятся матрицы `Md` и `Nd`, необходимо воспользоваться разделяемой памятью (shared memory). Для этого каждый фрагмент матриц `Md` и `Nd` сначала будем копировать в разделяемую память, а затем производить вычисления части скалярного произведения. Размер фрагмента определим заранее константой `tile_width`. Определим массивы `shared_M` и `shared_N` в разделяемой памяти, при этом можно использовать двумерные массивы. Копирование элементов из глобальной памяти в эти массивы производится параллельно, и после копирования необходимо произвести синхронизацию потоков `__syncthreads()`. Далее при вычислении скалярного произведения требуется обращение только к разделяемой памяти. Модифицированный код приведен ниже.

```
__global__ void MatrixBlockMulKernel(float* Md,
```

```
            float* Nd, float* Pd, int width)
{
  int tx=threadIdx.x;
  int ty=threadIdx.y;
  int bx=blockDim.x*blockIdx.x+tx;
  int by=(blockDim.y*blockIdx.y+threadIdx.y)*width;
  float pij = 0;
  __shared__ float shared_M[tile_width][tile_width];
  __shared__ float shared_N[tile_width][tile_width];
  for (int i=0; i<gridDim.x; i++)
  {
    int ix=i*blockDim.x;
    int iy=i*blockDim.y*width;
    shared_M[ty][tx]=Md[by+ix+tx];
    shared_N[ty][tx]=Nd[bx+iy+ty*width];
    __syncthreads();
    for (int k=0; k<blockDim.x; k++)
    {
      float mik = shared_M[ty][k];
      float nkj = shared_N[k][tx];
      pij += mik * nkj;
    }
  }
  Pd[bx+by] = pij;
}
```

4.3. Проверка выхода за границу массива

Для обеспечения вычислений произведения матриц, размер которых не кратен размеру фрагмента, необходимо осуществить проверку пределов изменения индексов массивов. Они не должны выходить за пределы размера матриц `width`. Окончательный текст функции приведен ниже.

```
__global__ void MatrixBlockMulKernel(float* Md,
            float* Nd, float* Pd, int width)
```

```
{
  int tx=threadIdx.x;
  int ty=threadIdx.y;
  int bx=blockDim.x*blockIdx.x+tx;
  int by=(blockDim.y*blockIdx.y+threadIdx.y)*width;
  float pij=0;
  __shared__ float shared_M[tile_width][tile_width];
  __shared__ float shared_N[tile_width][tile_width];
  for (int i=0; i<gridDim.x; i++)
  {
    int ix=i*blockDim.x;
    int iy=i*blockDim.y*width;
    if (ix+tx<width && by<width*width)
      shared_M[ty][tx]=Md[by+ix+tx];
    if (iy+ty*width<width*width && bx<width)
      shared_N[ty][tx]=Nd[bx+iy+ty*width];
    __syncthreads();
    if (bx<width && by<width*width)
      for (int k=0; k<blockDim.x; k++)
      {
        float mik = shared_M[ty][k];
        float nkj = shared_N[k][tx];
        pij += mik * nkj;
      }
  }
  if (bx<width && by<width*width)
    Pd[bx+by] = pij;
}
```

4.4. Практическое задание

Цель работы – реализовать программу на CUDA с использованием блочной иерархии.

Ход работы:

1. Изучите теоретический материал.
2. Модернизируйте функцию вычисления матричной функции на CUDA, выполненную в лабораторной работе 3, с целью разбиения матрицы на фрагменты, вычисление которых осуществляется в одном блоке потоков с использованием разделяемой памяти для временного хранения строк и столбцов фрагментов матриц.
3. Определите максимальный размер фрагмента, который можно разместить на вашем GPU.
4. Оцените изменение производительности программы по сравнению с программой, выполненной в лабораторной работе 3.
5. Определите максимальный размер матриц, которые можно перемножить полученной программой. Установите основные ограничения, связанные с вычислениями на конкретном GPU. Оцените максимальный размер матриц для вашего GPU.
6. Определите максимальный размер матрицы, вычисление которой производится на вашем компьютере не более, чем за 5 минут.

Контрольные вопросы

1. Каково максимальное количество потоков в одном блоке допустимо на вашей конфигурации системы?

2. Для чего разработчики CUDA решили разбивать все параллельные потоки на блоки?

3. Каким образом программа узнает, в каком вычислительном потоке она выполняется?

4. Как можно осуществить реализацию программы, в которой разные потоки выполняют разные вычисления?

5. Как осуществляется выбор входных данных для разных потоков?

6. Объясните назначение переменных `blockIdx` и `threadIdx`?

7. Какова размерность индексов `blockIdx` и `threadIdx`?

