Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра ПМиК

Расчетно-графическое задание

по дисциплине

«Технология программирования графических ускорителей»

Выполнил:

студент гр. МГ-211 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Бурдуковский И.А./

подпись

Проверил:

Профессор

кафедры ПМиК \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Малков Е. А./

ОЦЕНКА, подпись

Новосибирск 2023

**Оглавление**

[Оглавление 2](#_Toc154508005)

[Задание 3](#_Toc154508006)

[Теоретическая часть 4](#_Toc154508007)

[Технология CUDA 4](#_Toc154508008)

[Thrust 5](#_Toc154508009)

[cuBLAS 5](#_Toc154508010)

[Результат работы программы 6](#_Toc154508011)

[Сравнение производительности алгоритмов 7](#_Toc154508012)

[Листинг 9](#_Toc154508013)

# Задание

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

За основу алгоритма линейной алгебры был взят алгоритм SAXPY

# Теоретическая часть

## Технология CUDA

Это программно-аппаратная вычислительная архитектура Nvidia, основанная на расширении языка Си, которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. CUDA помогает реализовывать алгоритмы, выполнимые на графических процессорах видеоускорителей Geforce восьмого поколения и старше (серии Geforce 8, Geforce 9, Geforce 200), а также Quadro и Tesla.

CUDA использует параллельную модель вычислений, когда каждый из SIMD процессоров выполняет ту же инструкцию над разными элементами данных параллельно. GPU является вычислительным устройством, сопроцессором (device) для центрального процессора (host), обладающим собственной памятью и обрабатывающим параллельно большое количество потоков. Ядром (kernel) называется функция для GPU, исполняемая потоками (аналогия из 3D графики — шейдер).

Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (thread blocks). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме, и может состоять из 512 потоков на текущем аппаратном обеспечении.

Блоки потоков выполняются в виде небольших групп, называемых варп (warp), размер которых — 32 потока. Это минимальный объём данных, которые могут обрабатываться в мультипроцессорах. И так как это не всегда удобно, CUDA позволяет работать и с блоками, содержащими от 64 до 512 потоков.

Группировка блоков в сетки позволяет уйти от ограничений и применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае, блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных.

Модель памяти в CUDA отличается возможностью побайтной адресации, поддержкой как gather, так и scatter. Доступно довольно большое количество регистров на каждый потоковый процессор, до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые или числа с плавающей точкой.

СUDA имеет несколько типов памяти. Эта технология предполагает специальный подход к разработке, не совсем такой, как принят в программах для CPU. Нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируется и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах.

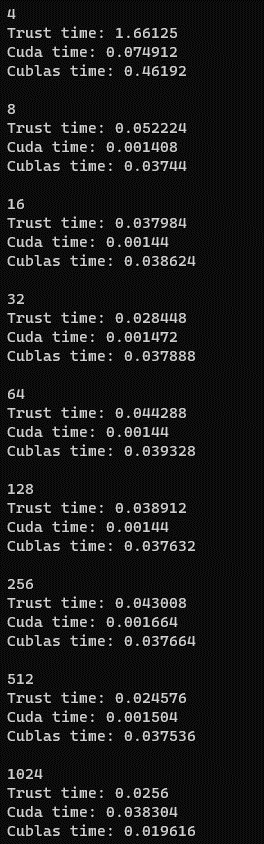
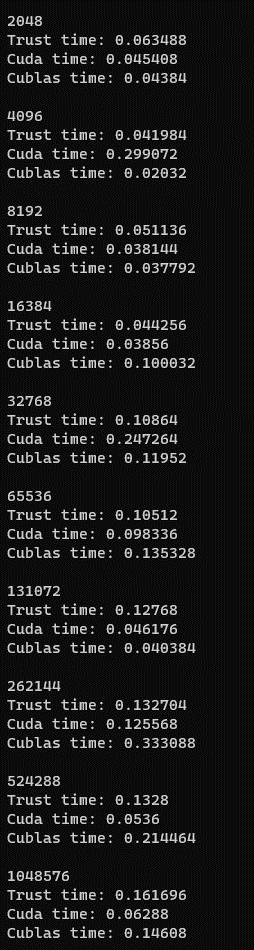
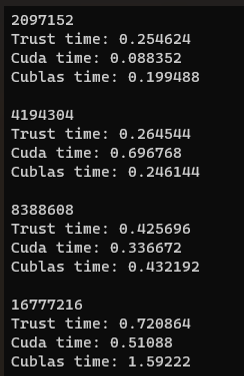
## Thrust

