Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

09.03.01 "Информатика и вычислительная техника"
профиль "Программное обеспечение средств
вычислительной техники и автоматизированных систем"

ОТЧЕТ
по дисциплине «Программирование графических процессоров»
лабораторная работа 3

Выполнил:
Разумов Д. Б.
студент группы ИП-811

Проверил:
преподаватель Малков Е.А.

Новосибирск 2020

Оглавление

Задание

Задание 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.

Задание 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.

Листинг

Файл lab3.cu

```
#include <stdio.h>
#include <time.h>
#include <malloc.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);\
	}\
} //макрос для обработки ошибок

__global__ void gInitVectors(long long n, double* vector1, double* vector2) {
	int i = threadIdx.x + blockIdx.x * blockDim.x;
	if (i >= n)
		return;
	vector1[i] = (double)i;
	vector2[i] = (double)i;
}

int main(int argc, char *argv[]) {
	//установить предпочтительную конфигурацию кэша для текущего устройства:
	cudaFuncSetCacheConfig(gInitVectors, cudaFuncCachePreferL1);
```

```
    if (argc < 3) {
        printf("Error: run program with 2 args: vector size, threads per block\n");
        return 1;
    }
    long long vector_size, threads;
    vector_size = atoi(argv[1]);
    threads = atoi(argv[2]);
    double *vector1_d, *vector2_d;

    for (int i = 0; i < 10; i++) {
        CUDA_CHECK_RETURN(cudaMalloc((void**)&vector1_d, vector_size * sizeof(double)));
        CUDA_CHECK_RETURN(cudaMalloc((void**)&vector2_d, vector_size * sizeof(double)));
        gInitVectors <<< vector_size / threads, threads >>> (vector_size, vector1_d, vector2_d);
        CUDA_CHECK_RETURN(cudaGetLastError());

        cudaFree(vector1_d);
        cudaFree(vector2_d);
    }
    return 0;
}
```

Файл task1.sh

```
#!/bin/bash
```

```
nvcc lab3.cu -o lab3.exe

for (( i=32; i<=1024; i+=32 ))
do
	sudo nvprof -m achieved_occupancy ./lab3.exe 1048576 $i
done
```

Файл task2.sh

```
#!/bin/bash

nvcc lab3.cu -o lab3.exe

for (( i=1024; i<=2**25; i*=2 )) # ** это возведение в степень
do
	sudo nvprof -m achieved_occupancy ./lab3.exe $i 128
done
```

Файл matrix.cu

```
#include <stdio.h>
#include <time.h>
#include <malloc.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);\
        }\
} //макрос для обработки ошибок

__global__ void gTranspose0(float* storage_d, float* storage_d_t){
        int i=threadIdx.x+blockIdx.x*blockDim.x;
        int j=threadIdx.y+blockIdx.y*blockDim.y;
        int N=blockDim.x*gridDim.x;
        storage_d_t[j+i*N]=storage_d[i+j*N];
}

__global__ void gInitializeMatrixByRows(long long n, double* matrix_d){
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int N = blockDim.x * gridDim.x;
        matrix_d[i+j*N] = (double)(i+j*N);
}

__global__ void gInitializeMatrixByColumns(long long n, double* matrix_d){
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int j = threadIdx.y + blockIdx.y * blockDim.y;
        int N = blockDim.x * gridDim.x;
        matrix_d[j+i*N] = (double)(j+i*N);
}

int main(int argc, char *argv[]) {
        //установить предпочтительную конфигурацию кэша для текущего
устройства:
```

```
	//cudaFuncSetCacheConfig(gInitVectors, cudaFuncCachePreferL1);

	if (argc < 3) {
		printf("Error: run program with 2 args: n, threads per block\n");
		return 1;
	}

	long long n, threads;
	n = atoi(argv[1]);
	threads = atoi(argv[2]);
	double *matrix1_d, *matrix2_d;

	for (int i = 0; i < 10; i++) {
		CUDA_CHECK_RETURN(cudaMalloc((void**)&matrix1_d, n * n *
sizeof(double)));
		gInitializeMatrixByRows <<< n / threads, threads >>> (n, matrix1_d);
		CUDA_CHECK_RETURN(cudaGetLastError());
		cudaFree(matrix1_d);

		CUDA_CHECK_RETURN(cudaMalloc((void**)&matrix2_d, n * n *
sizeof(double)));
		gInitializeMatrixByColumns <<< n / threads, threads >>> (n,
matrix2_d);
		CUDA_CHECK_RETURN(cudaGetLastError());
		cudaFree(matrix2_d);
	}
	return 0;
}
```

Скриншоты

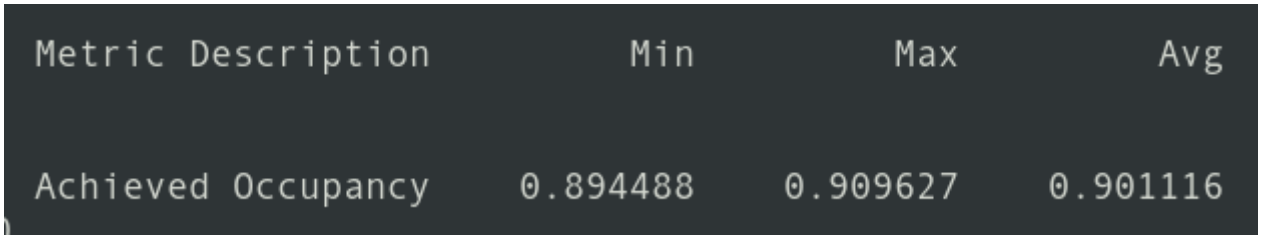

Рисунок 1 — выполнение задания 1, заполняемость мультипроцессоров при 128 нитях на блок

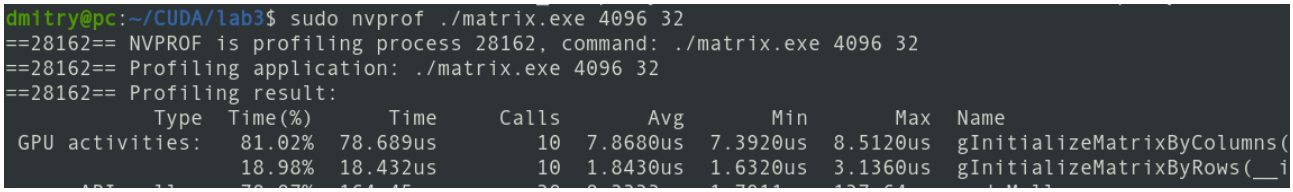

Рисунок 2 — выполнение задания 2

Заключение

Лучшая заполняемость мультипроцессоров оказалась при 128 нитях на блок. Заполняемость в данном случае составила чуть больше 90% (размер вектора равен 1<<20, модель видеокарты Geforce MX350).

Так же я выяснил, что чем больше размер вектора, тем больше загружены мультипроцессоры.

Построчное заполнение матрицы быстрее заполнения по столбцам, так как при построчном заполнении доступ к памяти последовательный. Благодаря последовательному доступу к памяти возникает меньше кэш-промахов.