8. Что такое разделяемая память? Для чего она используется?

9. В чем состоит различие между потоками в одном блоке и в разных?

10. Каким объемом ограничена разделяемая память на вашем устройстве?

Глава 5. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ В ЗАДАЧАХ КОМПЬЮТЕРНОЙ ГРАФИКИ

Задачи компьютерной графики как никакие другие подходят к расчетам на GPU в параллельном режиме. Задачи расчета цвета пиксела при визуализации различных моделей, имеют параллельный характер. Как правило, каждый пиксел рассчитывается независимо от других. Это дает возможность параллельной реализации методов визуализации. Расчет каждого пиксела в этом случае производится в отдельном потоке. Проиллюстрируем эффективность параллельной реализации на примере построения фрактального множества.

5.1. Построение фрактального множества

Фрактал можно определить как объект довольно сложной формы, получающийся в результате выполнения простого итерационного цикла. Итерационность (рекурсивность) обуславливает такие свойства фракталов, как «самоподобие». Отдельные части похожи по форме на весь фрактал в целом. В 1975 году французский математик Бенуа Мандельброт издал книгу «The Fractal Geometry of Nature», чем ознаменовал появление понятия фрактала. Книга неоднократно переиздавалась [15].

Будем рассматривать фрактальные множества на комплексной плоскости. Такие множества образуются при помощи применения итерационной процедуры для вычисления последовательности комплексных чисел для каждой точки изображения.

Количество итераций до выполнения заданного условия завершения вычислений покажет цвет окраски точки.

Фрактал Мандельброта можно построить с помощью следующей итерационной процедуры. Для каждой точки изображения необходимо выполнить цикл итераций

$$Z_{k+1} = Z_k^2 + Z_0, \text{ где } k = 0, 1, 2, ...$$

Используемые здесь комплексные числа $Z_k = X_k + iY_k$ представляются на экране компьютера цветными точками с координатами (X_0, Y_0). Для каждой точки выполняется ограниченное число итераций: до тех пор, пока $|Z_k| \leq A$. Число итераций задает определенный цвет точки. Квадрат комплексного числа вычисляется по формуле

$$Z_k^2 = (X_k^2 - Y_k^2) + i(2X_kY_k).$$

Для *фрактала Жюлиа* итерационная процедура имеет вид

$$Z_{k+1} = Z_k^2 + C,$$

где C – комплексная константа. Условие завершения $|Z_k| \leq A$.

Для *фрактала Ньютона* итерационная процедура задается формулой

$$Z_{k+1} = \frac{3Z_k^4 + 1}{4Z_k^3}.$$

Условие завершения $|Z_k| < \varepsilon$.

Ниже приведен текст ядра, обеспечивающего вычисление множества Мандельброта.

```
#define MAX_V 4.0
#define WIDTH 512
#define HEIGHT 512
__global__ void mandelbrotmenge(pcolor *colors,
```

```
            iter_type max_iter)
{
  const uint kx = gridDim.x * blockDim.x;
  const uint ky = gridDim.y * blockDim.y;
  uint iter = 1;
  double x = 0, y = 0;
  uint cx = blockDim.x * blockIdx.x + threadIdx.x;
  double cxx = ((double)cx / kx - (double)(WIDTH / 2)
    / kx) * 4.0;
  uint cy = blockDim.y * blockIdx.y + threadIdx.y;
  double cyy = ((double)cy / ky - (double)(HEIGHT / 2)
    / ky) * 4.0;
  double x2, y2;
  do {
    x2 = x * x;
    y2 = y * y;
    if (x2 + y2 > MAX_V)
    {
      colors[cy*WIDTH+cx] = iter;
      return;
    }
    y = 2 * x * y + cyy;
    x = x2 - y2 + cxx;
    iter++;
  } while (iter <= max_iter);
  colors[cy * WIDTH + cx] = 0;
}
```

5.2. Вспомогательные функции

Далее представлена функция, которая будет выделять место под массив точек, запускать ядро, копировать результат в оперативную память и возвращать указатель на этот массив.

```
uint* createPixelColors()
{
    uint *dcolors;
```

```
    cudaMalloc(&dcolors, size);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(WIDTH/dimBlock.x, HEIGHT/dimBlock.y);

    mandelbrotmenge <<<dimGrid, dimBlock>>>(dcolors,
      32);

    uint *hcolors = new uint[size];
    cudaMemcpy(hcolors, dcolors, size,
      cudaMemcpyDeviceToHost);

    cudaFree(dcolors);
    return hcolors;
}
```

Важно не забыть освободить память от созданного массива после того, как фрактал нарисован.

```
void destroyPixelColors(uint *colors)
{
    delete []colors;
}
```

