Федеральное агентство связи (Россвязь)

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики (ПМиК)

Отчет по лабораторной работе №6

по дисциплине «Программирование Графических Процессоров»

Выполнил: студент группы ИП-712

Алексеев С.В.

Проверил:

Нужнов А.В.

Новосибирск – 2020 г.

**Содержание**

Оглавление

[Задание 3](#_Toc38100674)

[Требуется: 3](#_Toc38100675)

[Цель: 3](#_Toc38100676)

[решение 4](#_Toc38100677)

[Выделение памяти 4](#_Toc38100678)

[Копирование памяти на устройство 4](#_Toc38100679)

[Выполнение кода на устрйстве 5](#_Toc38100680)

[Копирование памяти на хост 5](#_Toc38100681)

[Сравнение скорости двух методов 5](#_Toc38100682)

[Определение оптимального размера порций данных 5](#_Toc38100683)

[При сложении векторов 6](#_Toc38100684)

[При умножении векторов 7](#_Toc38100685)

[Заключение 7](#_Toc38100686)

[**Приложение** 7](#_Toc38100687)

# Задание

## Требуется:

разработать и программно реализовать алгоритм для сравнения производительности копирования устройство->хост (и наоборот) данных, размещенных в памяти выделенной на хосте обычным образом и с использованием закрепленных страниц;

подобрать оптимальный размер порции данных для реализации сложения векторов с использованием потоков CUDA для распараллеливания копирования и выполнения;

то же для реализации скалярного умножения.

## Цель:

Изучить преимущества использования потоков CUDA.

# решение

## Выделение памяти

Напишем функцию сложения векторов, который будет принимать на вход параметр use\_pages. Если данный флаг равен true, то метод будет использовать выделение памяти с использованием закрепленных страниц. Иначе – обычным способом.

float run(int FULL\_DATA\_SIZE, bool use\_pages)

#### Выделение памяти на хосте

if (use\_pages)

{

cudaHostAlloc((void\*\*)&host\_a, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

cudaHostAlloc((void\*\*)&host\_b, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

cudaHostAlloc((void\*\*)&host\_c, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

}

else

{

host\_a = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

host\_b = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

host\_c = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

}

### Копирование памяти на устройство

if (use\_pages)

{

cudaMemcpyAsync(dev\_a0, host\_a+i, chunk\_size\*sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_a1, host\_a+i+chunk\_size, chunk\_size\*sizeof(int), cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(dev\_b0, host\_b+i, chunk\_size\*sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_b1, host\_b+i+chunk\_size, chunk\_size\*sizeof(int), cudaMemcpyHostToDevice, stream1);

}

else

{

cudaMemcpy(dev\_a0, host\_a + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_a1, host\_a + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b0, host\_b + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b1, host\_b + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

}

### Выполнение кода на устрйстве

kernel << <chunk\_size / 256, 256, 0, stream0 >> >(dev\_a0, dev\_b0, dev\_c0);

kernel << <chunk\_size / 256, 256, 0, stream1 >> >(dev\_a1, dev\_b1, dev\_c1);

### Копирование памяти на хост

if (use\_pages)

{

cudaMemcpyAsync(host\_c+i, dev\_c0, chunk\_size\*sizeof(int), cudaMemcpyDeviceToHost, stream0);

cudaMemcpyAsync(host\_c+i+chunk\_size, dev\_c1, chunk\_size\*sizeof(int), cudaMemcpyDeviceToHost, stream1);