Thrust - это библиотека параллельных алгоритмов, напоминающая библиотеку стандартных шаблонов (STL). Интерфейс высокого уровня Thrust значительно повышает производительность программистов, обеспечивая при этом переносимость производительности между графическими процессорами и многоядерными процессорами. Совместимость с установленными технологиями (такими как CUDA, TBB и OpenMP) облегчает интеграцию с существующим программным обеспечением.

## cuBLAS

Библиотека cuBLAS обеспечивает реализацию с ускорением на GPU основных подпрограмм линейной алгебры (BLAS). cuBLAS ускоряет приложения AI и HPC с помощью стандартных отраслевых API-интерфейсов BLAS, оптимизированных для графических процессоров NVIDIA. Библиотека cuBLAS содержит расширения для пакетных операций, выполнения на нескольких графических процессорах, а также выполнения смешанной и низкой точности. Используя cuBLAS, приложения автоматически получают выгоду от регулярных улучшений производительности и новых архитектур GPU. Библиотека cuBLAS включена как в [NVIDIA HPC SDK, так](https://developer.nvidia.com/hpc-sdk) и в [CUDA Toolkit](https://developer.nvidia.com/cuda-downloads).

# Результат работы программы

# Сравнение производительности алгоритмов

Были собраны данные по времени выполнения алгоритмов «Сырой» CUDA C, cuBLAS, Thrust. Ниже представлен график на основе полученных данных.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| степень | размерность массива | thrust | cuda | cublas |
| 2 | 4 | 1,661250 | 0,074912 | 0,461920 |
| 3 | 8 | 0,052224 | 0,001408 | 0,037440 |
| 4 | 16 | 0,037984 | 0,014400 | 0,038624 |
| 5 | 32 | 0,028448 | 0,001472 | 0,037888 |
| 6 | 64 | 0,044288 | 0,001440 | 0,039328 |
| 7 | 128 | 0,038912 | 0,001440 | 0,037632 |
| 8 | 256 | 0,043008 | 0,001664 | 0,037664 |
| 9 | 512 | 0,024576 | 0,001504 | 0,037536 |
| 10 | 1024 | 0,025600 | 0,038304 | 0,019616 |
| 11 | 2048 | 0,063488 | 0,045408 | 0,043840 |
| 12 | 4096 | 0,041984 | 0,299072 | 0,020320 |
| 13 | 8192 | 0,051136 | 0,038144 | 0,037792 |
| 14 | 16384 | 0,044256 | 0,038560 | 0,100032 |
| 15 | 32768 | 0,108640 | 0,247264 | 0,119520 |
| 16 | 65536 | 0,105120 | 0,098336 | 0,135328 |
| 17 | 131072 | 0,127680 | 0,041760 | 0,040384 |
| 18 | 262144 | 0,132704 | 0,125568 | 0,333088 |
| 19 | 524288 | 0,132800 | 0,053600 | 0,214464 |
| 20 | 1048576 | 0,161696 | 0,062880 | 0,146080 |
| 21 | 2097152 | 0,254624 | 0,088352 | 0,199488 |
| 22 | 4194304 | 0,264544 | 0,696768 | 0,246144 |
| 23 | 8388608 | 0,425696 | 0,336672 | 0,432192 |



На графике видно, что по средней скорости выполнения лидирует «сырой» CUDA, но Thrust показал более стабильную скорость работы при увеличении объёма обрабатываемых данных, не смотря на общую медлительность алгоритма по сравнению с остальными.