Фрагмент программы для отображения фрактала с помощью графических функций Windows GDI.

```
    uint* colors = createPixelColors();
    hdc = GetDC(hWnd);
    for (uint i=0; i<WIDTH; i++) {
        for (uint j=0; j<HEIGHT; j++) {
            SetPixel(hdc, i, j, RGB(colors[i+j*WIDTH],
                     0, 0));
        }
    }
    ReleaseDC(hWnd,hdc);
    destroyPixelColors(colors);
```

5.3. Интеграция CUDA с OpenGL

Графический вывод результатов расчета можно реализовать различными способами. Один из самых простых заключается в интеграции CUDA с OpenGL. Приведем пример программы, реализующий такую интеграцию.

```
// файл "main.cpp"
#include <cuda_runtime_api.h>
#include <stdio.h>
#include <memory.h>
#include <glut.h>
#pragma comment(lib,"cudart.lib")
#pragma comment(lib,"glut32.lib")

// массив, содержащий изображение
int *hcolors;
// размер картинки
const int width =  512,
          lenght = 512;
          size = width*lenght*sizeof(int);

// прототип функции из файла "main.cu"
void run_cuda(int *colors);

void Idle();
void display();

int count;

int main(int argc, char* argv[])
{
  // выделение памяти
  hcolors = new int[size];
  memset(hcolors,0,size);

  // инициализация устройства CUDA
  int deviceCount = 0;
```

```
cudaGetDeviceCount(&deviceCount);
if (deviceCount == 0)
  printf("Не найдено совместимое устройство\n");
else
{
  printf("Обнаружено %d CUDA устройств\n",
    deviceCount);
  cudaSetDevice(0);
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, 0);

  printf("\nУстройство %d: \"%s\"\n", 0,
    deviceProp.name);

  int driverVersion = 0, runtimeVersion = 0;
  cudaDriverGetVersion(&driverVersion);
  cudaRuntimeGetVersion(&runtimeVersion);
  printf("Версия драйвера и библиотеки CUDA %d.%d
    / %d.%d\n",
    driverVersion/1000, (driverVersion%100)/10,
    runtimeVersion/1000, (runtimeVersion%100)/10);
  printf("Пределы совместимости CUDA: %d.%d\n\n",
    deviceProp.major, deviceProp.minor);

  // запуск вычислительной функции
  run_cuda(hcolors);

  // вывод результата в OpenGL
  glutInit(&argc, argv);
  glutInitWindowSize(512, 512);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
    GLUT_DEPTH);
  glutCreateWindow("Графическое CUDA приложение");
  glOrtho(0.0, 512.0, 0.0, 512.0, -1.0, 1.0);
  glutDisplayFunc(display);
  glutIdleFunc(Idle);
  glutMainLoop();
}
```

```
  delete [] hcolors;
  return 0;
}

void Idle()
{
    count++;
    glutPostRedisplay();
}

void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POINTS);
  for (int i=0; i<512; ++i)
    for (int j=0; j<512; ++j)
    {
      glColor3ub(0, hcolors[i+j*512]+count, 0);
      glVertex2f(i, j);
    }
  glEnd();
  glutSwapBuffers();
}


// файл "main.cu"
const
__global__ void run_device(int *colors)
{
  int cx=blockDim.x*blockIdx.x+threadIdx.x;
  int cy=blockDim.y*blockIdx.y+threadIdx.y;
  colors[cy*512+cx]=cx*cy;
}

void run_cuda(int *colors)
{
  int *dcolors;
  cudaMalloc((void**) &dcolors, size);
  dim3 dimBlock(16, 16);
```

```
  dim3 dimGrid(width/dimBlock.x, lenght/dimBlock.y);
  run_device<<<dimGrid, dimBlock>>>(dcolors);
  cudaMemcpy(colors, dcolors, size,
    cudaMemcpyDeviceToHost);
  cudaFree(dcolors);
}
```

Использование библиотеки GLUT преследует две цели:

1) создание кроссплатформенного кода;

2) GLUT позволяет облегчить изучение OpenGL.

Чтобы начать программировать под OpenGL, используя GLUT, требуется всего страница кода. Все функции GLUT начинаются с префикса glut.

5.4. Практическое задание

Цель работы – создать программу на CUDA, обеспечивающую параллельную реализацию алгоритма компьютерной графики.

Ход работы:

1. Изучите теоретический материал.
2. Используя приведенные примеры программ, создайте свою программу, которая будет вычислять фрактал (табл. 3) с помощью CUDA и отображать его с помощью OpenGL.

Контрольные вопросы

1. Почему задачи компьютерной графики так хорошо поддаются распараллеливанию?

