Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

ЛАБОРАТОРНАЯ РАБОТА №9

По дисциплине: «Программирование графических процессоров»

Выполнили:

Студенты 3 курса группы ИП-111
Корнилов А.А.,
Попов М.И.,
Толкач А.А.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2024

Задание:

- 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 распараллеливания копирования и выполнения на основе потоков CUDA;
- то же для реализации скалярного умножения.

Цель: изучить преимущества использования потоков CUDA.

Оборудование: Видеокарта GTX 1050TI (Pascal)

Выполнение работы:

Для выполнения работы была написана программа для сравнения скорости копирования вектора используя стандартное копирование и используя закрепленную память с устройства на хост и обратно.

```
#include <iostream>
#include <cstdlib>
#include <cuda_runtime.h>

using namespace std;

int main() {
    int num = 1 << 12;
    int size = 32 * num;
    float *device, *hostPinned, *host, time = 0;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    host = (float *) malloc(size * sizeof(float));
    cudaMallocHost((void **) &hostPinned, size * sizeof(float));
    cudaMalloc((void **) &device, size * sizeof(float));
    cudaMemset(device, 1024, size * sizeof(float));
    cudaMemcpy(device, host, size * sizeof(float), cudaMemcpyHostToDevice);

    cudaEventRecord(start, nullptr);
    cudaMemcpy(host, device, size * sizeof(float), cudaMemcpyDeviceToHost);
    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    cout << "Стандартное копирование с device на host:             "<< time <<
endl;

    cudaStream_t stream;
```

```
    cudaStreamCreate(&stream);
    cudaEventRecord(start, nullptr);
    cudaMemcpyAsync(hostPinned, device, size * sizeof(float),
cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);
    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    time = 0;
    cudaEventElapsedTime(&time, start, stop);
    cout << "Закрепленная память (pinned memory) с device на хост: "<< time  <<
endl;

    cudaEventRecord(start, nullptr);
    cudaMemcpy(device, host, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    time = 0;
    cudaEventElapsedTime(&time, start, stop);
    cout << "Стандартное копирование с host на device:             " << time <<
endl;

    cudaEventRecord(start, nullptr);
    cudaMemcpyAsync(device, hostPinned, size * sizeof(float),
cudaMemcpyHostToDevice, stream);
    cudaStreamSynchronize(stream);
    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    time = 0;
    cudaEventElapsedTime(&time, start, stop);
    cout << "Закрепленная память (pinned memory) с хост на device: " << time <<
endl;

    cudaFree(device);
    cudaFreeHost(host);
    free(host);
    return 0;
}
```

Листинг 1 – программа LR09_G1.cu

Команда компиляции и результат работы программы:

```
Стандартное копирование с device на host:          0.347648
Закрепленная память (pinned memory) с device на хост: 0.093056
Стандартное копирование с host на device:          0.139168
Закрепленная память (pinned memory) с хост на device: 0.088032

Process finished with exit code 0
```

В результате использование копирования используя стандартное копирование работает медлен при копировании с устройства на хост чем при использовании закреплённой памяти, и также наоборот копирование с хост на устройство выигрывает стандартная память

Для второго задания написана программа для сложения векторов используя закреплённую память и cudastream. Используется деление на потоки от размера portion_size, размер взят 32*2^12

