**Выполнила: Белоусова Е., ИП-911**

**Задача**

• 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

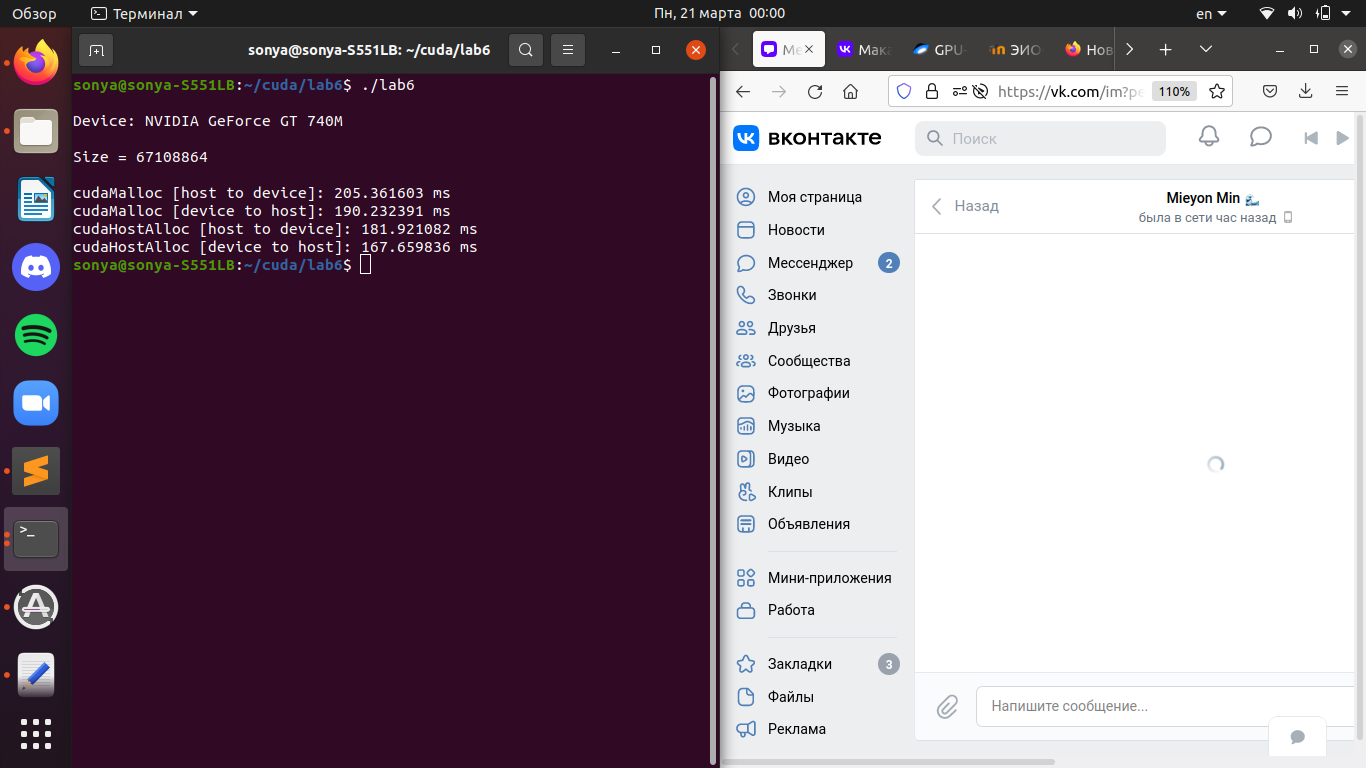
**Описание работы программы**

Для выделения памяти на хосте обычным образом будем использовать malloc, для выделения памяти с использованием закрепленных страниц будем использовать cudaHostAlloc.

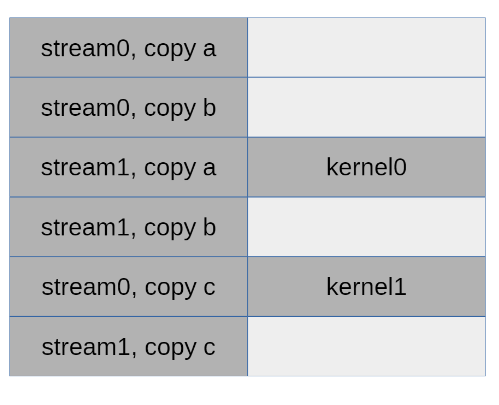
Из документации:

cudaHostAlloc выделяет размер байт памяти хоста, которая заблокирована по страницам и доступна для устройства. Драйвер отслеживает диапазоны виртуальной памяти, выделенные с помощью этой функции, и автоматически ускоряет вызовы таких функций, как cudaMemcpy(). Поскольку доступ к памяти может осуществляться непосредственно устройством, ее можно считывать или записывать с гораздо большей пропускной способностью, чем доступную для страниц память, полученную с помощью таких функций, как malloc(). Выделение чрезмерных объемов закрепленной памяти может снизить производительность системы, поскольку это уменьшает объем памяти, доступной системе для подкачки. В результате эту функцию лучше всего использовать экономно для выделения промежуточных областей для обмена данными между хостом и устройством.