2. Какими средствами можно выводить графические результаты вычислений на CUDA?

Таблица 3

Варианты заданий

№	*Фрактал*		№	*Фрактал*
1	$Z_{k+1} = Z_k^2 + Z_0$		14	$Z_{k+1} = Z_k^6 + C$
2	$Z_{k+1} = Z_k^2 + C$		15	$Z_{k+1} = Z_k - \frac{Z_k^6 - 1}{6Z_k^5}$
3	$Z_{k+1} = \frac{3Z_k^4 + 1}{4Z_k^3}$		16	$Z_{k+1} = Z_k^7 + Z_0$
4	$Z_{k+1} = Z_k^3 + Z_0$		17	$Z_{k+1} = Z_k^7 + C$
5	$Z_{k+1} = Z_k^3 + C$		18	$Z_{k+1} = Z_k - \frac{Z_k^7 - 1}{7Z_k^6}$
6	$Z_{k+1} = Z_k - \frac{Z_k^3 - 1}{3Z_k^2}$		19	$Z_{k+1} = Z_k^8 + Z_0$
7	$Z_{k+1} = Z_k^4 + Z_0$		20	$Z_{k+1} = Z_k^8 + C$
8	$Z_{k+1} = Z_k^4 + C$		21	$Z_{k+1} = Z_k - \frac{Z_k^8 - 1}{8Z_k^7}$
9	$Z_{k+1} = Z_k - \frac{Z_k^4 - 1}{4Z_k^3}$		22	$Z_{k+1} = Z_k^9 + Z_0$
10	$Z_{k+1} = Z_k^5 + Z_0$		23	$Z_{k+1} = Z_k^9 + C$
11	$Z_{k+1} = Z_k^5 + C$		24	$Z_{k+1} = Z_k - \frac{Z_k^9 - 1}{9Z_k^8}$
12	$Z_{k+1} = Z_k - \frac{Z_k^5 - 1}{5Z_k^4}$		25	$Z_{k+1} = Z_k^{10} + Z_0$
13	$Z_{k+1} = Z_k^6 + Z_0$			

3. Что позволяет обеспечить библиотека OpenGL при работе совместно с CUDA?

4. Какие функции необходимо вызвать для инициализации программы, использующей GLUT?

5. Является ли программа, написанная на CUDA совместно с GLUT кроссплатформенной?

6. Что такое фрактал? Каков вид фрактала Мандельброта?

7. Какие средства для работы с графикой имеются в CUDA?

Глава 6. РЕАЛИЗАЦИЯ АЛГОРИТМА ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ

Имитационное моделирование является важным и востребованным инструментом исследования различных процессов природы и техники. Успех использования средств имитационного моделирования во многом зависит от эффективности программной реализации. Поэтому современные среды моделирования обязаны использовать технологии параллельного программирования, чтобы обеспечить высокую производительность.

В данной главе приведем пример реализации алгоритма моделирования поведения клеточного автомата с применением технологии CUDA.

6.1. Модель клеточного автомата

Теория автоматов – раздел дискретной математики, изучающий абстрактные автоматы, то есть математические модели вычислительных машин.

Детерминированным клеточным автоматом называется кортеж из пяти элементов $(Q, \Sigma, \delta, S_0, F)$, где:

- Q – множество состояний автомата;
- Σ – алфавит языка автомата;
- δ – функция перехода $\delta : Q \times \Sigma \to Q$;
- $S_0 \in Q$ – начальное состояние;
- $F \subseteq Q$ – множество конечных состояний.

Автомат читает конечную строку символов $a_1, a_2, \ldots, a_n$, где $a_i \in \Sigma$, которая называется *входным словом*. Набор всех слов записывается как Σ^*. Слово $w \in \Sigma^*$ принимается автоматом, если $q_n \in F$. Говорят, что язык L читается автоматом M, если он состоит из слов w на базе алфавита Σ таких, что если эти слова вводятся в M, по окончанию обработки он приходит в одно из принимающих состояний F

$$L = \{w \in \Sigma^* | \delta(S_0, w) \in F\}.$$

Автомат переходит из состояния в состояние с помощью функции перехода δ, читая при этом один символ из ввода.

Клеточный автомат можно определить как множество конечных автоматов, каждый из которых может находиться в одном из состояний $q \in \Sigma \equiv \{0, 1, 2, \ldots, k-1, k\}$. Изменение состояний автомата происходит согласно правилу перехода

$$\sigma_{i,j}(t+1) = \phi(\sigma_{k,l}(t) | \sigma_{k,l}(t) \in \mathcal{N}),$$

где $\mathcal{N}$ – множество автоматов, составляющих окрестность. К примеру, окрестность фон Неймана определяется как

$$\mathcal{N}_N^1(i,j) = \{\sigma_{k,l} |\ |i-k| + |j-l| \leq 1\},$$

а окрестность Мура

$$\mathcal{N}_M^1(i,j) = \{\sigma_{k,l} |\ |i-k| \leq 1, |j-l| \leq 1\}.$$

Одним из интересных клеточных автоматов является так называемая игра «Жизнь».