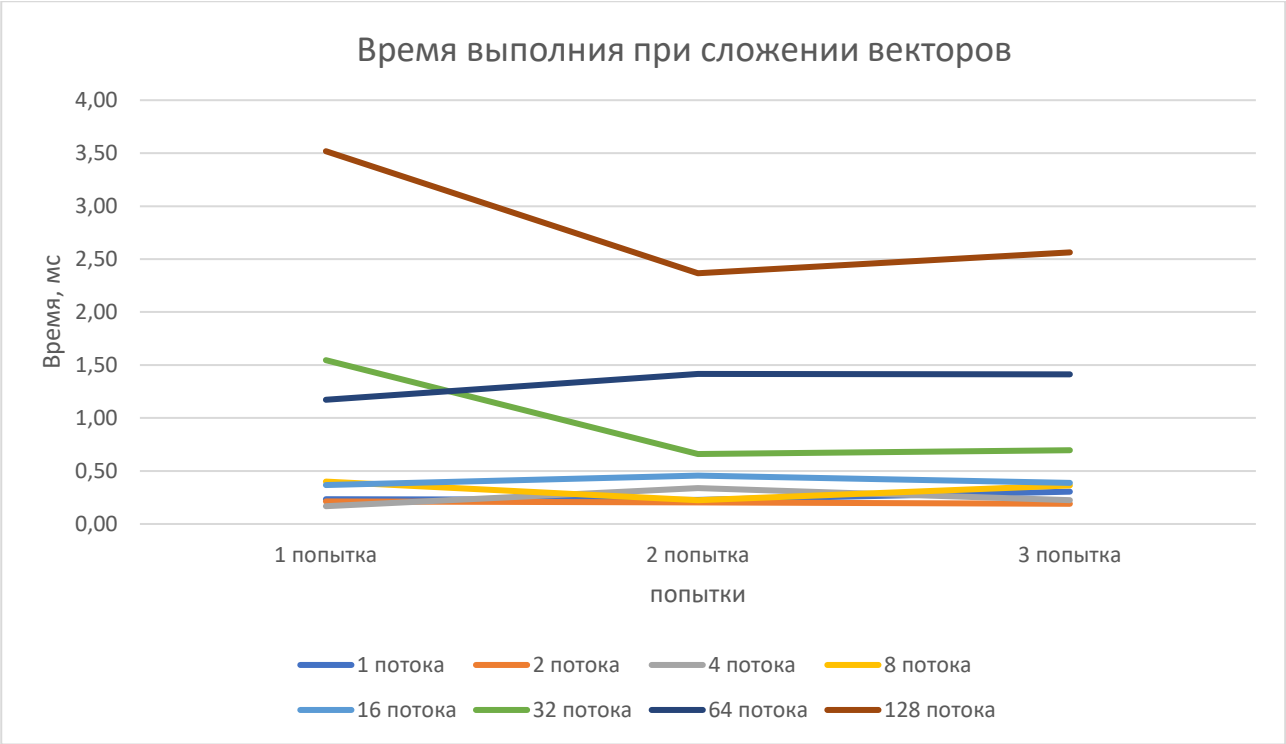
```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

using namespace std;

void show_mass(float *a, int num){
    for (int i = 0; i < num; i++) {
        printf("%f ",a[i]);
        if (i%10 == 0) printf("\n");
    }
    printf("\n");
}
__global__ void addVectors(float *a, float *b, float *c, int n) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n){
        c[i] = a[i] + b[i];
    }
}

int main() {
    int stream_num = 1;
    int num = 1 << 12;
    int size = 32 * num;
    int portion_size = size / stream_num;

    float *h_a, *h_b, *h_c;
    float *d_a, *d_b, *d_c;
    float time = 0;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    auto *streams = (cudaStream_t*)calloc(stream_num, sizeof(cudaStream_t));
    for (int i = 0; i < stream_num; i++) cudaStreamCreate(&streams[i]);

    cudaMallocHost((void **) &h_a, size * sizeof(float));
    cudaMallocHost((void **) &h_b, size * sizeof(float));
    cudaMallocHost((void **) &h_c, size * sizeof(float));
    cudaMalloc((void **)&d_a, size * sizeof(float));
    cudaMalloc((void **)&d_b, size * sizeof(float));
    cudaMalloc((void **)&d_c, size * sizeof(float));
```

