Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №6 по теме: «Программирование графических процессоров (NVIDIA CUDA)»

**Выполнил:**  
студент группы ПВ-31  
Адаменко И. И.

**Проверил:**к. т. н., доцент  
Михелёв В. М.

Белгород  
2015

**Цель работы:** получить практический навык использования технологии NVIDIA CUDA при решении прикладных задач.

# Теоретическая часть

CUDA — это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).

На сегодняшний день продажи CUDA процессоров достигли миллионов, а разработчики программного обеспечения, ученые и исследователи широко используют CUDA в различных областях, включая обработку видео и изображений, вычислительную биологию и химию, моделирование динамики жидкостей, восстановление изображений, полученных путем компьютерной томографии, сейсмический анализ, трассировку лучей и многое другое.

Направление вычислений эволюционирует от «централизованной обработки данных» на центральном процессоре до «совместной обработки» на CPU и GPU. Для реализации новой вычислительной парадигмы компания NVIDIA изобрела архитектуру параллельных вычислений CUDA, на данный момент представленную в графических процессорах GeForce, ION, Quadro и Tesla и обеспечивающую необходимую базу разработчикам ПО.

Говоря о потребительском рынке, стоит отметить, что почти все основные приложения для работы с видео уже оборудованы, либо будут оснащены поддержкой CUDA-ускорения, включая продукты от Elemental Technologies, MotionDSP и LoiLo.

Область научных исследований с большим энтузиазмом встретила технологию CUDA. К примеру, сейчас CUDA ускоряет AMBER, программу для моделирования молекулярной динамики, используемую более 60 000 исследователями в академической среде и фармацевтическими компаниями по всему миру для сокращения сроков создания лекарственных препаратов.

На финансовом рынке компании Numerix и CompatibL анонсировали поддержку CUDA в новом приложении анализа риска контрагентов и достигли ускорения работы в 18 раз. Numerix используется почти 400 финансовыми институтами.

Показателем роста применения CUDA является также рост использования графических процессоров Tesla в GPU вычислениях. На данный момент более 700 GPU кластеров установлены по всему миру в компаниях из списка Fortune 500, таких как Schlumberger и Chevron в энергетическом секторе, а также BNP Paribas в секторе банковских услуг.

Благодаря недавно выпущенным системам Microsoft Windows 8.1 и Apple Mac OS X Snow Leopard, вычисления на GPU займут свои позиции в секторе массовых решений. В этих новых операционных системах GPU предстанет не только графическим процессором, но также и универсальным процессором для параллельных вычислений, работающим с любым приложением.

Платформа параллельных вычислений CUDA® обеспечивает набор расширений для языков C и С++, позволяющих выражать как параллелизм данных, так и параллелизм задач на уровне мелких и крупных структурных единиц. Программист может выбрать средства разработки: языки высокого уровня, такие как C, C++, Fortran или же открытые стандарты, такие как директивы OpenACC. Платформа параллельных вычислений CUDA используется на сегодняшний день в тысячах GPU-ускоренных приложений и тысячах опубликованных научных статьях.

# Практическая часть

# Задание №1

Разработать программу для вычисления значения функции на отрезке с шагом , где и составить таблицу времени табулирования на CPU и GPU.

Функция:

## Итеративный алгоритм решения

В итеративном алгоритме мы, зная количество шагов для табулирования функции, запускаем цикл от 0 до S, где S — количество шагов. Вычисляем и заносим вычисленные значения в массив. Код итеративного алгоритма следующий:

1. **for** (**long** **long** i = 0; i < countStep; i++)
2. {
3. result[i] = f(1 + h \* i);
4. }

## Параллельный алгоритм решения

Для параллельного выполнения данного алгоритма с помощью технологии CUDA нам необходимо определить, сколько потоков понадобится для вычислений. Пусть каждый шаг табулирования у нас будет выполняться в собственном потоке. Тогда, поскольку имеющаяся карта поддерживает максимальное количество потоков в блоке равное 512, то необходимо посчитать количество блоков. Однако, тут тоже имеется ограничение — не более 65 535 блоков по каждому измерению сетки, а значит необходимо это предусмотреть и заранее вычислить сколько у нас будет строк.

В каждой строке сетки мы имеем 65 535 \* 512 = 33 553 920 потоков.