6.2. Описание игры «Жизнь»

Игра «Жизнь» – клеточный автомат, придуманный математиком Джоном Конвеем в 1970 году [11].

Правила. Место действия – размеченная на клетки плоскость (в пределе неограниченная). Каждая клетка на плоскости может находиться в двух состояниях: быть «живой» или быть «мертвой» (пустой). Каждая клетка имеет 8 соседей. Распределение клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по следующим правилам:

- в пустой клетке, рядом с которой ровно три «живые» клетки, рождается жизнь;
- если у «живой» клетки есть две или три «живые» соседки, то эта клетка продолжает жить; в противном случае (если соседей меньше двух или больше трех) клетка умирает («от одиночества» или «от перенаселенности»).

Игра прекращается, если на поле не остается ни одной «живой» клетки, если при очередном ходе не меняет своего состояния (стабильная конфигурация) или если конфигурация на очередном шаге в точности (без сдвигов и поворотов) повторит себя же на одном из более ранних шагов (периодическая конфигурация).

Игрок не принимает прямого участия в игре, а лишь расставляет начальную конфигурацию. Затем наблюдает за изменением поколений.

Существуют различные модификации игры по размерности, цветности, направлению алгоритма, константам эволюции, размерам игрового поля, активности поля, количеству игроков, активности игры, геометрии поля.

6.3. Интеграция с Qt

Реализацию игры «Жизнь» совместим с интеграцией CUDA со средой разработки Qt. Покажем, что использование CUDA может совмещаться с различными средами разработки. Отметим наиболее важные моменты интеграции.

Проект Qt-приложения помимо стандартно генерируемых файлов будет содержать два заголовочных файла (`.h`), два файла исходного кода (`.cpp`) и один файл CUDA (`.cu`). Назначение этих файлов:

- `main.cpp` – главный файл проекта. Содержит главную функцию `main`, которая создает окно Qt-приложения и запускает обработчик событий;
- `life.h` – файл описания функций игры;
- `life.cu` – реализация функций игры на CUDA. Содержит три функции-ядра: инициализация поля (`init`), копирование поля (`dup`), шаг игры (`life`); и две функции доступные извне: инициализация игры (`game_init`), шаг игры (`game_step`);
- `mainwidow.h` – файл описания основного окна приложения;
- `mainwindow.cpp` – файл реализации обработчика событий приложения.

Приведем текст файлов проекта с комментариями.

```
// файл "main.cpp"
#include <QtGui/QApplication>
#include "life.h"
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
```

```
    MainWindow w;
    w.show();
    return a.exec();
}
```

Функция `main` в этом проекте стандартная для Qt-приложения.

```
// ƒайл "life.h"
#ifndef LIFE_H
#define LIFE_H
typedef unsigned int uint;
uint* game_init(uint width, uint height, uint *P);
void game_step(uint width, uint height, uint *M, uint *P);
#endif // LIFE_H
```