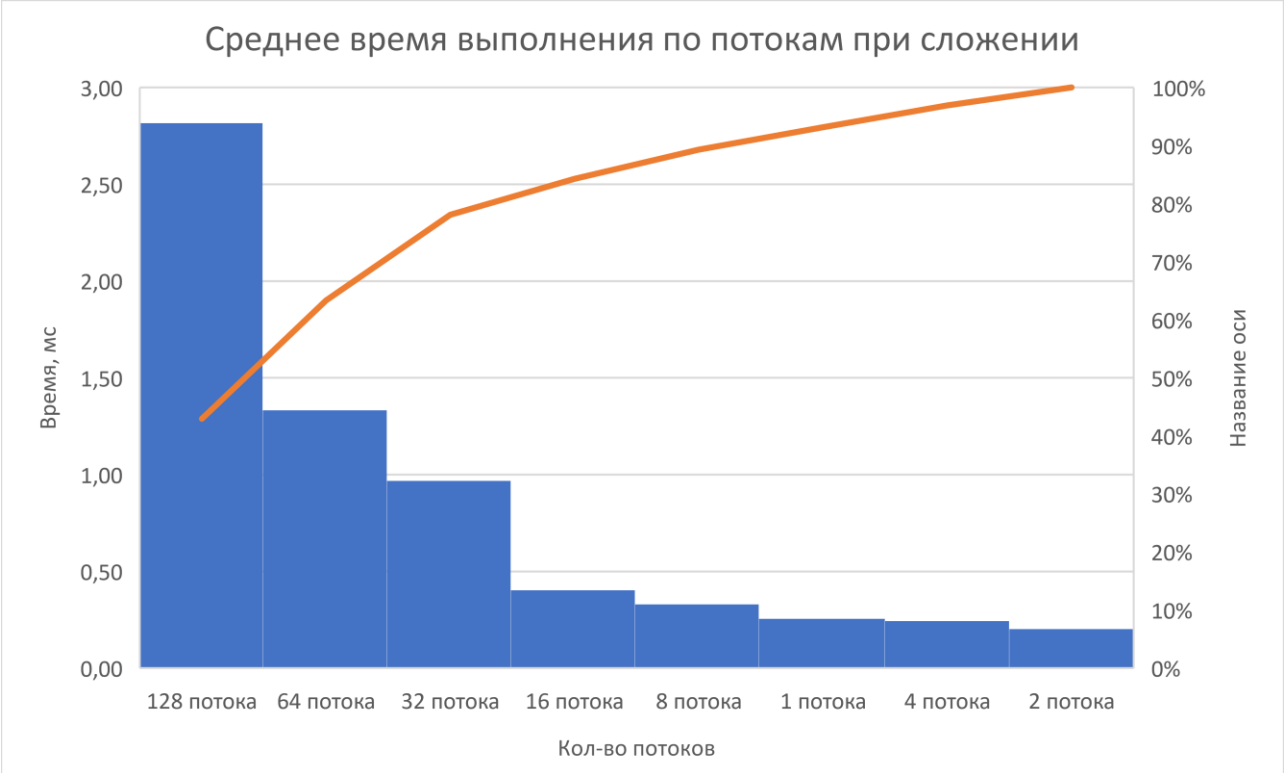
```
    for (int i = 0; i < size; i++) {
        h_a[i] = i;
        h_b[i] = i + 1;
    }

    for (int i = 0; i < stream_num; i++) {
        cudaMemcpyAsync(d_a + i * portion_size, h_a + i * portion_size,
portion_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
        cudaMemcpyAsync(d_b + i * portion_size, h_b + i * portion_size,
portion_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
    }
    for (int i = 0; i < stream_num; i++) cudaStreamSynchronize(streams[i]);

    dim3 threadsPerBlock(256);
    dim3 numBlocks((portion_size + threadsPerBlock.x - 1) / threadsPerBlock.x);

    cudaEventRecord(start, nullptr);
    for (int i = 0; i < stream_num; i++)
        addVectors<<<numBlocks, threadsPerBlock, 0, streams[i]>>>(d_a + i *
portion_size, d_b + i * portion_size, d_c + i * portion_size, portion_size);

    for (int i = 0; i < stream_num; i++) {
        cudaMemcpyAsync(h_c + i * portion_size, d_c + i * portion_size,
portion_size * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
    }
    for (int i = 0; i < stream_num; i++) cudaStreamSynchronize(streams[i]);

    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    cout << "time = "<< time << endl;

    //show_mass(h_c, 100);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(h_a);
    cudaFree(h_b);
    cudaFree(h_c);
    for (int i = 0; i < stream_num; i++) cudaStreamDestroy(streams[i]);

    return 0;
}
```

Листинг 1 – программа LR09_G2.cu

Результат работы программы:

```
time = 5.97504

Process finished with exit code 0
```





В результате программы мы видим с увеличением количеством потоков время работы увеличивается, для размера вектора в 32*2^12 оптимально использовать 2 потока

Для проверки скалярного умножение была взята таже программа но добавлена функция скалярного умножения dotProduct.

```cpp
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

using namespace std;

void show_mass(float *a, int num){
    for (int i = 0; i < num; i++) {
        printf("%f ",a[i]);
        if (i%10 == 0) printf("\n");
    }
    printf("\n");
}
__global__ void dotProduct(float *a, float *b, float *c, int n) {
    __shared__ float temp[256];
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    int tid = threadIdx.x;

    temp[tid] = 0;
    for (int i = index; i < n; i += stride) temp[tid] += a[i] * b[i];

    __syncthreads();

    int i = blockDim.x / 2;
    while (i != 0) {
        if (tid < i) temp[tid] += temp[tid + i];
        __syncthreads();
        i /= 2;
    }
    if (tid == 0) c[blockIdx.x] = temp[0];

}

int main() {
    int stream_num = 128;
    int num = 1 << 12;
    int size = 32 * num;
    int portion_size = size / stream_num;

    float *h_a, *h_b, *h_c;
    float *d_a, *d_b, *d_c;
    float time = 0;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    auto *streams = (cudaStream_t*)calloc(stream_num, sizeof(cudaStream_t));
    for (int i = 0; i < stream_num; i++) cudaStreamCreate(&streams[i]);

    cudaMallocHost((void **) &h_a, size * sizeof(float));
    cudaMallocHost((void **) &h_b, size * sizeof(float));
    cudaMallocHost((void **) &h_c, size * sizeof(float));
    cudaMalloc((void **)&d_a, size * sizeof(float));
    cudaMalloc((void **)&d_b, size * sizeof(float));
```

