Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(ФГБОУ ВО «СибГУТИ»)

*Кафедра прикладной математики и кибернетики*

Расчетно-графическое задание по дисциплине

«Программирование графических процессов»

Выполнила:

студентка гр. ИП-514 Сахарчук Д.П,

Проверил:

ассистент кафедры ПМиК

Нужнов А.В.

Новосибирск

2018

**Оглавление**

[**Постановка задачи 3**](#_Toc514274580)

[**Теоретическая часть 3**](#_Toc514274581)

[***CUDA C* 3**](#_Toc514274582)

[***Thrust 3***](#_Toc514274583)

[***cuBLAS* 3**](#_Toc514274584)

[**Ход работы 5**](#_Toc514274585)

[**Листинг 7**](#_Toc514274586)

[***cuda.cu* 7**](#_Toc514274587)

[***thrust.cu* 8**](#_Toc514274588)

[***cublas.cu* 10**](#_Toc514274589)

[**Заключение 12**](#_Toc514274590)

# Постановка задачи

Провести сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS, «сырого» CUDA C кода.

# Теоретическая часть

## CUDA C

CUDA (англ. Compute Unified Device Architecture) — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

CUDA C (или точнее C++) – Один из языков программирования, использующих специальные расширения для обеспечения поддержки CUDA

CUDA один из основных производителей видео чипов разработали платформу, которая позволяет перенаправить вычисления с центрального на графические сопроцессоры. Данная платформа выполнена с учётом прямого доступа к аппаратным возможностям видеокарт. Она ликвидировала важные ограничения предыдущих моделей GPGPU (General-Purpose computation on GPUs). Платформа CUDA С-подобный язык со своим компилятором и библиотеками для вычисления на GPU. Написание оптимального кода вручную трудная задача, но благодаря технологии CUDA это позволяет достичь большого контроля над аппаратными возможностями видеочипа и, следовательно, большого прироста в скорости.

**Thrust**

Одной из самых красивых и гибких библиотек для CUDA является библиотека thrust. Основным отличием данной библиотеки от других распространенных библиотек для CUDA является то, что thrust - это библиотека, основанная на использовании шаблонов (template) языка С++.

Все классы и функции в этой библиотеки - шаблонные, все, что вам нужно для работы с этой библиотекой - это подключить соответствующие заголовочные файлы - в thrust нет никаких стандартных библиотечных файлов (.lib, .a, .dll, .so).

Ряд понятий и подходов thrust заимствовала из STL. thrust предоставляет в ваше распоряжение набор различных параллельных примитивов, таких как различные преобразования, сортировка, операции reduce и scan. Применяя thrust, многие действия могут быть записаны просто и понятно с использованием минимального объема кода.

**cuBLAS**

CUBLAS — CUDA вариант BLAS (Basic Linear Algebra Subprograms), предназначенный для вычислений задач линейной алгебры и использующий прямой доступ к ресурсам GPU.

Библиотека NVIDIA cuBLAS представляет собой быструю ускоренную с помощью GPU реализацию стандартных базовых подпрограмм линейной алгебры (BLAS). Используя API-интерфейс cuBLAS, можно ускорить работу приложений путем развертывания интенсивных вычислений на одном графическом процессоре или масштабирования и эффективного распространения работы в конфигурациях с несколькими GPU.

# 

# Ход работы

В качестве алгоритма линейной алгебры было взято скалярное произведение векторов. В качестве элементов входных матриц были взяты рандомные вещественные числа типа float.

Для тестирования библиотеки Thrust был разработан алгоритм, использующий её стандартные методы для распараллеливания задач.

В библиотеке cuBLAS есть готовое семейство функций cublas<T>gemm. Для нашего типа входных данных подошла функция cublasSgemm(), работающая с вещественными числами одинарной точности.

В качестве платформ для тестирования всех вышеперечисленных алгоритмов были взяты кластер СибГУТИ Oak c видеокартой NVIDIA GeForce GT 630 (для тестирования интерфейса CUDA).