В заголовочном файле `life.h` описаны только C-функции, которые могут быть вызваны из других частей программы. А CUDA-функции остаются внутри файла `life.cu`.

```
// ƒайл "life.cu"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <stdlib.h>
#include <stdio.h>
#include "life.h"

__global__ void init(uint *M, uint width, uint height,
    uint *obj, uint w, uint h, uint x, uint y)
{
  uint mx = blockIdx.x*16+threadIdx.x;
  uint my = blockIdx.y*16+threadIdx.y;
  if (mx>=x && mx<x+w && my>=y && my<y+h)
    M[my*width+mx] = obj[(my-y)*w+mx-x];
  else M[my*width+mx] = 0;
}

__global__ void dup(uint *M, uint *T, uint width,
    uint height)
```

```
{
  uint mx = blockIdx.x*16+threadIdx.x;
  uint my = blockIdx.y*16+threadIdx.y;
  T[my*width+mx] = M[my*width+mx];
}

__global__ void life(uint *M, uint *T, uint width,
    uint height)
{
  uint mx = blockIdx.x*16+threadIdx.x;
  uint my = blockIdx.y*16+threadIdx.y;
  if (mx<width && my<height)
  {
    int s=0, x, y;
    for(int i=-1;i<2;i++)
    {
      x=mx+i;
      if (x>=width) x=0;
      if (x<0) x=width-1;
      for (int j=-1; j<2; j++)
      {
        y=my+j;
        if (y>=height) y=0;
        if (y<0) y=height-1;
        s+=T[y*width+x];
      }
    }
    if (s==3 || (s==4 && T[my*width+mx]))
      M[my*width+mx]=1;
    else M[my*width+mx]=0;
  }
}

uint* game_init(uint width, uint height, uint *P)
{
  uint size = width*height*sizeof(uint);
  uint *M, *cuobj, sobj, w, h;
  #ifdef static_init
```

```
    //static
    uint obj[] = {
        1, 1, 0, 0, 0, 0, 0, 0, 1, 1,
        1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
        1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 1, 1, 0, 0, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 1, 1, 1};
    w = 10;
    h = 10;
    //end of static
  #else
    //dynamic
    double k=0.3;
    uint obj[width*height];
    for (uint i=0; i<width*height; i++)
      if (random()-RAND_MAX*k>0) obj[i]=1;
        else obj[i]=0;
    w = width;
    h = height;
    //end of dynamic
  #endif
    sobj = w*h*sizeof(uint);
    cudaMalloc(&M, size);
    cudaMalloc(&cuobj, sobj);
    cudaMemcpy(cuobj, obj, sobj,
      cudaMemcpyHostToDevice);
    dim3 grid(width/16, height/16);
    dim3 block(16, 16);
    init<<<grid, block>>>(M, width, height, cuobj, w,
      h, 0, 0);
    cudaFree(obj);
    return M;
}
```

```
void game_step(uint width, uint height, uint *M, uint *P)
{
  uint size = width*height*sizeof(uint);
  uint *T;
  cudaMalloc(&T, size);
  dim3 grid(width/16, height/16);
  dim3 block(16, 16);
  dup<<<grid, block>>>(M, T, width, height);
  life<<<grid, block>>>(M, T, width, height);
  cudaMemcpy(P, M, size, cudaMemcpyDeviceToHost);
}
```

Вычислительное ядро построено по следующему принципу. Для вычисления состояния одной клетки используется один вычислительный поток. Все потоки группируются в блоки размером 16×16 потоков (256 потоков в одном блоке). Предполагается, что размер поля кратен 16 по горизонтали и вертикали.

Приведенная реализация не дает возможности игроку задавать начальную конфигурацию. Она либо задается статически внутри кода, либо генерируется случайным образом. Для этой цели используется массив `obj`. Для статической инициализации в начале программы необходимо определить константу `#define static_init`, иначе будет производиться динамическая инициализация.

Ядро `life` реализует конечный автомат одной клетки. Подсчитывается количество соседних клеток, в которых есть жизнь. Если их количество равно 3 (включая саму рассчитываемую клетку или без нее) или равно 4 (включая саму клетку), тогда клетка становится или остается «живой». В противном случае становится пустой. Состояние всех клеток берется из массива `T` и сохраняется в массив `M`. Перед каждой итерацией массив `M` копируется в `T` (ядро dup). Используемое в модели поле завернуто в тор. Левая граница склеена с правой, а нижняя –

с верхней. Это реализовано соответствующими проверками выхода за границы массивов.

```
// файл "mainwindow.h"
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QTimer>

namespace Ui{ class MainWindow; }

class MainWindow : public QMainWindow
{
  Q_OBJECT
public:
  explicit MainWindow(QWidget *parent = 0);
  ~MainWindow();
private:
  Ui::MainWindow *ui;
  uint w, h, *P, *M;
  QTimer t;
private slots:
  void draw();
};
#endif // MAINWINDOW_H
```

Главный класс приложения `MainWindow` помимо описания пользовательского интерфейса `ui` содержит размер поля `w` $\times$ `h` и массивы `P` и `M` для хранения и отображения состояния поля. Реализован только один метод `draw`, обеспечивающий вывод следующего поколения. Результаты вычислений выводятся с задержкой в 200 мс, что реализовано с помощью таймера `t`.

```
// файл "mainwindow.cpp"
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QPixmap>
```

```
#include <QPainter>
#include <QLabel>
#include "life.h"
#include <iostream>

MainWindow::MainWindow(QWidget *parent) :
  QMainWindow(parent),
  ui(new Ui::MainWindow)
{
  w = 48;
  h = 48;
  t.setInterval(200);
  connect(&t, SIGNAL(timeout()), SLOT(draw()));
  P = new uint[w * h];
  M = game_init(w, h, P);
  ui->setupUi(this);
  t.start();
}

MainWindow::~MainWindow()
{
  delete ui;
}

void MainWindow::draw()
{
  game_step(w, h, M, P);
  uint sc = 10;
  QPixmap pic(w*sc, h*sc);
  QPainter p(&pic);
  pic.fill(Qt::white);
  for (uint i=0; i<w; i++)
    for (uint j=0; j<h; j++)
      if (P[j*w+i])
        p.drawRect(i * sc, j * sc, sc, sc);
  ui->label->setPixmap(pic);
  ui->label->resize(pic.size());
}
```

Функция `draw` обеспечивает закраску поля согласно значениям массива `P`.