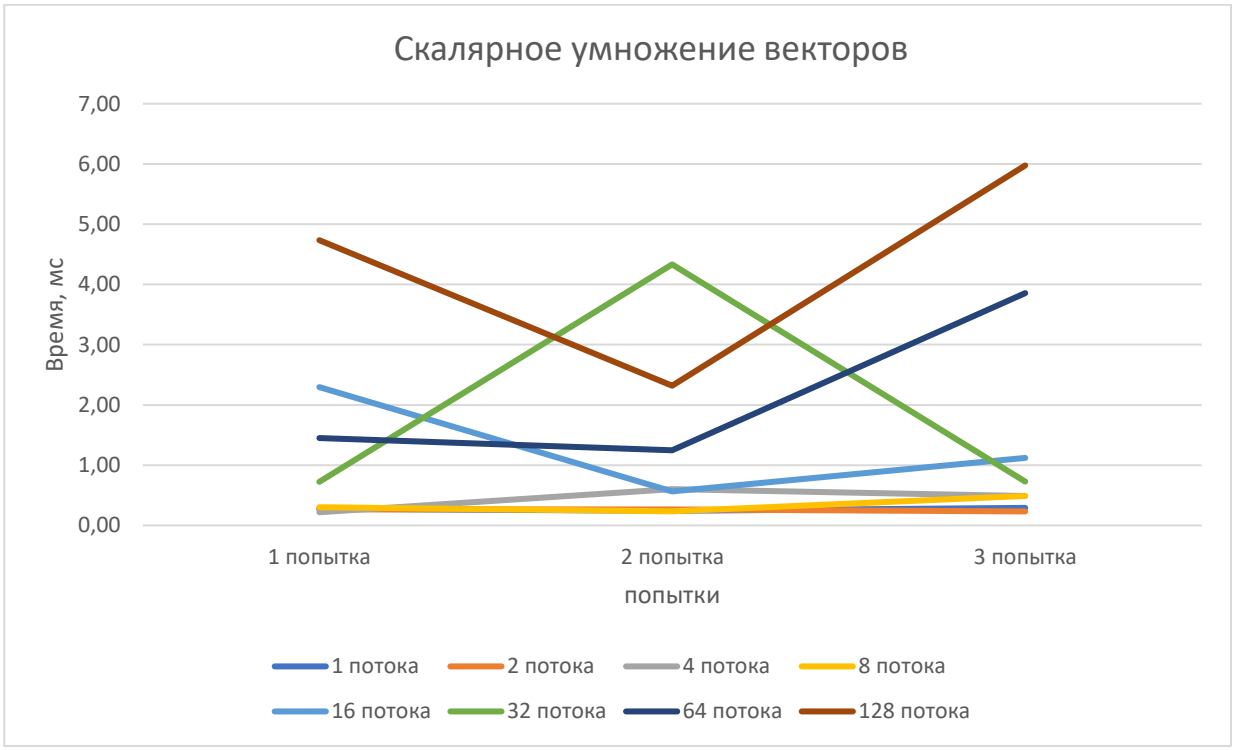
```
    cudaMalloc((void **)&d_c, size * sizeof(float));

    for (int i = 0; i < size; i++) {
        h_a[i] = i;
        h_b[i] = i + 1;
    }

    for (int i = 0; i < stream_num; i++) {
        cudaMemcpyAsync(d_a + i * portion_size, h_a + i * portion_size,
portion_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
        cudaMemcpyAsync(d_b + i * portion_size, h_b + i * portion_size,
portion_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
    }
    for (int i = 0; i < stream_num; i++) cudaStreamSynchronize(streams[i]);

    dim3 threadsPerBlock(256);
    dim3 numBlocks((portion_size + threadsPerBlock.x - 1) / threadsPerBlock.x);

    cudaEventRecord(start, nullptr);
    for (int i = 0; i < stream_num; i++)
        dotProduct<<<numBlocks, threadsPerBlock, 0, streams[i]>>>(d_a + i *
portion_size, d_b + i * portion_size, d_c + i * portion_size, portion_size);

    for (int i = 0; i < stream_num; i++) {
        cudaMemcpyAsync(h_c + i * portion_size, d_c + i * portion_size,
portion_size * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
    }
    for (int i = 0; i < stream_num; i++) cudaStreamSynchronize(streams[i]);

    cudaEventRecord(stop, nullptr);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    cout << "time = "<< time << endl;

    //show_mass(h_c, 100);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(h_a);
    cudaFree(h_b);
    cudaFree(h_c);
    for (int i = 0; i < stream_num; i++) cudaStreamDestroy(streams[i]);

    return 0;
}
```