}

else

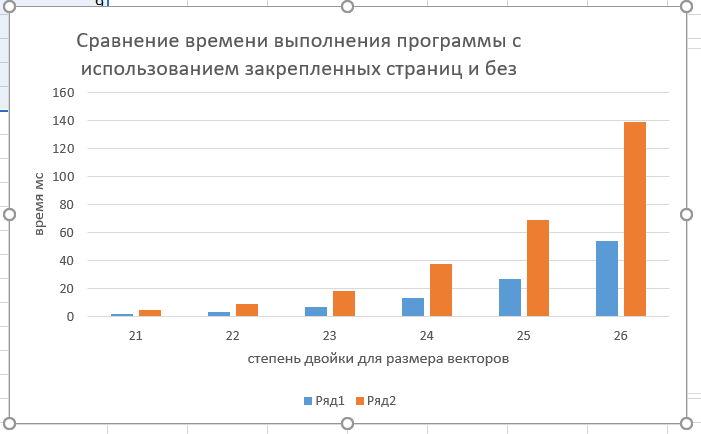
{

cudaMemcpy(host\_c + i, dev\_c0, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(host\_c + i + chunk\_size, dev\_c1, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost);

}

### Сравнение скорости двух методов



## Определение оптимального размера порций данных

Для вычисления оптимальной величины чанков, напишем следующую функцию:

float run(int chunk\_size, int FULL\_DATA\_SIZE, bool use\_pages, bool check\_result)

Взяв вектора размером 226, мы перебрали различные варианты размеров чанков:

for (int i = 0; i < FULL\_DATA\_SIZE; i += chunk\_size \* 2)

{

cudaMemcpyAsync(dev\_a0, host\_a + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_a1, host\_a + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(dev\_b0, host\_b + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_b1, host\_b + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream1);

kernel << <chunk\_size / 256, 256, 0, stream0 >> >(dev\_a0, dev\_b0, dev\_c0);

kernel << <chunk\_size / 256, 256, 0, stream1 >> >(dev\_a1, dev\_b1, dev\_c1);

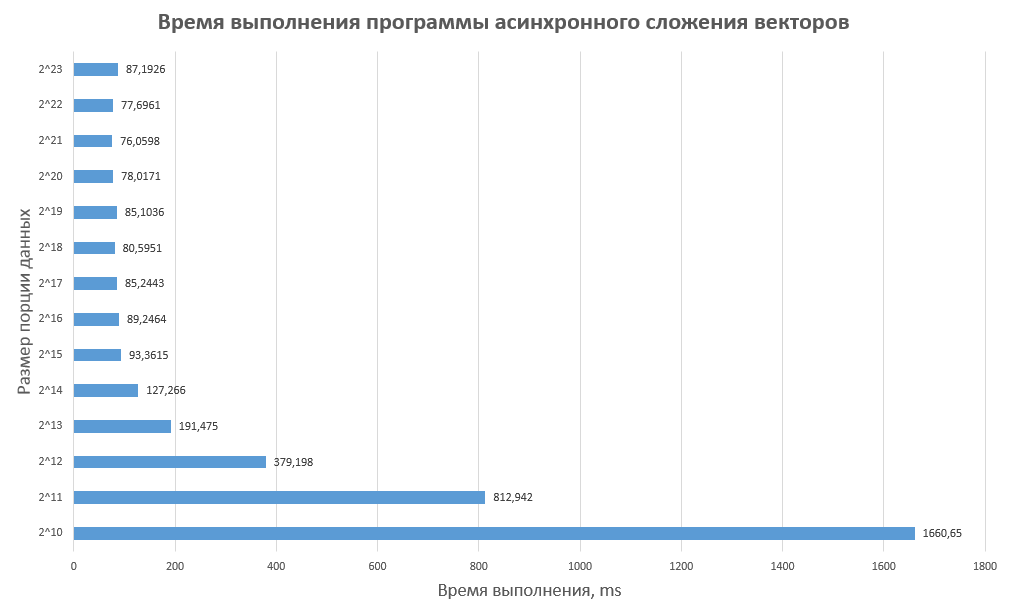
cudaMemcpyAsync(host\_c + i, dev\_c0, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost, stream0);

cudaMemcpyAsync(host\_c + i + chunk\_size, dev\_c1, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost, stream1);

