Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Факультет информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"

профиль "Программное обеспечение средств вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

**Расчётно-графическое задание по дисциплине  
 Программирование графических процессов**

Выполнил:

студент гр. ИП\_712 \_\_\_\_\_\_\_\_\_\_\_\_\_/Алексеев С.В./

ФИО студента

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Проверил

\_\_\_\_\_\_\_\_\_\_\_\_\_/Нужнов А. В./

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2020 г.

**Содержание**

[**Расчётно-графическое задание** 3](#_Toc40132736)

[**Теоретические сведения** 3](#_Toc40132737)

[**Кратко о GPU** 3](#_Toc40132737)

[**CUDA** 4](#_Toc40132737)

[**Thrust** 5](#_Toc40132737)

[**cuBLAS** 5](#_Toc40132737)

[**Сравнение производительности разных способов** 6](#_Toc40132738)

[**Листинг кода**](#_Toc40132739) 7

**Расчётно-графическое задание**

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

**Теоретические сведения**

**Кратко о GPU**

**Графический процессор** (англ. *graphics processing unit*, *GPU*) — отдельное устройство персонального компьютера или игровой приставки, выполняющее графический рендеринг; в начале 2000-х годов графические процессоры стали массово применяться и в других устройствах: планшетные компьютеры, встраиваемые системы, цифровые телевизоры.

Современные графические процессоры очень эффективно обрабатывают и отображают компьютерную графику, благодаря специализированной конвейерной архитектуре они намного эффективнее в обработке графической информации, чем типичный центральный процессор.

Графический процессор в современных видеокартах (видеоадаптерах) применяется в качестве ускорителя трёхмерной графики.

Может применяться как в составе дискретной видеокарты, так и в интегрированных решениях (встроенных в северный мост либо в гибридный процессор).

**Основные отличия GPU от CPU**

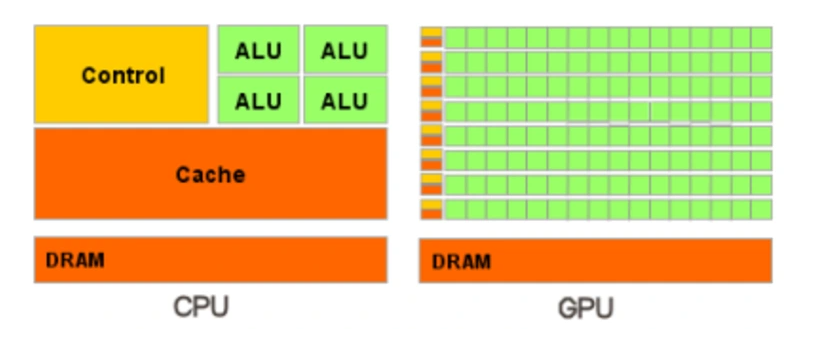


Рисунок 1. Сравнение устройства CPU и GPU

Архитектура GPU отличается от CPU большим количеством ядер и минималистичным набором команд, направленных в основном на векторные вычисления. На уровне архитектуры решены вопросы параллельной работы большого числа ядер и одновременного доступа к памяти. Современные GPU содержат от 2-х до 4-х тысяч шейдерных блоков, которые объединены в вычислительные юниты (Compute Unit). При параллельных вычислениях особенно остро стоит проблема одновременного доступа к памяти. Если каждый из потоковых процессоров попытается выполнить запись в ячейку памяти, то эти команды упрутся в блокировку и их необходимо будет поставить в очередь, что сильно снизит производительность. Поэтому потоковые процессоры выполняют команды небольшими группами: пока одна группа производит вычисления, другая загружает регистры и т.д. Также можно объединить ядра в рабочие группы, обладающие общей памятью и внутренними механизмами синхронизации.

Еще одной важной особенностью GPU является наличие векторных регистров и векторных АЛУ, которые могут выполнять операции одновременно для нескольких компонентов вектора. Это в первую очередь нужно для 3D графики, но поскольку наш мир трехмерный, ничто не мешает использовать это для многих физических вычислений. При наличии свободных векторных АЛУ их можно использовать и для вычисления скалярных величин.

