Кафедра прикладной математики и кибернетики

Выполнили:

Студенты 3 курса группы ИП-111  
Корнилов А.А.,  
Попов М.И.,

Толкач А.А.

Проверил:

Профессор кафедры ПМиК  
Малков Е.А.

ЛАБОРАТОРНАЯ РАБОТА №2

По дисциплине: «Программирование графических процессоров»

Сибирский государственный университет  
телекоммуникаций и информатики

Новосибирск, 2024

Министерство цифрового развития, связи  
и массовых коммуникаций Российской Федерации

**Задание:** 1. Определить при какой длине векторов имеет смысл распараллеливать операцию сложения, используя потоки CPU или GPU.

2. Определить оптимальное количество потоков POSIX для распараллеливания.

3. Определить зависимость времени выполнения операции сложения на GPU от длины векторов (выбирать количество нитей равным длине вектора).

**Цель:** начальное знакомство с распараллеливанием кода на GPU .

**Выполнение работы:**

Для первого задания были написаны программы для сложения n векторов для:

* Одного потока, линейная (лист. 1)
* для многопоточности, используя библиотеку thread (лист. 2)
* для вычисления на GPU, используя CUDA (лист. 3)

Для замеров времени используется STL библиотека Chrono, для вычисления с GPU еще и CudaEvent, в программа для примера n = 100000000, количество нитей 1024. Для вычислений с threads было взято максимальное количество потоков процесса с помощью thread::hardware\_concurrency(), в нашем случае это число было 12.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <thread>  #include <chrono>  using namespace std;  const int n = 100000000;  typedef std::chrono::milliseconds ms;  typedef std::chrono::nanoseconds ns;  int main() {  vector<float> a(n), b(n), c(n);  chrono::time\_point<chrono::system\_clock> start, end;  for (int i = 0; i < n; ++i) {  a[i] = i;  b[i] = i \* 2;  }    start = chrono::system\_clock::now();  for (int i = 0; i < n; ++i) {  c[i] = a[i] + b[i];  }  end = chrono::system\_clock::now();    cout << "Wasted time: " << chrono::duration\_cast<ms>(end - start).count() <<"ms" << endl  << chrono::duration\_cast<ns>(end - start).count() << "ns";  return 0;  } |

Листинг 1 – программа LR02\_1C.cpp

Результат работы программы:

|  |
| --- |
| Wasted time: 1195ms  1195286800ns |

|  |
| --- |
| #include <iostream>  #include <vector>  #include <thread>  #include <chrono>  using namespace std;  const int n = 100000000;  typedef std::chrono::milliseconds ms;  typedef std::chrono::nanoseconds ns;  void vectorAdd(const vector<float> &a, const vector<float> &b, vector<float> &c, int start, int end) {  for (int i = start; i < end; ++i) {  c[i] = a[i] + b[i];  }  }  int main() {  vector<float> a(n), b(n), c(n);  int numThreads = thread::hardware\_concurrency();  //int numThreads = 4;  chrono::time\_point<chrono::system\_clock> start, end;  for (int i = 0; i < n; ++i) {  a[i] = i;  b[i] = i \* 2;  }  vector<thread> threads;  for (int i = 0; i < numThreads; ++i) {  int start = i \* (n / numThreads);  int end = (i == numThreads - 1) ? n : (i + 1) \* (n / numThreads);  threads.emplace\_back(vectorAdd, ref(a), ref(b), ref(c), start, end);  }  start = chrono::system\_clock::now();  for (auto &thread : threads) {  thread.join();  }  end = chrono::system\_clock::now();  cout << "Wasted time: " << chrono::duration\_cast<ms>(end - start).count() << "ms" << endl  << chrono::duration\_cast<ns>(end - start).count() << "ns";  return 0;  } |

Листинг 1 – программа LR02\_2C.cpp

Результат работы программы:

|  |
| --- |
| Wasted time: 211ms  211703100ns |

|  |
| --- |
| #include "cuda\_runtime.h"  #include "device\_launch\_parameters.h"  #include <stdio.h>  #include <iostream>  #include <chrono>  using namespace std;  const int n = 100000000;  typedef std::chrono::milliseconds ms;  typedef std::chrono::nanoseconds ns;  \_\_global\_\_ void vectorAdd(const float\* a, const float\* b, float\* c, int n) {  int i = blockIdx.x \* blockDim.x + threadIdx.x;  if (i < n) {  c[i] = a[i] + b[i];  }  }  int main() {  float elapsedTime;  cudaEvent\_t start, stop;  chrono::time\_point<chrono::system\_clock> start\_chrono, end\_chrono;  float\* d\_a, \* d\_b, \* d\_c;  cudaMalloc((void\*\*)&d\_a, n \* sizeof(float));  cudaMalloc((void\*\*)&d\_b, n \* sizeof(float));  cudaMalloc((void\*\*)&d\_c, n \* sizeof(float));  float\* h\_a = new float[n];  float\* h\_b = new float[n];  for (int i = 0; i < n; ++i) {  h\_a[i] = i;  h\_b[i] = i \* 2;  }  cudaMemcpy(d\_a, h\_a, n \* sizeof(float), cudaMemcpyHostToDevice);  cudaMemcpy(d\_b, h\_b, n \* sizeof(float), cudaMemcpyHostToDevice);  // Вычисляем количество блоков и нитей на блок  int blockSize = 1024;  int numBlocks = n;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start, 0);  start\_chrono = chrono::system\_clock::now();  vectorAdd <<< numBlocks, blockSize >>> (d\_a, d\_b, d\_c, n);  cudaEventRecord(stop, 0);  end\_chrono = chrono::system\_clock::now();  cudaEventSynchronize(stop);  cudaEventElapsedTime(&elapsedTime, start, stop);  float\* h\_c = new float[n];  cudaMemcpy(h\_c, d\_c, n \* sizeof(float), cudaMemcpyDeviceToHost);  cout <<"CUDA Event time: "<< elapsedTime << endl  <<"Chrono time: "<< chrono::duration\_cast<ms>(end\_chrono - start\_chrono).count() << "ms"  << endl << chrono::duration\_cast<ns>(end\_chrono - start\_chrono).count() << "ns";  delete[] h\_a;  delete[] h\_b;  delete[] h\_c;  cudaFree(d\_a);  cudaFree(d\_b);  cudaFree(d\_c);  return 0;  } |

Листинг 1 – программа LR02\_1G.cu

Результат работы программы:

|  |
| --- |
| CUDA Event time: 0.06544  Chrono time: 0ms  84800ns |

Для вычисления оптимально метода и количества потоков были проведены замеры трех программам с количеством векторов: 2, 100, 10000 и 100000000. Результаты в таблице 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | single CPU | | 12 thread | | GPU | | |
|  | ms | ns | ms | ns | event | ms | ns |
| 2 | 0 | 100 | 1 | 1510700 | 0,079872 | 0 | 73600 |
| 0 | 200 | 1 | 1228000 | 0,058368 | 0 | 52400 |
| 0 | 100 | 1 | 1227300 | 0,067584 | 0 | 63900 |
| 0 | 100 | 1 | 1792000 | 0,095232 | 0 | 90800 |
| 0 | 100 | 1 | 1303300 | 0,092192 | 0 | 87600 |
| 100 | 0 | 1300 | 1 | 1223400 | 0,065536 | 0 | 59700 |
| 0 | 1400 | 1 | 1325500 | 0,057504 | 0 | 53600 |
| 0 | 1400 | 1 | 1157500 | 0,546816 | 0 | 51600 |
| 0 | 1300 | 1 | 1674700 | 0,007168 | 0 | 68100 |
| 0 | 1400 | 1 | 1251300 | 0,067584 | 0 | 62900 |
| 10000 | 0 | 183000 | 1 | 1701100 | 0,058368 | 0 | 53700 |
| 0 | 168000 | 1 | 1459400 | 0,06144 | 0 | 57100 |
| 0 | 127700 | 1 | 1486100 | 0,06144 | 0 | 55900 |
| 0 | 194200 | 1 | 1430200 | 0,056288 | 0 | 51500 |
| 0 | 117800 | 1 | 1302200 | 0,060416 | 0 | 56100 |
| 100000000 | 1193 | 1193753400 | 205 | 205361600 | 0,099648 | 0 | 88700 |
| 1181 | 1181136100 | 206 | 206149700 | 0,067264 | 0 | 79300 |
| 1191 | 1191234200 | 201 | 201283300 | 0,064896 | 0 | 77600 |
| 1202 | 1202666200 | 202 | 202233200 | 0,067968 | 0 | 84000 |
| 1197 | 1197465900 | 199 | 199579200 | 0,043424 | 0 | 70400 |

Таблица 1 – Замеры программ с разным количеством векторов.

В графиках ниже приведены визуальные сравнения работы трех программ по времени:

По графикам можно понять что в случае с вычисление процессором то для небольшого количества векторов (>1.000.000) можно использовать линейные вычисления, после 1.000.000 уже эффективней использовать вычисления с и пользованием многопоточности. В случае вычисление с GPU то её преимущество уже видно после 10.000 векторов

Таблица Excel будет приведена в том-же письме что и отчет, вычисления проводились на GTX 1050ti 4Gb