Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА АНАЛИЗА ДАННЫХ И ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Направление: 09.03.03 – Прикладная информатика

ОТЧЁТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ ПО ПРЕДМЕТУ «ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ CUDA»

Работа завершена:
Студент 4 курса группы 09-951 Колесников Д.А.

Работа проверена:
Доцент Тумаков Д.Н.

Казань — 2023

Содержание

1. Формулировка задания

Реализовать сортировку на CPU и на GPU. Определить количество затрачиваемых операций и времени. Сравнить результаты. Для сортировки использовать большие объёмы данных.

Оборудование:

Все вычисления проводились на моём компьютере:

CPU: Intel Core i7-8750H, турбобуст включён. Всегда использовалось только 1 ядро.

GPU: NVIDIA GeForce GTX 1050 Ti Mobile. Всегда использовался только 1 блок. На блок задействованы все 1024 потока.

RAM: 8+8ГиБ, 2 канала, 2666МГц

Программное обеспечение:

C: clang 15.0.7

CUDA C: nvcc 12.1. Версия архитектуры видеокарты 6.1

OS: Linux x86-64, версия ядра 6.2.7-arch1-1

2. Теория

Далее будут реализовываться 4 разных алгоритма сортировки. 2 на CPU, 2 аналогичных по смыслу на GPU. Рассмотрим их.

2.1. Сортировка пузырьком

Один из простейших алгоритмов сортировки. У алгоритма столько итераций, сколько элементов в массиве. Для каждой итерации нужно проходить по всему массиву, сравнивая 2 соседних числа. Если число слева больше правого (для сортировки по возрастанию), то нужно поменять их местами. На каждой итерации на своё место встаёт 1 элемент - последний непоставленный ранее больший.

Алгоритмическая сложность сортировки - $O(n^2)$

2.2. Quick-sort

Сортировка Хоара. Выбирается опорный элемент. Элементы массива разбиваются так, чтобы меньшие элементы были перед опорным, большие или равные - после. Для подмассивов с обеих сторон проделать то же самое.

Алгоритмическая сложность сортировки - $O(n * \log(n))$

2.3. Параллельная сортировка пузырьком

Идея та же, что и в обычном пузырьке - двигаем соседние элементы. Но при одном потоке мы ограничены проходом с одной стороны до другой. Здесь у нас 1024 потока. На каждое можно повесить проверку. Сложность от этого не поменяется, но в лучшем случае у нас сортировка будет за $O(n)$ по времени. Если элементов меньше 2048, то каждый поток за раз обрабатывает элемент и соседний от него. Итого получаем, что времени нужно столько, чтобы по каждому элементу пройтись 1 раз.

Алгоритмическая сложность сортировки - $O(n^2)$

2.4. Битонная сортировка

Легче всего объяснить на картинке. А вот и она, Рисунок 1.

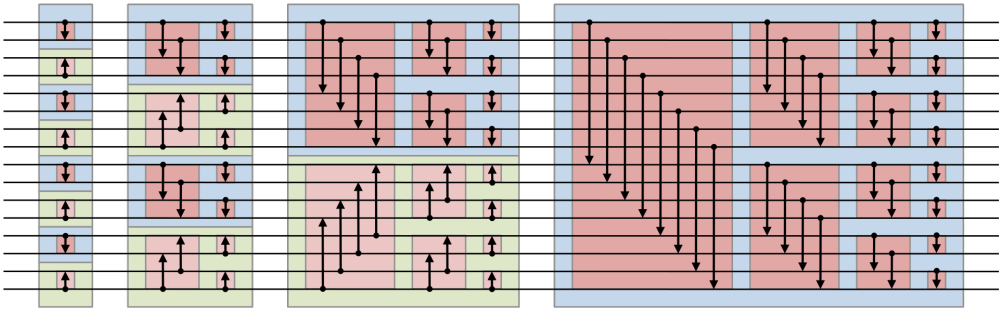

Рисунок 1. Битонная сортировка

Сразу бросается простота вычисления - глобальных шагов будет $\log(n)$. При этом каждый шаг содержит в себе аналог предыдущего. Синие прямоугольники - те, в которых элементы на следующем шаге будут идти по возрастанию. Зелёные - по убыванию. Стрелки показывают какой с каким элементы нужно менять. Если элемент у основания стрелки меньше того, на который она указывает, то их нужно поменять местами. Так достигается битонная последовательность. Сначала она не убывает, а потом не возрастает. На последнем шаге последовательность только возрастает. На первом - все элементы отсортированы в массиве только с собой.

