Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

ЛАБОРАТОРНАЯ РАБОТА №8

По дисциплине: «Программирование графических процессоров»

Выполнили:

Студенты 3 курса группы ИП-211
Оганесян А.С.
Лацук А.Ю.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2025

Задание: включите в компоновку исполняемого файла (компоновщик nvcc) файлы .ptx, основываясь на процедуре, представленной в Лекции 8.

Цель: знакомство с этапами компиляции nvcc.

Выполнение работы:

1. Напишем реализацию функции add в add.cu и main.cu, где эта реализация будет применена:

```
extern "C" __global__ void add(float* a, float* b, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        a[idx] += b[idx];
    }
}
```

Листинг 1 - add.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <vector>
#include <fstream>
#include <cuda.h>

char* loadPTXFile(const char* filePath, size_t* size) {
    std::ifstream file(filePath, std::ios::binary | std::ios::ate);
    if (!file.is_open()) {
        fprintf(stderr, "Не удалось открыть файл %s\n", filePath);
        return nullptr;
    }

    *size = file.tellg();
    file.seekg(0, std::ios::beg);

    char* buffer = new char[*size + 1];
    file.read(buffer, *size);
    buffer[*size] = '\0';

    file.close();
    return buffer;
}

int main() {
```

```
    int N = 1024;
    float *a, *b;
    float *d_a, *d_b;

    a = (float*)malloc(N * sizeof(float));
    b = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; i++) {
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    cudaMalloc(&d_a, N * sizeof(float));
    cudaMalloc(&d_b, N * sizeof(float));

    cudaMemcpy(d_a, a, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N * sizeof(float), cudaMemcpyHostToDevice);

    size_t ptxSize;
    char* ptxSource = loadPTXFile("kernel.ptx", &ptxSize);
    if (!ptxSource) {
        return 1;
    }

    CUresult result;
    CUdevice device;
    CUcontext context;
    CUmodule module;
    CUfunction kernel;

    result = cuInit(0);
    if (result != CUDA_SUCCESS) {
        fprintf(stderr, "Ошибка инициализации CUDA Driver API\n");
        delete[] ptxSource;
        return 1;
    }

    result = cuDeviceGet(&device, 0);
    if (result != CUDA_SUCCESS) {
        fprintf(stderr, "Ошибка получения устройства CUDA\n");
        delete[] ptxSource;
        return 1;
    }

    result = cuCtxCreate(&context, 0, device);
    if (result != CUDA_SUCCESS) {
```

```
        fprintf(stderr, "Ошибка создания контекста CUDA\n");
        delete[] ptxSource;
        return 1;
    }

    result = cuModuleLoadDataEx(&module, ptxSource, 0, 0, 0);
    if (result != CUDA_SUCCESS) {
      const char* errorStr;
      cuGetErrorString(result, &errorStr);
      printf("Ошибка загрузки PTX: %s\n", errorStr);
      delete[] ptxSource;
      return 1;
  }

    result = cuModuleGetFunction(&kernel, module, "add");
    if (result != CUDA_SUCCESS) {
        fprintf(stderr, "Ошибка получения функции ядра\n");
        delete[] ptxSource;
        cuModuleUnload(module);
        cuCtxDestroy(context);
        return 1;
    }

    // Запускаем ядро
    int blockSize = 128;
    int gridSize = (N + blockSize - 1) / blockSize;

    void* args[] = { &d_a, &d_b, &N };
    result = cuLaunchKernel(kernel,
                            gridSize, 1, 1,
                            blockSize, 1, 1,
                            0, 0,
                            args, 0);

    if (result != CUDA_SUCCESS) {
        fprintf(stderr, "Ошибка запуска ядра\n");
        delete[] ptxSource;
        cuModuleUnload(module);
        cuCtxDestroy(context);
        return 1;
    }

    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Ошибка ядра: %s\n", cudaGetErrorString(err));
        delete[] ptxSource;
```

```
        cuModuleUnload(module);
        cuCtxDestroy(context);
        return 1;
    }

    cudaMemcpy(a, d_a, N * sizeof(float), cudaMemcpyDeviceToHost);

    for (int i = 0; i < 5; i++) {
        printf("a[%d] = %f\n", i, a[i]);
    }

    delete[] ptxSource;
    cuModuleUnload(module);
    cuCtxDestroy(context);
    cudaFree(d_a);
    cudaFree(d_b);
    free(a);
    free(b);

    return 0;
}
```

Листинг 2 - main.cu

2. Сгенерируем .ptx файл при помощи команды:

```
nvcc --ptx -arch=sm_86 add.cu -o kernel.ptx
```

3. Скомпилируем основную программу, которая соберет этот .ptx файл

```
nvcc mst.cu -o main -lcuda -lcudart
```

Вывод программы:

```
a[0] = 3.000000
a[1] = 3.000000
a[2] = 3.000000
a[3] = 3.000000
a[4] = 3.000000
```

Вывод: Выполнив эту лабораторную, мы научились подставлять .ptx файлы в компоновку программы. Это может быть полезно для адаптации к разным GPU без перекомпиляции всей программы или генерации своего .ptx под разные архитектуры