Вычислив всё это мы можем запустить программу вычисления функции с помощью CUDA, вызвав её с помощью следующего кода:

1. kernel<<<gridDim, blockDim>>>(countStep, h, resultDev);

где:

* gridDim — структура, описывающая количество блоков в сетке по каждому измерению;
* blockDim — структура, описывающая количество потоков в блоке по каждому измерению.

Данная функция будет выполнена каждым потоком. Однако, поскольку каждый поток будет вычислять значение функции на определённом шаге, нам нужно как-то высчитать номер потока, чтобы корректно сохранить значения. Узнать номер потока можно по следующей формуле:

1. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS + (blockIdx.x \* blockDim.x + threadIdx.x);

где:

* blockIdx — индекс блока внутри сетки по каждому измерению;
* threadIdx — индекс потока внутри блока по каждому измерению;
* MAX\_BLOCK — ограничение на количество блоков в строке сетки (те самые 65 535);
* MAX\_THREADS — максимальное количество потоков в блоке (в данном случае 512).

Сама функция kernel выглядит следующим образом:

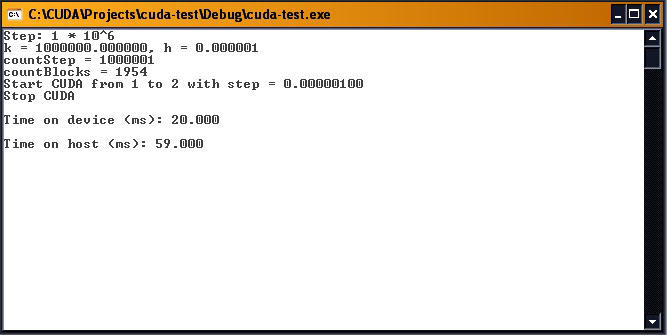
1. \_\_global\_\_ **void** kernel(**long** **long** countStep, **float** h, **float** \*result)
2. {
3. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS + (blockIdx.x \* blockDim.x + threadIdx.x);
5. **if** (step > countStep) {
6. **return**;
7. }
9. result[step] = f(1 + h \* step);
10. }

## Тестирование

|  |  |  |
| --- | --- | --- |
| **k** | **CPU (мс)** | **GPU (мс)** |
| **N × 104** | 0 | 0 |
| **N × 105** | 11 | 10 |
| **N × 106** | 74 | 24 |
| **N × 107** | 663 | 37 |
| **N × 108** | 6 875 | 1 204 |

График результата (выше — медленнее, а значит хуже):

Скриншот работы программы:



## Вывод

Как видно из графика и таблицы замеров работы программы, GPU даёт бо́льший прирост производительности при бо́льших размерах данных. При малом количестве итераций использование GPU не так эффективно, поскольку его ресурсы используются не полностью и большая часть времени тратится на распараллеливание вычислений, нежели на сами вычисления.