**CUDA**

Технология CUDA (англ. Compute Unified Device Architecture) - программно-аппаратная архитектура, позволяющая производить вычисления с использованием графических процессоров NVIDIA, поддерживающих технологию GPGPU (произвольных вычислений на видеокартах). Архитектура CUDA впервые появились на рынке с выходом чипа NVIDIA восьмого поколения - G80 и присутствует во всех последующих сериях графических чипов, которые используются в семействах ускорителей GeForce, ION, Quadro и Tesla.

CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, выполнимые на графических процессорах NVIDIA и включать специальные функции в текст программы на Cи. CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью, организовывать на нём сложные параллельные вычисления.

**Thrust**

Thrust – это C++ библиотека шаблонов для CUDA основанная на STL. Thrust, с помощью высокоуровневого интерфейса, позволяет реализовывать высокоскоростные параллельные вычисления при минимальных трудозатратах на программирование.

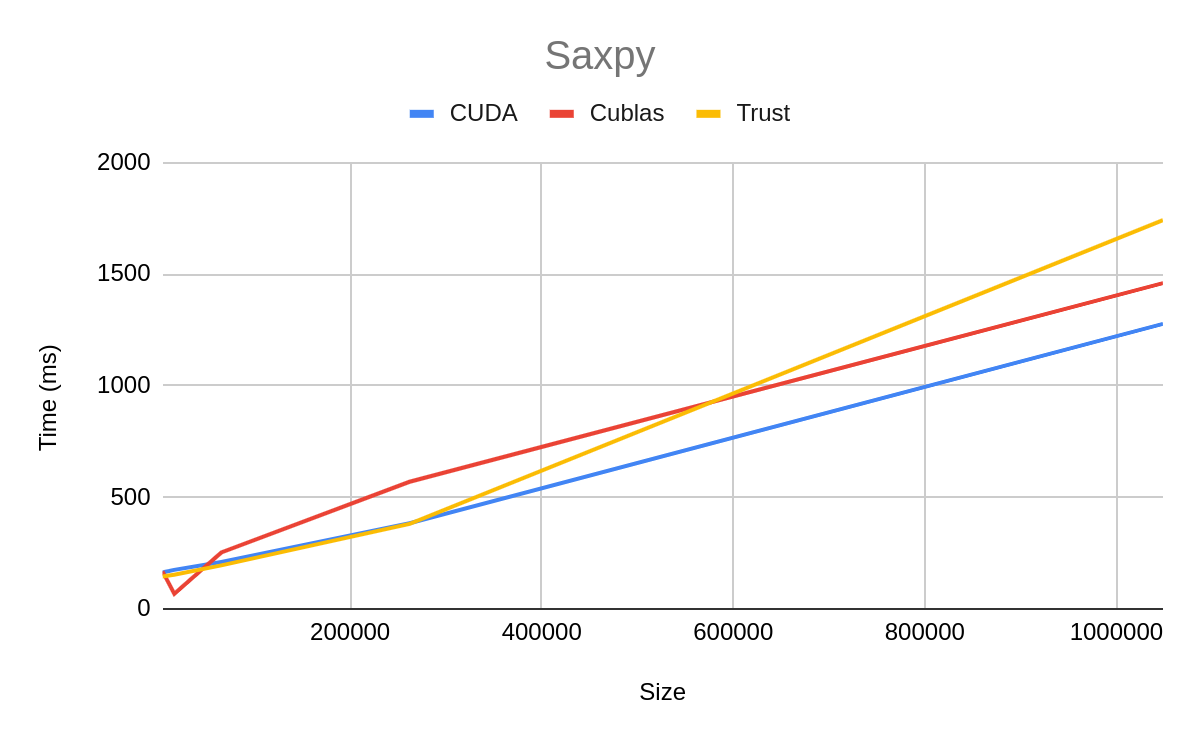
Thrust предоставляет богатый набор распараллеленых операций, таких как сортировка, суммирование и т. д., которые упрощают реализацию более сложных алгоритмов.