# Листинг

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

#include <thrust/transform.h>

#include <thrust/fill.h>

#include <thrust/sequence.h>

#include <cublas\_v2.h>

#pragma comment (lib, "cublas.lib")

#include <cublas\_v2.h>

#pragma comment (lib, "cufft.lib")

#include <cufft.h>

#include <cstdlib>

using namespace std;

struct saxpy\_functor

{

const float a;

saxpy\_functor(float \_a) : a(\_a) {}

\_\_host\_\_ \_\_device\_\_ float operator()(float x, float y) {

return a \* x + y;

}

};

void saxpy(float a, thrust::device\_vector<float>& x, thrust::device\_vector<float>& y){

saxpy\_functor func(a);

thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), func);

}

\_\_global\_\_ void cuda\_saxpy(float \*a, float \*b, float alpha){

int j = threadIdx.x + blockIdx.x \* blockDim.x;

a[j] = j;

b[j] = 0.87;

a[j] = alpha \* a[j] + b[j];

}

\_\_host\_\_ void print\_array(float \*data1, float \*data2, int num\_elem, const char \*prefix) {

printf("\n%s", prefix);

for (int i = 0; i < num\_elem; i++)

printf("\n%2d: %2.4f %2.4f", i + 1, data1[i], data2[i]);

}

void CudaTest(int shift){

printf("%d\n", 1 << shift);

float elapsedTime;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Trust

thrust::host\_vector<float> h1(1 << shift);

thrust::host\_vector<float> h2(1 << shift);

thrust::sequence(h1.begin(), h1.end());

thrust::fill(h2.begin(), h2.end(), 0.87);

thrust::device\_vector<float> d1 = h1;

thrust::device\_vector<float> d2 = h2;

cudaEventRecord(start, 0);

saxpy(3.0, d1, d2);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

printf("Trust time: %g\n", elapsedTime);

h2 = d2;

h1 = d1;

// Raw Cuda

float \*h, \*da, \*db, alpha = 3.0F;

int threads\_per\_block = 525, N = 1 << shift;

int num\_of\_blocks = N / threads\_per\_block;

h = (float\*)calloc(N, sizeof(float));

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

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

cudaEventRecord(start, 0);

cuda\_saxpy << <dim3(num\_of\_blocks), dim3(threads\_per\_block) >> > (da, db, alpha);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

printf("Cuda time: %g\n", elapsedTime);

cudaMemcpy(h, da, N \* sizeof(float), cudaMemcpyDeviceToHost);

// Cublas

const int num\_elem = 1 << shift;

const size\_t size\_in\_bytes = (num\_elem \* sizeof(float));

float \*ha, \*hb;

cudaMalloc((void\*\*)&da, size\_in\_bytes);

cudaMalloc((void\*\*)&db, size\_in\_bytes);

cudaMallocHost((void\*\*)&ha, size\_in\_bytes);

cudaMallocHost((void\*\*)&hb, size\_in\_bytes);

memset(ha, 0, size\_in\_bytes);

memset(hb, 0, size\_in\_bytes);

cublasHandle\_t cublas\_handle;

cublasCreate(&cublas\_handle);

for (int i = 0; i < num\_elem; i++) {

ha[i] = (float)i;

hb[i] = 0.87;

}

const int num\_rows = num\_elem;

const int num\_cols = 1;

const size\_t elem\_size = sizeof(float);

cublasSetMatrix(num\_rows, num\_cols, elem\_size, ha, num\_rows, da, num\_rows);

cublasSetMatrix(num\_rows, num\_cols, elem\_size, hb, num\_rows, db, num\_rows);

const int stride = 1;

alpha = 3.0F;

cudaEventRecord(start, 0);

cublasSaxpy(cublas\_handle, num\_elem, &alpha, da, stride, db, stride);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cublasGetMatrix(num\_rows, num\_cols, elem\_size, da, num\_rows, ha, num\_rows);

cublasGetMatrix(num\_rows, num\_cols, elem\_size, db, num\_rows, hb, num\_rows);

const int default\_stream = 0;

cudaStreamSynchronize(default\_stream);

cudaEventElapsedTime(&elapsedTime, start, stop);

printf("Cublas time: %g\n", elapsedTime);

printf("\n");

cublasDestroy(cublas\_handle);

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaFreeHost(ha);

cudaFreeHost(hb);

cudaFree(da);

cudaFree(db);

free(h);

}

int main() {

int shift = 2;

for(;shift <= 24; shift++){

CudaTest(shift);

}

system("pause");

return 0;

}