В качестве тестов производительности были произведены серии вычислений (по 10 раз) векторов фиксированного размера и замеров времени их выполнения. Среднее время выполнения каждого алгоритма приведено в таблице ниже:

*Таблица. Временные характеристики перемножения векторов*

|  |  |  |  |
| --- | --- | --- | --- |
| N | CUDA C | Thrust | cuBLAS |
| 16 | 0.135424 | 0.132736 | 0.048192 |
| 32 | 0.136448 | 0.136832 | 0.048096 |
| 64 | 0.138272 | 0.140544 | 0.048288 |
| 128 | 0.136640 | 0.127520 | 0.048544 |
| 256 | 0.157184 | 0.131808 | 0.048608 |
| 512 | 0.233280 | 0.130560 | 0.049440 |

На основе проведенных испытаний можно сделать следующие выводы:

* Функции Thrust не годятся для скалярного перемножения векторов. Несмотря на свою параллельность, они не специализированы для данной операции, а значит библиотека может помочь только удобными векторами для работы с данными.
* cuBLAS практически не зависит от размера входных векторов, что делает его идеальным кандидатом на использование при большом объеме данных.
* «Сырой» CUDA код без учета коалесинга выдает самые быстрые результаты на маленьких размерах векторов, но при увеличении объема данных начинает сдавать позиции. Для N ≥ 512 стоит либо использовать более оптимизированные алгоритмы, либо предпочесть cuBLAS.

# 

# Листинг

## cuda.cu

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include <sys/time.h>  #include <cuda.h>  #include <cuda\_runtime.h>  \_\_global\_\_ void vec\_add(int n, float \*x, float \*y)  {  int i = blockIdx.x \* blockDim.x + threadIdx.x;  if (i < n)  y[i] = x[i] + y[i];  }  int main(int argc, char \*argv[])  {  srand(time(NULL));  int size = argc > 1 ? atoi(argv[1]) : 512;  int threadsPerBlockDim = argc > 2 ? atoi(argv[2]) : 32;  float \*h\_x = NULL, \*h\_y = NULL, \*rez = NULL;  float \*d\_x = NULL, \*d\_y = NULL;  h\_x = (float \*)malloc(size \* sizeof(float));  h\_y = (float \*)malloc(size \* sizeof(float));  rez = (float \*)malloc(size \* sizeof(float));  for (int i = 0; i < size; i++)  {  h\_x[i] = (float)(rand()) / RAND\_MAX;  h\_y[i] = (float)(rand()) / RAND\_MAX;  }  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start, 0);  cudaMalloc(&d\_x, size \* sizeof(float));  cudaMalloc(&d\_y, size \* sizeof(float));  cudaMemcpy(d\_x, h\_x, size \* sizeof(float), cudaMemcpyHostToDevice);  cudaMemcpy(d\_y, h\_y, size \* sizeof(float), cudaMemcpyHostToDevice);  dim3 blockDim(threadsPerBlockDim, threadsPerBlockDim, 1);  int blocksPerGridDimX = ceilf(size / (float)threadsPerBlockDim);  int blocksPerGridDimY = ceilf(size / (float)threadsPerBlockDim);  dim3 gridDim(blocksPerGridDimX, blocksPerGridDimY, 1);  vec\_add <<< gridDim, blockDim>>>(size, d\_x, d\_y);  cudaDeviceSynchronize();  cudaMemcpy(rez, d\_y, size \* sizeof(float), cudaMemcpyDeviceToHost);  cudaEventRecord(stop, 0);  cudaEventSynchronize(stop);  float elapsed\_time;  cudaEventElapsedTime(&elapsed\_time, start, stop);  printf("Elapsed time: %f\n", elapsed\_time);  cudaFree(d\_y);  cudaFree(d\_x);  free(h\_x);  free(h\_y);  cudaDeviceReset();  return 0;  } |