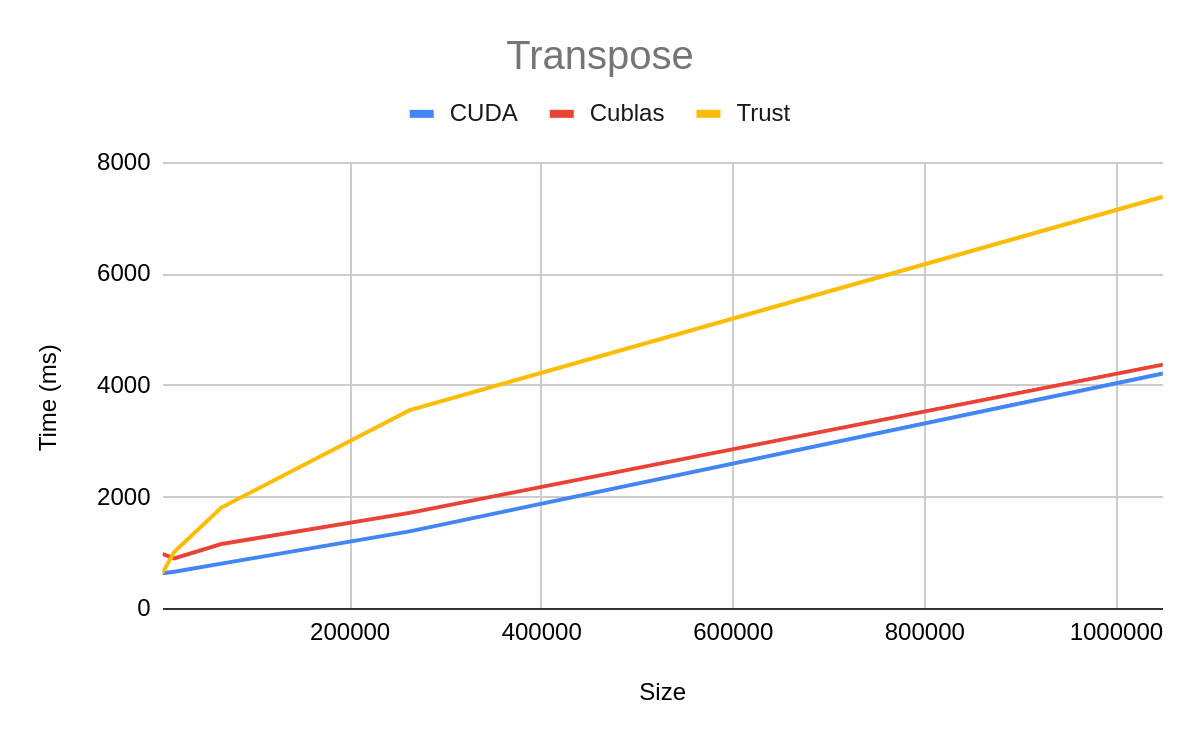
***cuBLAS***

cuBLAS – реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (Basic Linear Algebra Subprograms) для CUDA. Этот пакет позволяет получить доступ к вычислительным ресурсам графических процессоров NVIDIA. Перед тем, как начать изучение непосредственно библиотеки cuBLAS, стоит ознакомится с интерфейсом BLAS, который реализует данная библиотека.

Для работы с графическим процесоором использовался графический адаптер NVIDIA GEFORCE GTX 1050, находящийся в конфигурации класстера cyber.sibsutis.

**Сравнение произовдительности разных способов**





**Листинг кода**

**cuda\_sample:**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cstdio>

#include <ctime>

const size\_t N = 1 << 20;

const size\_t Nx = 1 << 10;

const size\_t Ny = 1 << 10;

\_\_global\_\_ void transpose(float \*matrixOrigin, float \*matrixRes)

{

size\_t x = blockIdx.x \* blockDim.x + threadIdx.x;

size\_t y = blockIdx.y \* blockDim.y + threadIdx.y;

size\_t width = gridDim.x \* blockDim.x;

matrixRes[x + y \* width] = matrixOrigin[y + x \* width];

}

\_\_global\_\_ void saxpy(float \*vectorA, float \*vectorB, float alpha)

{

size\_t index = blockIdx.x \* blockDim.x + threadIdx.x;

vectorA[index] = vectorA[index] \* alpha + vectorB[index];

}

int main()