Результаты:



С помощью потоков cuda реализуем сложение и умножение векторов. В одном потоке будем вызывать cudaMemcpyAsync для порции данных, вызывать ядро и потом копировать результат так же асинхронно. Внутри потока все команды выполняются синхронно, но сами потоки между собой асинхронные. Будем создавать два потока, в которых будем делить данные на пакеты, а пакеты будем делить на две части между потоками. Потоки перемежаемые. Возможно выполнение разных команд параллельно.



**Листинг**

//lab6.cu

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#include <cuda.h>

#include <cuda\_runtime.h>

#define CUDA\_CHECK\_RETURN(value) {\

cudaError\_t \_m\_cudaStat = value;\

if (\_m\_cudaStat != cudaSuccess) {\

fprintf(stderr, "Error %s at line %d in file %s\n",\

cudaGetErrorString(\_m\_cudaStat),\_\_LINE\_\_,\_\_FILE\_\_);\

exit(1);\

}}

void allocTest(int size, bool hostToDevice, float &elapsed\_time)

{

cudaEvent\_t start, stop;

int \*a, \*dev\_a;

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

CUDA\_CHECK\_RETURN(cudaHostAlloc((void\*\*)&a, size \* sizeof(\*a), cudaHostAllocDefault));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&dev\_a, size \* sizeof(\*dev\_a)));

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

a[i] = i;

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

if (hostToDevice == true) {

CUDA\_CHECK\_RETURN(cudaMemcpy(dev\_a, a, size \* sizeof(\*a), cudaMemcpyHostToDevice));

}

else {

CUDA\_CHECK\_RETURN(cudaMemcpy(a, dev\_a, size \* sizeof(\*dev\_a), cudaMemcpyDeviceToHost));

}

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&elapsed\_time, start, stop));

CUDA\_CHECK\_RETURN(cudaFreeHost(a));

CUDA\_CHECK\_RETURN(cudaFree(dev\_a));

CUDA\_CHECK\_RETURN(cudaEventDestroy(start));

CUDA\_CHECK\_RETURN(cudaEventDestroy(stop));

}

void mallocTest(int size, bool hostToDevice, float& elapsed\_time)

{

cudaEvent\_t start, stop;

int \*a, \*dev\_a;

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

a = (int\*)malloc(size \* sizeof(int));

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

a[i] = i;

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&dev\_a, size \* sizeof(\*dev\_a)));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

if (hostToDevice == true) {

CUDA\_CHECK\_RETURN(cudaMemcpy(dev\_a, a, size \* sizeof(\*a), cudaMemcpyHostToDevice));

}

else {

CUDA\_CHECK\_RETURN(cudaMemcpy(a, dev\_a, size \* sizeof(\*dev\_a), cudaMemcpyDeviceToHost));

}

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&elapsed\_time, start, stop));

free(a);

CUDA\_CHECK\_RETURN(cudaFree(dev\_a));

CUDA\_CHECK\_RETURN(cudaEventDestroy(start));

CUDA\_CHECK\_RETURN(cudaEventDestroy(stop));

}

int main(int argc, char const \*argv[])

{

cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, 0);

printf("\nDevice:\t%s\n\n", deviceProp.name);

int size = 0;

if (argc == 2) {

size = 1 << atoi(argv[1]);

} else {

size = 1 << 13;

}

size \*= size;

float elapsed\_time;

printf("Size = %d\n\n", size);

mallocTest(size, true, elapsed\_time);

printf("cudaMalloc [host to device]: %.6f ms\n", elapsed\_time);

mallocTest(size, false, elapsed\_time);

printf("cudaMalloc [device to host]: %.6f ms\n", elapsed\_time);

allocTest(size, true, elapsed\_time);

printf("cudaHostAlloc [host to device]: %.6f ms\n", elapsed\_time);

allocTest(size, false, elapsed\_time);

printf("cudaHostAlloc [device to host]: %.6f ms\n", elapsed\_time);

return 0;

}

//lab6\_1.cu

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#include <cuda.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#define N (1024\*1024)

#define FULL\_DATA\_SIZE (N\*20)

bool check(float \*host\_a, float \*host\_b, float \*host\_c)

{

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

if (host\_a[i] + host\_b[i] - host\_c[i] > 1e-5) {

fprintf(stderr, "Wrong result!\n");

return false;

}

}

return true;

}

bool check(long long \*host\_a, long long \*host\_b, long long \*host\_c)

{

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

if (host\_a[i] \* host\_b[i] != host\_c[i]) {

fprintf(stderr, "Wrong result!\n");

return false;

}

}

return true;