## Исходный код

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include <time.h>
5. #include <cuda\_runtime.h>
7. #define N 1
8. #define MAX\_BLOCKS 65535
10. /\* максимальное количество потоков в блоке;
11. \* получено так:
12. \* cudaGetDeviceProperties(&deviceProp, i);
13. \* printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);
14. \*/
15. #define MAX\_THREADS 512
17. \_\_device\_\_ \_\_host\_\_ **float** f(**float** x)
18. {
19. **return** sqrtf(x - 1) + 1 / (x - 3);
20. }
22. \_\_global\_\_ **void** kernel(**long** **long** countStep, **float** h, **float** \*result)
23. {
24. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS +  
     (blockIdx.x \* blockDim.x + threadIdx.x);
26. **if** (step > countStep) {
27. **return**;
28. }
30. result[step] = f(1 + h \* step);
31. }
33. \_\_host\_\_ **void** calculateOnHost(**long** **long** countStep, **float** h, **float** \*result)
34. {
35. **for** (**long** **long** i = 0; i < countStep; i++)
36. {
37. result[i] = f(1 + h \* i);
38. }
39. }
41. **int** main(**int** argc, **char**\*\* argv)
42. {
43. **int** multiplier = 4;
44. printf("Step: 1 \* 10^%d\n", multiplier);
46. **float** k = N \* powf(10.0, multiplier);
47. **float** h = N / k;
48. printf("k = %f, h = %f\n", k, h);
50. // вычисляем количество шагов
51. **long** **long** countStep = N / h + 1;
53. // подсчитываем количество блоков
54. **long** **long** countBlocks = countStep / MAX\_THREADS;
56. // если после деления остался остаток -- добавляем ещё блок
57. **if** (countStep % MAX\_THREADS != 0) {
58. countBlocks++;
59. }
61. **int** countY = 1;
62. **if** (countBlocks > MAX\_BLOCKS) {
63. countY = 1 + countBlocks / MAX\_BLOCKS;
64. countBlocks = MAX\_BLOCKS;
65. }
67. // высчитываем количество блоков и строк в сетке
68. // def: dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1
69. dim3 gridDim = dim3(countBlocks, countY);
70. dim3 blockDim = dim3(MAX\_THREADS);
72. **float** \*result = **new** **float**[countStep];
73. **float** \*resultDev;
74. **long** **long** size = countStep \* **sizeof**(**float**);
76. printf("countStep = %d\n", countStep);
77. printf("countBlocks = %d\n", countBlocks);
79. /\* CUDA START \*/
80. result = **new** **float**[countStep];
82. // инициализируем события, чтобы потом посчитать время
83. cudaEvent\_t start, stop;
84. cudaEventCreate(&start);
85. cudaEventCreate(&stop);
87. // запускаем событие начала
88. cudaEventRecord(start, 0);
89. // выделяем память
90. cudaMalloc((**void**\*\*) &resultDev, size);
91. printf("Start CUDA from %d to %d with step = %0.8f\n", 1, N + 1, h);
92. // запускаем вычисления
93. kernel<<<gridDim, blockDim>>>(countStep, h, resultDev);
95. // барьерная синхронизация
96. cudaDeviceSynchronize();
97. // копируем данные в result из resultDev размера size из GPU
98. cudaMemcpy(result, resultDev, size, cudaMemcpyDeviceToHost);
99. // чистим память
100. cudaFree(resultDev);
102. // запускаем событие завершения работы
103. cudaEventRecord(stop, 0);
104. // ждём, пока всё завершится
105. cudaEventSynchronize(stop);
106. printf("Stop CUDA\n");
108. **float** totalTimeDevice;
109. // вычисляем время
110. cudaEventElapsedTime(&totalTimeDevice, start, stop);
112. printf("\nTime on device (ms): %1.3lf\n", totalTimeDevice);
114. // удаляем события
115. cudaEventDestroy(start);
116. cudaEventDestroy(stop);
118. // чистим память
119. free(result);
120. result = NULL;
122. /\* CUDA END \*/
124. /\* HOST START \*/
126. // засекаем время
127. **time\_t** startTimeHost = clock();
129. result = **new** **float**[countStep];
131. // подсчитываем
132. calculateOnHost(countStep, h, result);
134. // считаем разницу во времени
135. **time\_t** endTimeHost = clock();
136. **double** totalTimeHost = (**double**)(endTimeHost - startTimeHost) /  
      CLOCKS\_PER\_SEC \* 1000;
138. printf("\nTime on host (ms): %1.3lf\n", totalTimeHost);
140. /\* HOST END \*/
142. getchar();
144. // чистим память
145. free(result);
146. cudaDeviceReset();
148. **return** 0;
149. }

# Задание №2

Разработать программу для нахождения минимального значения среди элементов вектора.

## Итеративный алгоритм решения

В итеративном алгоритме мы в цикле проходим по массиву, сравниваем каждый элемент с минимальным (которое изначально равно максимальному значению типа), если минимальный больше — изменяем его. В коде это выглядит вот так:

1. **int** min = INT\_MAX;
2. **for** (**long** **long** i = 0; i < countStep; i++)
3. {
4. **if** (min > vector[i]) {
5. min = vector[i];
6. }
7. }

## Параллельный алгоритм решения

Для параллельного выполнения данного алгоритма с помощью технологии CUDA нам необходимо определить, сколько потоков понадобится для вычислений. Для этого определим костанту PARTS, которая будет отвечать за количество итераций, производимых в каждом потоке. Тогда, поскольку имеющаяся карта поддерживает максимальное количество потоков в блоке равное 512, то необходимо посчитать количество блоков. Однако, тут тоже имеется ограничение — не более 65 535 блоков по каждому измерению сетки, а значит необходимо это предусмотреть и заранее вычислить сколько у нас будет строк.