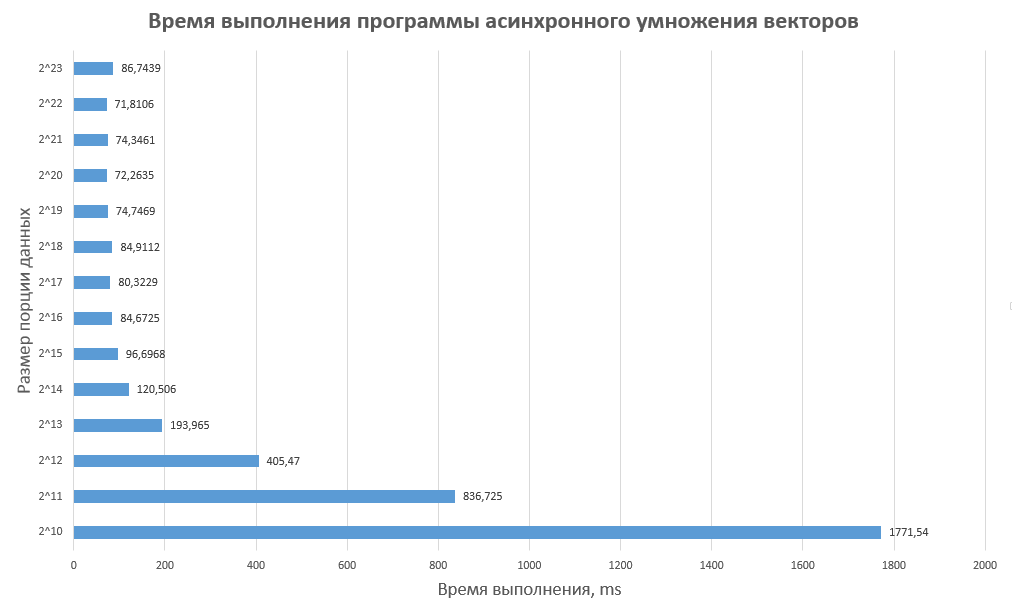
}

Результаты показали, что оптимальным размер колеблется в районе 218-222. Полученные результаты:

### При сложении векторов



### При умножении векторов



## Заключение

В ходе лабораторной работы разработан и реализован алгоритм для сравнения производительности копирования устройство->хост (и наоборот) данных, размещенных в памяти выделенной на хосте обычным образом и с использованием закрепленных страниц.

Также подобран оптимальный размер порции данных для реализации сложения и умножения векторов с использованием потоков CUDA для распараллеливания копирования и выполнения

# **Приложение**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <iostream>

using namespace std;

#define SIZE 1024

\_\_global\_\_ void kernel(int\* a, int\* b, int\* c)

{

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

c[idx] = (2 \* 1024 \* a[idx]) / 1024 \* b[idx];

}

float run(int chunk\_size, int FULL\_DATA\_SIZE, bool use\_pages, bool check\_result)

{

cout << endl << endl;

cout << "Running on states:" << endl;

cout << "CHUNK\_SIZE : " << chunk\_size << endl;

cout << "FULL\_DATA\_SIZE : " << FULL\_DATA\_SIZE << "; k = " << FULL\_DATA\_SIZE / chunk\_size \* 1.0f << endl;

cout << "USE\_PAGES : " << use\_pages << endl;

cout << "CHECK\_RESULT : " << check\_result << endl;

cudaDeviceProp prop;

int whichDevice;

cudaGetDevice(&whichDevice);

cudaGetDeviceProperties(&prop, whichDevice);

if (!prop.deviceOverlap)

{

printf("Device does not support overlapping\n");

return 0;

}

cudaEvent\_t start, stop;

float elapsedTime;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaStream\_t stream0;

cudaStream\_t stream1;

cudaStreamCreate(&stream0);

cudaStreamCreate(&stream1);

int \*host\_a, \*host\_b, \*host\_c;

int \*dev\_a0, \*dev\_b0, \*dev\_c0;

int \*dev\_a1, \*dev\_b1, \*dev\_c1;

cudaMalloc((void\*\*)&dev\_a0, chunk\_size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_b0, chunk\_size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_c0, chunk\_size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_a1, chunk\_size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_b1, chunk\_size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_c1, chunk\_size \* sizeof(int));

if (use\_pages)