6.4. Практическое задание

Цель работы – создать программу на CUDA, обеспечивающую параллельную реализацию алгоритма моделирования клеточного автомата игры «Жизнь».

Ход работы:

1. Изучите теоретический материал.
2. Используя приведенные примеры программ, создайте свою программу, которая будет проводить расчет состояния клеточного автомата и выводить результат моделирования на экран.
3. Внесите модификацию в реализованную модель согласно выбранному варианту (табл. 4).

Таблица 4

Варианты заданий

№	*Модификация правил игры*
1	Изменить поле на ограниченный прямоугольник с размером не кратным 16
2	Сделать возможным изменение размера поля в процессе выполнения модели
3	Реализовать трехцветное поле. Третий цвет используется для обозначения пустой клетки, которая на прошлом ходе была «живой»
4	Изменить правило появления «живой» клетки. Появляется «живая» клетка, если число соседей 3 или 4. Исчезает клетка, если соседей менее 3 или более 5
5	Изменить правило появления «живой» клетки. Появляется «живая» клетка, если число соседей 2 или 3. Исчезает клетка, если соседей менее 2 или более 4

№	*Модификация правил игры*
6	Реализовать трехцветное поле. Третий цвет используется для обозначения «живой» клетки, которая на прошлом ходе была пустой
7	Реализовать трехмерное поле. Продумать способ отображения поля. Подобрать правила появления и исчезновения «живых» клеток
8	Обеспечить блокировку клеток, используя третий цвет для обозначения клетки, в которой появление новой клетки невозможно
9	Реализовать возможность создания начальной конфигурации клеток с помощью графического интерфейса
10	Обеспечить возможность завершения моделирования и изменения конфигурации клеток
11	Реализовать возможность случайного появления «живой» клетки на поле на каждом шаге без учета правил
12	Создать модель конкурирующих популяций. Третий цвет используется для обозначения конкурирующей клетки. Подправить правила появления и исчезновения «живых» клеток. Для появления новой необходимо 3 родственные клетки. Исчезновение происходит, если соседей больше 5, учитывая обе популяции
13	Использовать окрестность фон Неймана для поиска соседей. Подправить правило появления и исчезновения клеток
14	Использовать окрестность Мура радиуса 2. Подобрать подходящее правило появления и исчезновения клеток
15	Использовать окрестность фон Неймана радиуса 2. Подобрать подходящее правило появления и исчезновения клеток
16	Реализовать трехмерное поле. Продумать способ отображения поля. Использовать трехмерную окрестность фон Неймана для поиска соседей
17	Реализовать трехмерное поле. Противоположные края поля склеить

№	*Модификация правил игры*
18	Создать модель «хищник–жертва». Первый цвет используется для обозначения пустой клетки. Второй – для обозначения жертвы с прежними правилами. Третий – для обозначения «хищной» клетки со следующими правилами: появление новой, когда есть 1 «хищный» сосед и 2 клетки жертвы; исчезновение, когда любых соседей более 4
19	Реализовать активность поля. Имеется два состояния поля: активное и пассивное. На активном поле клетка может существовать, если соседей от 1 до 5. На пассивном – как в стандартных правилах. Обеспечить возможность создания карты активности поля пользователем
20	Реализовать трехцветное поле. Третий цвет используется для обозначения новой клетки. Эта клетка на следующем шаге становится обычной «живой», но на текущем не используется для вычисления соседей
21	Реализовать четырехцветное поле. Третий цвет как в варианте 20. Четвертый цвет – старые клетки. Они не влияют на появление новых, но влияют на исчезновение
22	Реализовать возрастную модель клеток. После рождения молодая клетка не используется для появления новых клеток в течение 10 ходов. Затем она становится взрослой и в течение 30 ходов участвует в появлении новых клеток. Потом она становится старой и в течение 20 ходов не участвует в появление новых клеток, затем исчезает. Подкорректировать правило появления и исчезновения клеток
23	Реализовать возможность запаздывания процесса исчезновения клеток. Если клетка должна была исчезнуть этим ходом, то она меняет свой цвет, а на следующем шаге исчезнет, если ситуация не изменится в сторону сохранения клетки
24	Объединить варианты 18 и 19
25	Предложить собственную модификацию правил игры, несовпадающую ни с одним из перечисленных вариантов

Контрольные вопросы

1. Почему алгоритмы вычисления клеточных автоматов хорошо поддаются распараллеливанию?