В каждой строке сетки мы имеем 65 535 \* 512 = 33 553 920 потоков.

Однако, мы не можем просто найти в каждом потоке минимальное значение из PARTS. После этого нам нужно объединить все найденные минимальные значения, а затем найти минимальное значение из них. При большом объёме данных эти шаги тоже можно распараллеливать.

Вычислив всё это мы можем запустить программу вычисления функции с помощью CUDA, вызвав её с помощью следующего кода:

1. kernel<<<gridDim, blockDim>>>(count, vector, resultDev);

где:

* gridDim — структура, описывающая количество блоков в сетке по каждому измерению;
* blockDim — структура, описывающая количество потоков в блоке по каждому измерению.

Данная функция будет выполнена каждым потоком. Однако, поскольку каждый поток будет вычислять значение функции на определённом шаге, нам нужно как-то высчитать номер потока, чтобы корректно сохранить значения. Узнать номер потока можно по следующей формуле:

1. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS + (blockIdx.x \* blockDim.x + threadIdx.x);

где:

* blockIdx — индекс блока внутри сетки по каждому измерению;
* threadIdx — индекс потока внутри блока по каждому измерению;
* MAX\_BLOCK — ограничение на количество блоков в строке сетки (те самые 65 535);
* MAX\_THREADS — максимальное количество потоков в блоке (в данном случае 512).

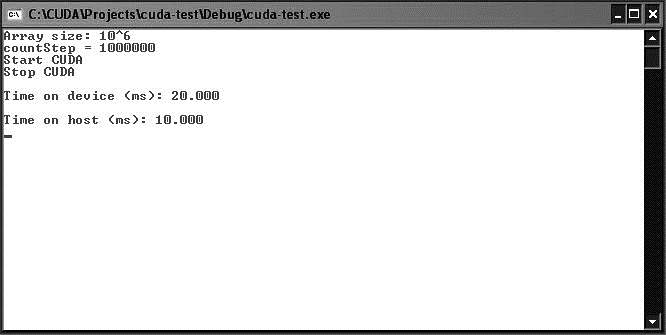
Сама функция kernel выглядит следующим образом:

1. \_\_global\_\_ **void** kernel(**long** **long** countStep, **int** \*vector, **int** \*result)
2. {
3. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS + (blockIdx.x \* blockDim.x + threadIdx.x);
5. **if** (step > countStep) {
6. **return**;
7. }
9. **int** min = INT\_MAX;
10. **for** (**long** **long** i = step \* PARTS; i < step + PARTS; i++)
11. {
12. **if** (min > vector[i]) {
13. min = vector[i];
14. }
15. }
17. result[step] = min;
18. }

## Тестирование

|  |  |  |
| --- | --- | --- |
| **k** | **CPU (мс)** | **GPU (мс)** |
| **N × 104** | 0 | 0 |
| **N × 105** | 0 | 10 |
| **N × 106** | 11 | 18 |
| **N × 107** | 118 | 30 |
| **N × 108** | 728 | 92 |

График результата (выше — медленнее, а значит хуже):

Скриншот работы программы:

## Вывод

Как видно из графика и таблицы замеров работы программы, GPU даёт бо́льший прирост производительности при бо́льших размерах данных. При малом количестве итераций использование GPU не так эффективно, поскольку его ресурсы используются не полностью и большая часть времени тратится на распараллеливание вычислений, нежели на сами вычисления.