{

clock\_t start;

start = clock();

cudaStream\_t stream0;

cudaStreamCreate(&stream0);

float \*matrix, \*matrix\_dev\_origin, \*matrix\_dev\_res;

cudaHostAlloc((void \*\*) &matrix, N \* sizeof(float), cudaHostAllocDefault);

for(int i = 0; i < N; ++i)

matrix[i] = i;

cudaMalloc((void \*\*) &matrix\_dev\_origin, sizeof(float) \* N);

cudaMalloc((void \*\*) &matrix\_dev\_res, sizeof(float) \* N);

cudaMemcpyAsync(matrix\_dev\_origin, matrix, sizeof(float) \* N, cudaMemcpyHostToDevice, stream0);

transpose <<< dim3(Nx / 32, Ny / 32), dim3(32, 32) >>> (matrix\_dev\_origin, matrix\_dev\_res);

cudaMemcpyAsync(matrix, matrix\_dev\_res, sizeof(float) \* N, cudaMemcpyDeviceToHost, stream0);

cudaStreamSynchronize(stream0);

cudaFree(matrix\_dev\_origin);

cudaFree(matrix\_dev\_res);

cudaFreeHost(matrix);

start = clock();

float \*vecA, \*vecB, \*vecA\_device, \*vecB\_device;

cudaStream\_t stream\_m0;

cudaStreamCreate(&stream\_m0);

cudaStream\_t stream1;

cudaStreamCreate(&stream1);

cudaHostAlloc((void \*\*) &vecA, N \* sizeof(float), cudaHostAllocDefault);

cudaHostAlloc((void \*\*) &vecB, N \* sizeof(float), cudaHostAllocDefault);

for (int i = 0; i < N; ++i)

{

vecA[i] = i;

vecB[i] = i \* 2 - 1;

}

cudaMalloc((void \*\*) &vecA\_device, sizeof(float) \* N);

cudaMalloc((void \*\*) &vecB\_device, sizeof(float) \* N);

cudaMemcpyAsync(vecA\_device, vecA, sizeof(int) \* N, cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(vecB\_device, vecB, sizeof(int) \* N, cudaMemcpyHostToDevice, stream1);

saxpy <<< N / 2 / 1024, 1024, 0, stream0 >>> (vecA\_device, vecB\_device, 2.25);

saxpy <<< N / 2 / 1024, 1024, 0, stream1 >>> (vecA\_device + N / 2, vecB\_device + N / 2, 2.25);

cudaMemcpyAsync(vecA, vecA\_device, sizeof(float) \* N / 2, cudaMemcpyDeviceToDevice, stream0);

cudaMemcpyAsync(vecA + N / 2, vecA\_device + N / 2, sizeof(float) \* N / 2, cudaMemcpyDeviceToDevice, stream1);

cudaStreamSynchronize(stream0);

cudaStreamSynchronize(stream1);

printf("SAXPY time (s) - %f\n", double(clock() - start) / CLOCKS\_PER\_SEC);

cudaFree(vecA\_device);

cudaFree(vecB\_device);

cudaFreeHost(vecA);

cudaFreeHost(vecB);

}

**trust\_sample:**

**#include <thrust/host\_vector.h>**

**#include <thrust/device\_vector.h>**

**#include <ctime>**

**struct fun**

**{**

**float alpha;**

**fun(float \_alpha): alpha(\_alpha)**

**{}**

**\_\_host\_\_ \_\_device\_\_**

**float operator()(float x, float y)**

**{**

**return x \* alpha + y;**

**}**

**};**

**int main()**

**{**

**size\_t N = 1 << 20;**

**size\_t Nx = 1 << 10;**

**size\_t Ny = 1 << 10;**

**clock\_t start = clock();**

**thrust::device\_vector<float> matrix\_origin\_dev(N);**

**thrust::device\_vector<float> matrix\_res\_dev(N);**

**thrust::sequence(matrix\_origin\_dev.begin(), matrix\_origin\_dev.end());**

**thrust::counting\_iterator<size\_t> indices(0);**

**thrust::device\_vector<float> temp(Ny);**

**for (size\_t i = 0; i < Nx; ++i)**

**{**

**thrust::sequence(temp.begin(), temp.end(), 0 + i, Nx);**

**thrust::copy(thrust::make\_permutation\_iterator(matrix\_origin\_dev.begin(), temp.begin()),thrust::make\_permutation\_iterator(matrix\_origin\_dev.begin(), temp.end()), matrix\_res\_dev.begin() + i \* Ny);**