Алгоритмическая сложность сортировки - $O(n * \log(n))$

3. Практика

3.1. Реализация сортировки пузырьком на CPU

Листинг 1. Сортировка пузырьком на CPU

```
long int sort(long int arr[]) {
    long int operations = 0;
    long int tmp = 0;

    for (long int i = 0; i < ARR_SIZE - 1; ++i) {
        for (long int j = 0; j < ARR_SIZE - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }

    return operations;
}
```

3.2. Реализация Quick-sort на CPU

Листинг 2. Quicksort на CPU

```
long int sort(long int arr[], long int first, long int last)
{
    long int operations = 0;
    long int i = first;
    long int j = last;
    long int pivot = last;
    long int tmp = 0;

    if (first < last) {
        while (i < j) {
            while (arr[i] <= arr[pivot] && i < last) {
                i++;
            }
            while (arr[j] > arr[pivot]) {
                j--;
            }
            if (i < j) {
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
```

```
21              }
22          }
23          tmp = arr[pivot];
24          arr[pivot] = arr[j];
25          arr[j] = tmp;
26          operations += sort(arr, first, j - 1);
27          operations += sort(arr, j + 1, last);
28      }
29      return operations;
30  }
```

3.3. Реализация параллельной сортировки пузырьком на GPU

Листинг 3. Параллельная пузырьковая сортировка на GPU

```
1  __global__ void sort (long int* data, unsigned long long* operations)
2  {
3      long int operationOnThread = ARR_SIZE / BLOCK_SIZE / 2 + 1;
4      for (long int i = 0; i < ARR_SIZE / 2 + 1; ++i) {
5          for (long int iOperation = 0; iOperation < operationOnThread; ++
               iOperation) {
6              long int realIndex = (BLOCK_SIZE * iOperation + threadIdx.x) *
                   2;
7              long int nextIndex = realIndex + 1;
8              if (nextIndex < ARR_SIZE) {
9                  if (data[realIndex] > data[nextIndex]) {
10                     long int tmp = data[realIndex];
11                     data[realIndex] = data[nextIndex];
12                     data[nextIndex] = tmp;
13                 }
14             }
15         }
16         __syncthreads();
17         for (long int iOperation = 0; iOperation < operationOnThread; ++
               iOperation) {
18             long int realIndex = (BLOCK_SIZE * iOperation + threadIdx.x) *
                   2 + 1;
19             long int nextIndex = realIndex + 1;
20             if (nextIndex < ARR_SIZE) {
21                 if (data[realIndex] > data[nextIndex]) {
22                     long int tmp = data[realIndex];
23                     data[realIndex] = data[nextIndex];
24                     data[nextIndex] = tmp;
25                 }
26             }
27         }
28         __syncthreads();
```

```
29      }
30  }
```

3.4. Реализация битонной сортировки на GPU

Листинг 4. Битонная сортировка на GPU

```
 1  __global__ void sort (long int* data, unsigned long long* operations)
 2  {
 3      long int arrSizeCopy = ARR_SIZE;
 4      int iterations = 0;
 5      while (arrSizeCopy > 0) {
 6          arrSizeCopy = arrSizeCopy >> 1;
 7          ++iterations;
 8      }
 9      long int fakeArrSize = 1 << iterations;
10      int direction = 0;
11      int half = 0;
12      long int tmp = 0;
13
14      for (int i = 0; i < iterations; ++i) {
15          long int rectSize = 1 << (i + 1);
16          long int halfRectSize = rectSize >> 1;
17
18          long int stableRectSize = rectSize;
19          while (rectSize > 1) {
20              for (long int iElement = threadIdx.x; iElement < fakeArrSize;
                    iElement += BLOCK_SIZE) {
21                  // -1 -          , 1 -
22                  direction = -1;
23                  if ((iElement / stableRectSize) % 2 == 0) {
24                      direction = 1;
25                  }
26
27                  // 0 -              , 1 -
28                  half = 1;
29                  if (iElement % rectSize < rectSize / 2) {
30                      half = 0;
31                  }
32                  if ((direction == 1) && (half == 0)) {
33                      if ((iElement < ARR_SIZE) && (iElement + halfRectSize
                            < ARR_SIZE)) {
34                          if (data[iElement] > data[iElement + halfRectSize
                                ]) {
35                              tmp = data[iElement + halfRectSize];
36                              data[iElement + halfRectSize] = data[iElement
                                    ];
```

```
37                              data[iElement] = tmp;
38                          }
39                      }
40                  }
41                  if ((direction == -1) && (half == 1)) {
42                      if ((iElement < ARR_SIZE) && (iElement - halfRectSize
                            < ARR_SIZE)) {
43                          if (data[iElement] > data[iElement - halfRectSize
                                ]) {
44                              tmp = data[iElement - halfRectSize];
45                              data[iElement - halfRectSize] = data[iElement
                                    ];
46                              data[iElement] = tmp;
47                          }
48                      }
49                  }
50              }
51              __syncthreads();
52              rectSize = rectSize >> 1;
53              halfRectSize = rectSize >> 1;
54          }
55      }
56  }
```

3.5. Дополнительно

Код из Листингов 5 и 6 был опущен из листингов выше. Потому что он много раз повторяется. Этот код компилируется только при выставлении макроса FULL. Из технологий CUDA здесь используется atomicAdd.

