Федеральное государственное бюджетное образовательное учреждение высшего

образования

«Сибирский государственный университет

телекоммуникаций и информатики»

Кафедра ПМиК

Лабораторная работа № 3

Тема: «Заполняемость мультипроцессоров и совместный доступ к глобальной памяти»

Выполнили: студенты III курса

ИВТ, гр. ИП-713

Лебедев Н.А.

Проверил: ассистент кафедры ПМиК

Нужнов А.В.

Новосибирск 2020

Задание на лабораторную работу

Задание 1:

1. Определить зависимость теоретической заполняемости мультипроцессоров от числа нитей в блоке.
2. Для программы инициализации вектора определить достигнутую заполняемость в зависимости от длины вектора.

Задание 2:

1. Применяя двумерную индексацию нитей в блоке и блоков в гриде написать программу инициализации матрицы, сравнить эффективность кода ядра при двух различных линейных индексациях массива.
2. Написать программу транспонирования матрицы.

Цель лабораторной работы

Изучить модель выполнения CUDA, варпы, совместный доступ к глобальной памяти.

Ход выполнения лабораторной работы

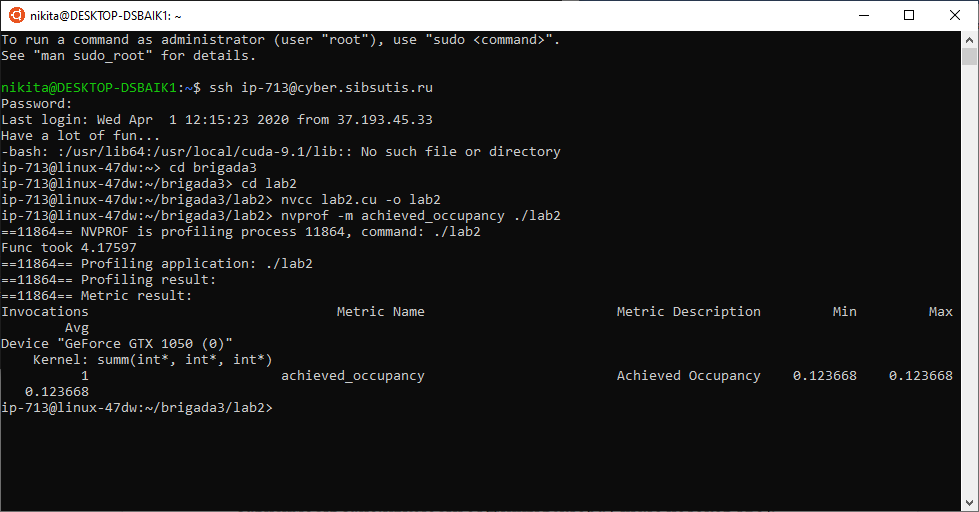
Задание 1

Для выполнения лабораторных работ используется две видеокарты Nvidia GeForce GTX 1050 c размером варпа 32 и максимальным количеством нитей на блок равному 1024. И например, при 256 нитей на блок, максимальном количестве регистров на поток (255) и памяти на блок (49125), заполняемость каждого мультипроцессора равна 13%. Заполняемость высчитывается по формуле: варпы на мультипроцессоре (blocks/SM \* Warps/Block = Warps/SM) на максимальное количество варпов на мультипроцессор (64). И как раз количество варпов на блок (Warps/Block) меняется из-за количества нитей на блок, а именно количество на блок делённое на максимальное количество нитей на варп (32).