**}**

**thrust::host\_vector<float> host\_result = matrix\_res\_dev;**

**printf("Transpose time (s) - %f\n", double(clock() - start) / CLOCKS\_PER\_SEC);**

**start = clock();**

**fun alpha(2.25);**

**thrust::device\_vector<float> vectorA(N);**

**thrust::device\_vector<float> vectorB(N);**

**thrust::sequence(vectorA.begin(), vectorA.end());**

**thrust::sequence(vectorB.begin(), vectorB.end(), -1, 2);**

**thrust::transform(vectorA.begin(), vectorA.end(), vectorB.begin(), vectorA.begin(), alpha);**

**thrust::host\_vector<float> resVector = vectorA;**

**printf("SAXPY time (s) - %f\n", double(clock() - start) / CLOCKS\_PER\_SEC);**

**}**

**cuBLAS\_sample:**

**#include <cublas\_v2.h>**

**#include <cstdio>**

**#include <iostream>**

**#include <ctime>**

**#include <cstddef>**

**int main()**

**{**

**size\_t Nx = 1 << 10;**

**size\_t Ny = 1 << 10;**

**size\_t N = 1 << 20;**

**clock\_t start = clock();**

**cublasHandle\_t handle;**

**cublasCreate(&handle);**

**float\* matrix;**

**cudaMallocHost((void\*\*)&matrix, N \* sizeof(float));**

**for (int i = 0; i < N; ++i)**

**matrix[i] = (float)i;**

**float\* matrix\_in\_dev;**

**cudaMalloc((void\*\*)&matrix\_in\_dev, N \* sizeof(float));**

**float\* matrix\_out\_dev;**

**cudaMalloc((void\*\*)&matrix\_out\_dev, N \* sizeof(float));**

**cublasSetMatrix(Ny, Nx, sizeof(float), matrix, Ny, matrix\_in\_dev, Ny);**

**float alpha = 1.;**

**float beta = 0.;**

**cublasSgeam(handle, CUBLAS\_OP\_T, CUBLAS\_OP\_T, Nx, Ny, &alpha, matrix\_in\_dev, Ny, &beta, matrix\_in\_dev, Ny, matrix\_out\_dev, Nx);**

**cublasGetMatrix(Ny, Nx, sizeof(float), matrix\_out\_dev, Ny, matrix, Ny);**

**cudaStreamSynchronize(NULL);**

**cudaFreeHost(matrix);**

**cudaFree(matrix\_in\_dev);**

**cudaFree(matrix\_out\_dev);**

**cublasDestroy(handle);**

**printf("%ld\n", (clock() - start));**

**start = clock();**

**cublasHandle\_t handle1;**

**cublasCreate(&handle1);**

**float\* vecA;**

**cudaMallocHost((void\*\*)&vecA, N \* sizeof(float));**

**float\* vecB;**

**cudaMallocHost((void\*\*)&vecB, N \* sizeof(float));**

**for (int i = 0; i < N; ++i)**

**{**

**vecA[i] = (float)i;**

**vecB[i] = (float)(i \* 2 - 1);**

**}**

**float\* vec\_A\_dev;**

**cudaMalloc((void\*\*)&vec\_A\_dev, N \* sizeof(float));**

**float\* vec\_B\_dev;**

**cudaMalloc((void\*\*)&vec\_B\_dev, N \* sizeof(float));**

**cublasSetMatrix(N, 1, sizeof(float), vecA, N, vec\_A\_dev, N);**

**cublasSetMatrix(N, 1, sizeof(float), vecB, N, vec\_B\_dev, N);**

**float alpha1 = 2.25;**

**cublasSaxpy(handle1, N, &alpha1, vec\_A\_dev, 1, vec\_B\_dev, 1);**

**cublasGetMatrix(N, 1, sizeof(float), vec\_B\_dev, N, vecB, N);**

**cudaStreamSynchronize(NULL);**

**cublasDestroy(handle1);**

**cudaFreeHost(vecA);**

**cudaFreeHost(vecB);**

**cudaFree(vec\_A\_dev);**

**cudaFree(vec\_B\_dev);**

**printf("%d", (clock() - start));**

**}**