}

\_\_global\_\_ void add(float\*a, float \*b, float \*c)

{

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

if (idx < N) {

c[idx] = a[idx] + b[idx];

}

}

\_\_global\_\_ void mult(long long \*a, long long \*b, long long \*c)

{

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

if (idx < N) {

c[idx] = a[idx] \* b[idx];

}

}

void addVectors()

{

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

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

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

cudaMalloc((void\*\*)&dev\_a0, N \* sizeof(float));

cudaMalloc((void\*\*)&dev\_b0, N \* sizeof(float));

cudaMalloc((void\*\*)&dev\_c0, N \* sizeof(float));

cudaMalloc((void\*\*)&dev\_a1, N \* sizeof(float));

cudaMalloc((void\*\*)&dev\_b1, N \* sizeof(float));

cudaMalloc((void\*\*)&dev\_c1, N \* sizeof(float));

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

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

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

srand(time(NULL));

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

host\_a[i] = i + 1;

host\_b[i] = i + 1;

}

float elapsedTime;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaStream\_t stream0, stream1;

cudaStreamCreate(&stream0);

cudaStreamCreate(&stream1);

cudaEventRecord(start, 0);

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

printf("i = %d\n", i);

cudaMemcpyAsync(dev\_a0, host\_a + i, N \* sizeof(float), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_a1, host\_a + i + N, N \* sizeof(float), cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(dev\_b0, host\_b + i, N \* sizeof(float), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_b1, host\_b + i + N, N \* sizeof(float), cudaMemcpyHostToDevice, stream1);

add << <N / 256, 256, 0, stream0 >> >(dev\_a0, dev\_b0, dev\_c0);

add << <N / 256, 256, 0, stream1 >> >(dev\_a1, dev\_b1, dev\_c1);

cudaMemcpyAsync(host\_c + i, dev\_c0, N \* sizeof(float), cudaMemcpyDeviceToHost, stream0);

cudaMemcpyAsync(host\_c + i + N, dev\_c1, N \* sizeof(float), cudaMemcpyDeviceToHost, stream1);

}

cudaStreamSynchronize(stream0);

cudaStreamSynchronize(stream1);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

if (!check(host\_a, host\_b, host\_c))

printf("Something goes wrong!\n");

printf("Elapsed time: %3.1f ms\n", elapsedTime);

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);

}

void multVectors()

{

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

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

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

cudaMalloc((void\*\*)&dev\_a0, N \* sizeof(long long));

cudaMalloc((void\*\*)&dev\_b0, N \* sizeof(long long));

cudaMalloc((void\*\*)&dev\_c0, N \* sizeof(long long));

cudaMalloc((void\*\*)&dev\_a1, N \* sizeof(long long));

cudaMalloc((void\*\*)&dev\_b1, N \* sizeof(long long));

cudaMalloc((void\*\*)&dev\_c1, N \* sizeof(long long));

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

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

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

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

host\_a[i] = i + 1;

host\_b[i] = i + 1;

}

float elapsedTime;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaStream\_t stream0, stream1;

cudaStreamCreate(&stream0);

cudaStreamCreate(&stream1);

cudaEventRecord(start, 0);

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

printf("i = %d\n", i);

cudaMemcpyAsync(dev\_a0, host\_a + i, N \* sizeof(long long), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_a1, host\_a + i + N, N \* sizeof(long long), cudaMemcpyHostToDevice, stream1);

cudaMemcpyAsync(dev\_b0, host\_b + i, N \* sizeof(long long), cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(dev\_b1, host\_b + i + N, N \* sizeof(long long), cudaMemcpyHostToDevice, stream1);

mult << <N / 256, 256, 0, stream0 >> >(dev\_a0, dev\_b0, dev\_c0);

mult << <N / 256, 256, 0, stream1 >> >(dev\_a1, dev\_b1, dev\_c1);

cudaMemcpyAsync(host\_c + i, dev\_c0, N \* sizeof(long long), cudaMemcpyDeviceToHost, stream0);

cudaMemcpyAsync(host\_c + i + N, dev\_c1, N \* sizeof(long long), cudaMemcpyDeviceToHost, stream1);

}

cudaStreamSynchronize(stream0);

cudaStreamSynchronize(stream1);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

if (!check(host\_a, host\_b, host\_c))

printf("Something goes wrong!\n");

printf("Elapsed time: %3.1f ms\n", elapsedTime);

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);

}

int main(int argc, char const \*argv[])

{

cudaDeviceProp prop;

int whichDevice;

cudaGetDevice(&whichDevice);

cudaGetDeviceProperties(&prop, whichDevice);

if (!prop.deviceOverlap) {

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

return 0;

}

printf("FULL\_DATA\_SIZE = %d\n", FULL\_DATA\_SIZE);

printf("Add vectors: \n");

addVectors();

printf("Mult vectors: \n");

multVectors();

return 0;

}