## thrust.cu

|  |
| --- |
| #include <thrust/host\_vector.h>  #include <thrust/device\_vector.h>  #include <thrust/generate.h>  #include <time.h>  #include <math.h>  #include <stdlib.h>  void vec\_add(thrust::device\_vector<float>& x, thrust::device\_vector<float>& y)  {  thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), thrust::plus<float>());  }  int main(int argc, char \*argv[])  {  srand(time(NULL));  int size = argc > 1 ? atoi(argv[1]) : 512;  thrust::host\_vector <float> v1(size), v2(size);  for (int i = 0; i < size; i++)  {  v1[i] = (float)(rand()) / RAND\_MAX;  v2[i] = (float)(rand()) / RAND\_MAX;  }  cudaEvent\_t start, stop;  float elapsed\_time;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start, 0);  thrust::device\_vector <float> v3 = v1, v4 = v2;  vec\_add(v3, v4);  cudaEventRecord(stop, 0);  cudaEventSynchronize(stop);  cudaEventElapsedTime(&elapsed\_time, start, stop);  printf("Elapsed time custom func: %f\n", elapsed\_time);  cudaEventDestroy(start);  cudaEventDestroy(stop);  return 0;  } |

## cublas.cu

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <cublas\_v2.h>  #include <math.h>  int main(int argc, char \*argv[])  {  int num\_elem = argc > 1 ? atoi(argv[1]) : 512;  size\_t size\_in\_bytes = (num\_elem \* sizeof(float));  float \*A\_dev, \*B\_dev, \*A\_h, \*B\_h;  cudaMalloc((void \*\*) &A\_dev, size\_in\_bytes);  cudaMalloc((void \*\*) &B\_dev, size\_in\_bytes);  cudaMallocHost((void \*\*) &A\_h, size\_in\_bytes);  cudaMallocHost((void \*\*) &B\_h, size\_in\_bytes);  memset(A\_h, 0, size\_in\_bytes);  memset(B\_h, 0, size\_in\_bytes);  cublasHandle\_t cublas\_handle;  cublasCreate(&cublas\_handle);  for (int i = 0; i < num\_elem; i++)  {  A\_h[i] = (float)(rand()) / RAND\_MAX;  B\_h[i] = (float)(rand()) / RAND\_MAX;  }  int num\_rows = num\_elem;  int num\_cols = 1;  size\_t elem\_size = sizeof(float);  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start, 0);  cublasSetMatrix(num\_rows, num\_cols, elem\_size, A\_h, num\_rows, A\_dev, num\_rows);  cublasSetMatrix(num\_rows, num\_cols, elem\_size, B\_h, num\_rows, B\_dev, num\_rows);  const int stride = 1;  float alpha = 1.0F;  cublasSaxpy(cublas\_handle, num\_elem, &alpha, A\_dev, stride, B\_dev, stride);  cublasGetMatrix(num\_rows, num\_cols, elem\_size, A\_dev, num\_rows, A\_h, num\_rows);  cublasGetMatrix(num\_rows, num\_cols, elem\_size, B\_dev, num\_rows, B\_h, num\_rows);  cudaStreamSynchronize(0);  cudaEventRecord(stop, 0);  cudaEventSynchronize(stop);  float elapsed\_time;  cudaEventElapsedTime(&elapsed\_time, start, stop);  printf("Elapsed time: %f\n", elapsed\_time);  cublasDestroy(cublas\_handle);  cudaFree(A\_dev);  cudaFree(B\_dev);  cudaFreeHost(A\_h);  cudaFreeHost(A\_h);  cudaFreeHost(B\_h);  cudaEventDestroy(start);  cudaEventDestroy(stop);  cudaDeviceReset();  return 0;  } |

# 

# Заключение

В ходе выполнения расчетно-графического задания был проведен сравнительный анализ производительности программ, реализующих алгоритмы скалярного перемножения векторов с использованием библиотек Thrust, cuBLAS, «сырого» CUDA C кода.

В результате проведенного анализа можно сделать выводы, что максимально быстродейственным к поставленной задаче оказался cuBLAS, а наименее быстродейственным - Thrust. CUDA C показывает также хорошие результаты, но при увеличении размера векторов ее скорость ощутимо падает, что делает ее хорошим выбором именно для маленького размера векторов (N < 512).