{

cudaHostAlloc((void\*\*)&host\_a, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

cudaHostAlloc((void\*\*)&host\_b, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

cudaHostAlloc((void\*\*)&host\_c, FULL\_DATA\_SIZE \* sizeof(int), cudaHostAllocDefault);

}

else

{

host\_a = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

host\_b = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

host\_c = (int\*)calloc(FULL\_DATA\_SIZE, sizeof(int));

}

for (int i = 0; i < FULL\_DATA\_SIZE; i++)

{

host\_a[i] = 1;

host\_b[i] = 1;

}

cudaEventRecord(start, 0);

for (int i = 0; i < FULL\_DATA\_SIZE; i += chunk\_size \* 2)

{

//cout << " CYCLE : " << i << endl;

if (use\_pages)

{

cudaMemcpyAsync(dev\_a0, host\_a + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_a1, host\_a + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(dev\_b0, host\_b + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_b1, host\_b + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice, stream1);

kernel << <chunk\_size / 256, 256, 0, stream0 >> >(dev\_a0, dev\_b0, dev\_c0);

kernel << <chunk\_size / 256, 256, 0, stream1 >> >(dev\_a1, dev\_b1, dev\_c1);

cudaMemcpyAsync(host\_c + i, dev\_c0, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost, stream0);

cudaMemcpyAsync(host\_c + i + chunk\_size, dev\_c1, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost, stream1);

}

else

{

cudaMemcpy(dev\_a0, host\_a + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_a1, host\_a + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b0, host\_b + i, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b1, host\_b + i + chunk\_size, chunk\_size \* sizeof(int), cudaMemcpyHostToDevice);

kernel << <chunk\_size / 256, 256, 0 >> >(dev\_a0, dev\_b0, dev\_c0);

kernel << <chunk\_size / 256, 256, 0 >> >(dev\_a1, dev\_b1, dev\_c1);

cudaMemcpy(host\_c + i, dev\_c0, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(host\_c + i + chunk\_size, dev\_c1, chunk\_size \* sizeof(int), cudaMemcpyDeviceToHost);

}

}

cudaStreamSynchronize(stream0);

cudaStreamSynchronize(stream1);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

cout << "ELAPSED\_TIME : " << elapsedTime << "ms" << endl;

if (check\_result)

{

int sum = 0;

for (int i = 0; i < FULL\_DATA\_SIZE; i++) {

sum += host\_c[i];

}

cout << "CHECKING\_RESULT: " << sum / (FULL\_DATA\_SIZE \* 2.0f)

<< ((sum == FULL\_DATA\_SIZE \* 2) ? " = SUCCESS" : " = FAIL") << endl;

}

cudaFreeHost(host\_a);

cudaFreeHost(host\_b);

cudaFreeHost(host\_c);

cudaFree(dev\_a0);

cudaFree(dev\_b0);

cudaFree(dev\_c0);

cudaFree(dev\_a1);

cudaFree(dev\_b1);

cudaFree(dev\_c1);

cudaStreamDestroy(stream0);

cudaStreamDestroy(stream1);

return elapsedTime;

}

int main()

{

int FULL\_DATA\_SIZE = SIZE \* SIZE \* 64;

for (int n = SIZE; n < FULL\_DATA\_SIZE ; n \*= 2)

{

run(n, FULL\_DATA\_SIZE, true, true);

}

//for (int i = 2; i <= 64; i\*=2) {

// int N = SIZE \* SIZE;

// int FULL\_DATA\_SIZE = SIZE \* SIZE \* i;

// run(N, FULL\_DATA\_SIZE, true, true);

// run(N, FULL\_DATA\_SIZE, false, true);

//}

return 0;

}