2. Поясните причины применения параллельных вычислений в программах моделирования.

3. Объясните, что означает просмотр соседей клетки с точки зрения параллельных вычислений?

4. Почему необходимо делать копию массива состояний клеток перед их модификацией на каждом шаге?

5. Является ли программа, написанная на CUDA совместно с Qt, кроссплатформенной?

6. Для чего используется поле в виде тора при моделировании клеточных автоматов?

Рекомендуемая литература

1. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. М. : ДМК, 2010.

2. *Виноградов В.С. и др.* Опыт использования технологии CUDA для реализации турбодекодера // Техника радиосвязи. 2014. № 1(21). С. 3–23.

3. *Коробицын В.В., Белозеров А.С.* Реализация вычислений на графическом процессоре с использованием платформы NVIDIA CUDA // Программные продукты и системы. 2010. № 1. С. 62–64.

4. *Коробицын В.В., Ильин С.С.* Реализация симметричного шифрования по алгоритму ГОСТ–28147 на графическом процессоре с использованием технологии CUDA // Информационные технологии. 2011. № 4. С. 41–46.

5. *Коробицын В.В. и др.* Модель программирования CUDA : учебник. Омск : Изд-во Ом. гос. ун-та, 2012.

6. *Коробицын В.В., Лыфарь Д.А.* Параллельные алгоритмы реляционного соединения на графическом процессоре // Наука и образование : электронное научно-техническое издание. 77-30569/234879. № 10, октябрь 2011. URL: http://technomag.edu.ru/doc/234879.html (01.10.2015).

7. *Коробицын В.В., Суравикин А.Ю.* Оценка производительности связывания CUDA с графическими API на примере

задачи SAXPY // Математические структуры и моделирование. Омск, 2010. Вып. 21. С. 107–114.

8. *Коробицын В.В., Суравикин А.Ю.* Построение изоповерхности на основе октодеревьев для визуализации потока жидкости // Вестник Омского университета. 2010. № 4. С. 164–167.

9. *Коробицын В.В., Фролова Ю.В.* Основы программирования алгоритмов компьютерной графики : учебно-методическое пособие. Омск : Изд-во Ом. гос. ун-та, 2010.

10. *Коробицын В.В., Фролова Ю.В.* Параллельная реализация на CUDA метода последовательных приближений для численного решения систем обыкновенных дифференциальных уравнений // Вестник Омского университета. 2011. № 4(62). С. 157–163.

11. *Gardner M.* Mathematical Games - The fantastic combinations of John Conway's new solitare game "life"// Scientific American. № 223. P. 120–123.

12. GPU gems 3 / eds. H. Nguyen. Boston: Pearson Education, 2008. URL: http://developer.nvidia.com/object/gpu-gems-3.html (5.10.2009).

13. *Harris M.* CUDAPP library. NVIDIA Corporation. URL: http://gpgpu.org/developer/cudapp.

14. *Harris M.* Parallel Prefix Sum (Scan) with Cuda. NVIDIA corp.

15. *Mandelbrot B.* The Fractal Geometry of Nature. W.H. Freeman and Co., 1982.

16. *He B., Fang W., Luo Q.* Mars: A MapReduce Framework on Graphics Processors. Microsoft research.

17. *Kilgariff E., Fernando R.* The GeForce 6 Series GPU Architecture // GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Eds. Pharr M., Fernando R. Addison Wesley Professional, 2005. P. 471–491.

18. *Kirk D.B., Hwu W.W.* Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010. 280 p.

19. NVIDIA CUDA Programming Guide 1.1. URL: http://www.nvidia.com/object/cuda_develop.html (01.04.2009).

20. NVIDIA OpenGL SDK Guide. URL: http://developer.download.nvidia.com/SDK/10.5/opengl/OpenGL_SDK_Guide.pdf (5.10.2009).

Учебное издание

Коробицын *Виктор Викторович,*
Фролова *Юлия Владимировна*

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА CUDA

Учебно-методическое пособие

Редактор *М.Е. Жумабаева*
Технический редактор *М.В. Быкова*
Дизайн обложки *З.Н. Образова*

Подписано в печать 05.10.2016. Формат бумаги 60х84 1/16.
Печ. л. 4,25. Усл. печ. л. 3,95. Уч.-изд. л. 3,0.
Тираж 70 экз. Заказ 135.

Издательство Омского государственного университета
644077, Омск-77, пр. Мира, 55а

Отпечатано на полиграфической базе ОмГУ
644077, Омск-77, пр. Мира, 55а