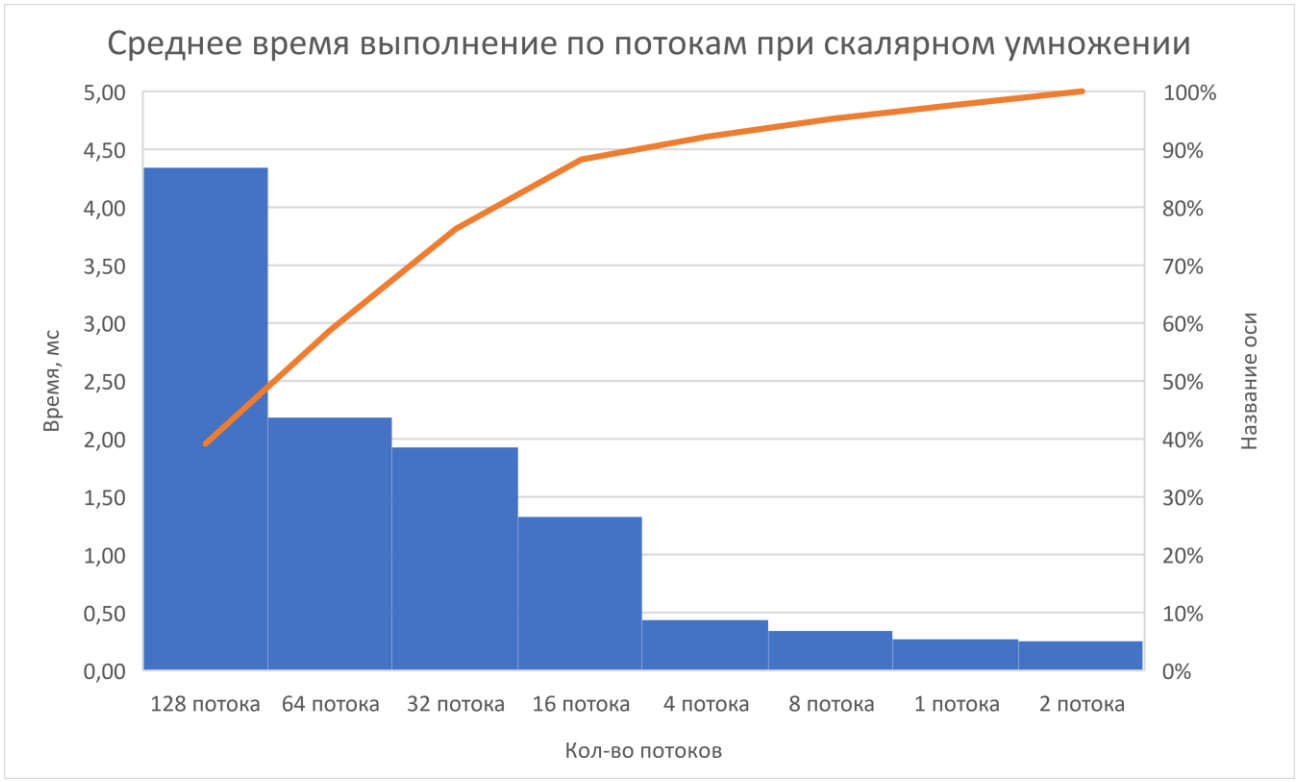
Листинг 1 – программа LR09_G3.cu

Результат работы программы:

```
time = 6.53005
```

```
Process finished with exit code 0
```





При умножении также наблюдаем что при увелечении количества потоков время выполения увеличиваеться. Также видно что при использовании 2 потоков время выполнения меньше всего.

Вывод: в ходе выполнения лабораторной работы, была исследована и применена работа с библиотекой с закреплённой память, стало понятно что работа с ней быстрее по сравнению с обычным выделение памяти, также была исследована работа с cudastream, можно сделать вывод что при использовании одной видеокарты не имеет смысл использовать cudastream так с увеличением количества потоков время выполнения растет.