В коде на CPU возвращается количество затраченных операций. В коде для GPU это количество передаётся по указателю.

Листинг 5. Подсчёт количества операций в сортировках на CPU

```
1 #if defined(FULL)
2     ++operations;
3 #endif
```

Листинг 6. Подсчёт количества операций в сортировках на GPU

```
1 #if defined(FULL)
2     atomicAdd(operations, (unsigned long long) 1);
3 #endif
```

3.6. Сравнение

Время замерялось от начала сортировки до конца сортировки встроенными средствами. То есть время на запуск и чтение уже опущено. Также подсчёт операций не влияет на скорость вычислений. В каждой программе объявлен макрос FULL, который влияет на компиляцию кусков кода с подсчётом. То есть столбцы заполнялись двумя независимыми запусками.

Рассмитрим Таблицу 1. На начальном этапе в 1024 элемента всё неплохо. У Quicksort большое преимущество по операциям, но уже здесь видно отставание от сортировок на GPU. В 15 раз.

Название	Операции (шт)	Время (с)
CPU пузырёк	3.917.029	0.00251
CPU quicksort	724.128	0.00209
GPU параллельный пузырёк	8.951.704	0.00061
GPU битонная	2.473.857	0.00014

Таблица 1. Сравнение для массива из 1024 элементов

Рассмитрим Таблицу 2. Стало интереснее - Quicksort почти догнал битонную сортировку по операциям. Больше по ним преимущества нет. Теперь разница между CPU и GPU стала более выражена. Видна выраженная разница между пузырьком и параллельным пузырьком. За счёт 1024 потоков время значительно уменьшилось.

Название	Операции (шт)	Время (с)
CPU пузырёк	251.490.088	0,04554
CPU quicksort	11.682.064	0.03211
GPU параллельный пузырёк	102.992.488	0.00697
GPU битонная	13.266.457	0.00055

Таблица 2. Сравнение для массива из 4096 элементов

Рассмитрим Таблицу 3. Предел разумного использования пузырька. Почти 1 секунда. При этом битонная сортировка догнала пузырёк из Таблицы 1. Время почти такое же, элементов в 16 раз больше.

Рассмитрим Таблицу 4. Все сортировки кроме битонной перешагнули через 1 секунду. В остальном ничего интересного.

Рассмитрим Таблицу 5. Первый провал. Quicksort, основанный на рекурсии, не выдержал. Выдаёт ошибку на этих данных. Пузырёк после 60 мил-

Название	Операции (шт)	Время (с)
CPU пузырёк	1.006.372.810	0,84676
CPU quicksort	178.954.112	0.53797
GPU параллельный пузырёк	1.266.438.754	0.10325
GPU битонная	69.494.781	0.00249

Таблица 3. Сравнение для массива из 16384 элементов

Название	Операции (шт)	Время (с)
CPU пузырёк	16.110.614.796	15.40325
CPU quicksort	2.871.346.472	8.86929
GPU параллельный пузырёк	18.760.564.082	1.40010
GPU битонная	353.824.128	0.01258

Таблица 4. Сравнение для массива из 65536 элементов

лиардов операций выдал результат, но потратил на это минуту. Дальше считать на CPU смысла нет.

Название	Операции (шт)	Время (с)
CPU пузырёк	64.387.439.672	61.73878
CPU quicksort	-	-
GPU параллельный пузырёк	74.027.453.534	5.46474
GPU битонная	790.676.076	0.02783

Таблица 5. Сравнение для массива из 131072 элементов

Рассмитрим Таблицу 6. Параллельный пузырёк тоже перестал иметь смысл. Полторы минуты. При этом битонная сортировка дошла до уровня параллельного пузырька на 16384 элемента. Пузырёк дошёл до ответа, но потратил на это больше 16 минут. Можно ещё заметит, что примерно одинаковое количетво операций на CPU и GPU выполнились в пузырьках с разницей во времени в 10 раз.

Название	Операции (шт)	Время (с)
CPU пузырёк	1.030.784.790.368	992.33990
CPU quicksort	-	-
GPU параллельный пузырёк	1.172.264.597.452	91.28490
GPU битонная	3.883.140.794	0.128574

Таблица 6. Сравнение для массива из 524288 элементов

Дальнейшие эксперименты невозможны. Появлятся какое-то ограниче-

ние языка. Все 4 сортировки выдают ошибку.

Дополнительно можно заметить, что Quicksort работает не сильно быстрее пузырька. Обычно всего в 2 раза. Скорее всего, это связано с оптимизациями со стороны компилятора.

Заключение

Изучены алгоритмы сортировки, рассчитаные на параллельность. Реализованы 2 из них. Увиденно значительное преимущество больших вычислений на GPU перед CPU.

Исходные коды всех программ, в том числе и программы для генерации последовательностей, можно найти по ссылке:

https://github.com/zalimannard/homework-4/tree/main/cuda.