## Исходный код

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include <time.h>
5. #include <climits>
6. #include <cuda\_runtime.h>
8. #define MAX\_BLOCKS 65535
10. /\* максимальное количество потоков в блоке;
11. \* получено так:
12. \* cudaGetDeviceProperties(&deviceProp, i);
13. \* printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);
14. \*/
15. #define MAX\_THREADS 512
17. // количество итераций в каждом потоке
18. #define PARTS 1000
20. \_\_global\_\_ **void** kernel(**long** **long** countStep, **int** \*vector, **int** \*result)
21. {
22. **long** **long** step = blockIdx.y \* MAX\_BLOCKS \* MAX\_THREADS +  
     (blockIdx.x \* blockDim.x + threadIdx.x);
24. **if** (step > countStep) {
25. **return**;
26. }
28. // находим минимальное
29. **int** min = INT\_MAX;
30. **for** (**long** **long** i = step \* PARTS; i < step + PARTS; i++)
31. {
32. **if** (min > vector[i]) {
33. min = vector[i];
34. }
35. }
37. result[step] = min;
38. }
40. \_\_host\_\_ **int** calculateOnHost(**long** **long** countStep, **int** \*vector)
41. {
42. // находим минимальное
43. **int** min = INT\_MAX;
44. **for** (**long** **long** i = 0; i < countStep; i++)
45. {
46. **if** (min > vector[i]) {
47. min = vector[i];
48. }
49. }
51. **return** min;
52. }
54. **int** main(**int** argc, **char**\*\* argv)
55. {
56. **int** multiplier = 6;
57. printf("Array size: 10^%d\n", multiplier);
59. **long** **long** countStep = (**long** **long**)pow(10.0, multiplier);
61. // объявлем массив и заполняем его данными
62. **int** \*vector = **new** **int**[countStep];
63. srand(time(NULL));
64. **for** (**long** **long** i = 0; i < countStep; i++)
65. {
66. vector[i] = rand();
67. }
69. // подсчитываем количество блоков
70. **long** **long** countBlocks = countStep / PARTS;
72. // если после деления остался остаток -- добавляем ещё блок
73. **if** (countStep % PARTS != 0) {
74. countBlocks++;
75. }
77. // высчитываем количество блоков и строк в сетке
78. **int** countY = 1;
79. **if** (countBlocks > MAX\_BLOCKS) {
80. countY = 1 + countBlocks / MAX\_BLOCKS;
81. countBlocks = MAX\_BLOCKS;
82. }
84. // def: dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1
85. dim3 gridDim = dim3(countBlocks, countY);
86. dim3 blockDim = dim3(PARTS);
88. **long** **long** size = countStep \* **sizeof**(**int**);
90. printf("countStep = %d\n", countStep);
92. /\* CUDA START \*/
94. // инициализируем события, чтобы потом посчитать время
95. cudaEvent\_t start, stop;
96. cudaEventCreate(&start);
97. cudaEventCreate(&stop);
99. // запускаем событие начала
100. cudaEventRecord(start, 0);
102. printf("Start CUDA\n");
104. **int** count = countStep;
105. **while** (count) {
106. **int** \*resultDev;
107. // выделяем память
108. **long** **long** size = count \* **sizeof**(**int**);
109. cudaMalloc((**void**\*\*) &resultDev, size);
110. // запускаем вычисления
111. kernel<<<gridDim, blockDim>>>(count, vector, resultDev);
112. // барьерная синхронизация
113. cudaDeviceSynchronize();
114. // копируем данные в vector из resultDev размера size из GPU
115. cudaMemcpy(vector, resultDev, size, cudaMemcpyDeviceToHost);
116. // чистим память
117. cudaFree(resultDev);
118. count /= PARTS;
119. }
121. // запускаем событие завершения работы
122. cudaEventRecord(stop, 0);
123. // ждём, пока всё завершится
124. cudaEventSynchronize(stop);
125. printf("Stop CUDA\n");
127. **float** totalTimeDevice;
128. // вычисляем время
129. cudaEventElapsedTime(&totalTimeDevice, start, stop);
130. printf("\nTime on device (ms): %1.3lf\n", totalTimeDevice);
132. // удаляем события
133. cudaEventDestroy(start);
134. cudaEventDestroy(stop);
136. // чистим память
137. free(vector);
138. vector = NULL;
140. /\* CUDA END \*/
142. /\* HOST START \*/
143. vector = **new** **int**[countStep];
144. **for** (**long** **long** i = 0; i < countStep; i++)
145. {
146. vector[i] = rand();
147. }
149. // засекаем время
150. **time\_t** startTimeHost = clock();
152. // подсчитываем
153. calculateOnHost(countStep, vector);
155. // считаем разницу во времени
156. **time\_t** endTimeHost = clock();
157. **double** totalTimeHost = (**double**)(endTimeHost - startTimeHost) / CLOCKS\_PER\_SEC \* 1000;
159. printf("\nTime on host (ms): %1.3lf\n", totalTimeHost);
161. /\* HOST END \*/
163. getchar();
165. // чистим память
166. free(vector);
167. cudaDeviceReset();
169. **return** 0;
170. }