Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики»

Лабораторная работа №3

Выполнил: студент 3 курса
ИВТ, гр. ИП-713
Михеев Н.А.

Новосибирск, 2020 г.

Задание на лабораторную работу

Часть 1:

• определить для своего устройства зависимость теоретической заполняемости мультипроцессоров от числа нитей в блоке;

• для программы инициализации вектора определить достигнутую заполняемость в зависимости от длины вектора.

Примечание: использовать nvprof (пример: nvprof -m achieved_occupancy ./lab3) или nvvp, добавив метрику achieved_occupancy.

Часть 2:

• применяя двумерную индексацию нитей в блоке и блоков в гриде написать программу инициализации матрицы, сравнить эффективность кода ядра при двух различных линейных индексациях массива;

• написать программу транспонирования матрицы.

Примечание: для профилирования программы использовать nvprof и nvpp.

Цель: изучить модель выполнения CUDA, варпы, совместный доступ к глобальной памяти.

Часть 1

Теоретическая «заполняемоесть» моей видеокарты — MX150 с видеопамятью размером в 2GB должна достигать 100% при варьировании количества нитей от 64 до 1024 с шагом в 2 раза. Что значит: от 64 потоков/32 блоков/2 варпов на блок до 1024 потоков/2 блоков/32 варпов на блок.

В ходе выполнения практической части лабораторной работы была разработана функция: gInitVector() - служит для инициализации вектора числом 1000.

Практическая зависимость заполняемости от длины вектора (число потоков 128) предоставлена в таблице ниже. Для определения уровня заполненности был использован профилировщик nvprof с использованием метрики achieved_occupancy

Длина	1024	16384	65536	262144	4194304	268435456
Заполняемость	0.164022	0.935681	0.934543	0.920396	0.894266	0.889851

Листинг программы №1

```
#include <iostream>

__global__
void gInitVector(float *vec, int n)
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i < n)
```

```
vec[i] = 1000.;
}

int main()
{
float *A;
int n;
std::cin >> n;
int blockSize = 128;
int numBlocks = (n + blockSize - 1) / blockSize;
A = new float [n];

cudaMallocManaged(&A, n * sizeof(float));
gInitVector<<<numBlocks, blockSize>>>(A, n);
cudaDeviceSynchronize();
cudaFree(A);

return 0;
}
```

Задание 2

Для выполнения второго задания были разработаны функции: matrix_init() - двухмерная инициализация матрицы. И две функции для сравнения двумерной инициализации по X и двумерной инициализации по У.

```
Projects/CUDA_Course/Lab3
❯ sudo nvprof -m achieved_occupancy ./lab3_2
1024
==6744== NVPROF is profiling process 6744, command: ./lab3_2
==6744== Profiling application: ./lab3_2
==6744== Profiling result:
==6744== Metric result:
Invocations                               Metric Name                    Metric Description         Min         Max         Avg
Device "GeForce MX150 (0)"
    Kernel: matrixInitByX(float*)
          1                    achieved_occupancy                    Achieved Occupancy    0.593203    0.593203    0.593203
    Kernel: matrixInitByY(float*)
          1                    achieved_occupancy                    Achieved Occupancy    0.580739    0.580739    0.580739
    Kernel: transponse0(float*, float*)
          1                    achieved_occupancy                    Achieved Occupancy    0.724268    0.724268    0.724268
```

Рис. 1 — результат работы профилировщика nvprof

На Рис. 1 видно, что эффективность кода реализации инициализации по X и по У отличается, но не существенно. При индексации по «оси» X достигаемая заполняемость — 0.5932, по «оси» У — 0.5807.
Для получения данных был использован профилировщик nvprof с метрикой achieved_occupancy

Листинг программы №2

```
#include <iostream>
#include <cmath>

#include <cuda_runtime.h>

__global__ void matrixInitByX(float *X)
```

```
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int l = blockDim.x * gridDim.x;
    X[i + j * l] = (float) (threadIdx.x + blockDim.y * blockIdx.x);
}

__global__ void matrixInitByY(float *X)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int J = blockDim.x * gridDim.x;
    X[i + j * J] = (float) (threadIdx.y + blockDim.x * blockDim.y);
}

__global__ void transponse0(float *X, float *X_t)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int N = blockDim.x * gridDim.x;

    X_t[j + i * N] = X[i + j * N];
}

int main(int argc, char* argv[])
{
    float *A_CUDA, *A_CUDA2, *A_CUDA_T;
    int n = 1024;
    std::cout << n << std::endl;
    int numThreads = 32;
    int blockNum = n / numThreads;
    cudaMalloc((void **) &A_CUDA, n * n * sizeof(float));
    cudaMalloc((void **) &A_CUDA2, n * n * sizeof(float));
    cudaMallocManaged(&A_CUDA_T, n * n * sizeof(float));

    matrixInitByX<<<dim3(blockNum, blockNum), dim3(numThreads, numThreads)>>>(A_CUDA);
    cudaDeviceSynchronize();
    matrixInitByY<<<dim3(blockNum, blockNum), dim3(numThreads, numThreads)>>>(A_CUDA2);
    cudaDeviceSynchronize();
    transponse0<<<dim3(blockNum, blockNum), dim3(numThreads, numThreads)>>>(A_CUDA,
A_CUDA_T);
    cudaDeviceSynchronize();

    cudaFree(A_CUDA);
    cudaFree(A_CUDA2);
    cudaFree(A_CUDA_T);
    return 0;
}
```