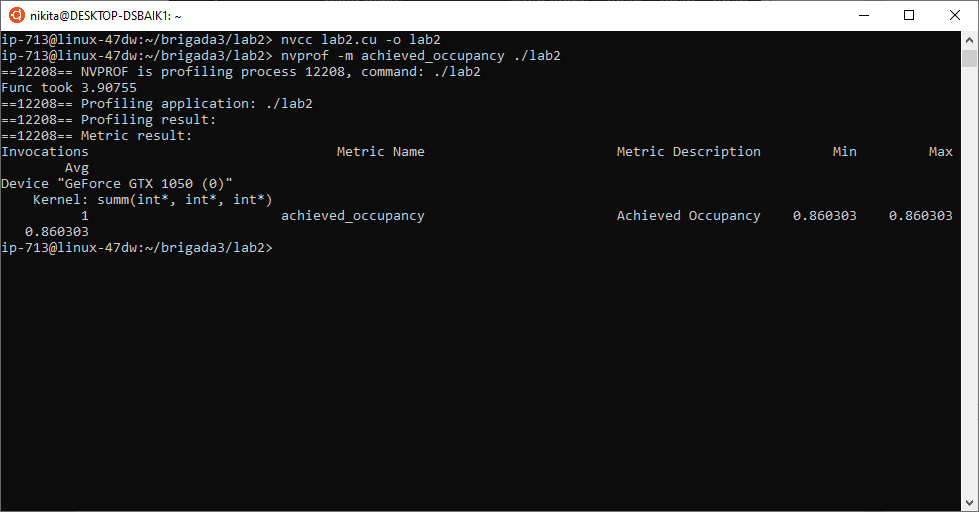
В данной лабораторной работе был также использован инструмент профилирования nvprof. Nvprof позволяет собирать временную шкалу связанных с CUDA действий как на CPU, так и на GPU, включая выполнение ядра, передачу данных, набор памяти и вызовы API CUDA, а также события или метрики для ядер CUDA.

В частности флаг -m, который показывает определённые метрики, в данной лабе мы будем использовать метрику achieved\_occupancy (достигнутая заполняемость).

Используя показатели профилировщика, можно проследить как изменяется заполняемость от размера вектора:

  
Рисунок 1 – длина массива 1<<10

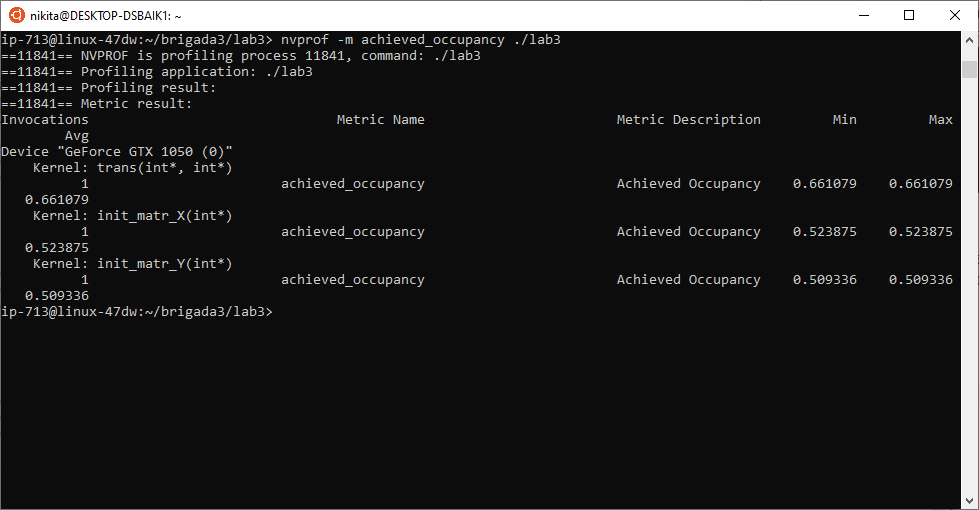
На рисунке №1 показан результат работы профилировщика при размере массива 1<<10 и количества нитей на блок = 256. Профилировщик выводит минимальные, максимальные и средние значения заполняемости, так как во время выполнения программы эти показатели могут меняться. Но в данной программе показатель не меняется и равен 0,123668.

  
Рисунок 2 – длина массива 1<<20

На рисунке №2 же длина массива 1<<20, но количество нитей всё тоже. Показатель же почти не меняется и равен 0,860303. Исходя из этого можно сказать, что изменение размера вектора влияет на заполняемость.

Задание 2

Во втором задании лабораторной работы требуется инициализировать матрицы, используя различные индексации массива(по x и по y).

  
Рисунок 3 – результат работы профилировщика

Исходя из данных представленных профилировщиком Nvprof с флагом -m achieved\_occupancy можно сделать вывод, что эффективность кода отличается, но не намного. При индексации по x – 0.523875, а по y – 0.509336.

Текст программы для задания 1

#include <cuda.h> #include <stdio.h> #include <stdlib.h> #include <ctime> #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);\ }} \_\_global\_\_ void summ(int\* a,int \*b,int \*c){ int idx = threadIdx.x + blockDim.x \* blockIdx.x; c[idx] = a[idx]+b[idx]; } main() { float elapsedTime; cudaEvent\_t start, stop; cudaEventCreate(&start); cudaEventCreate(&stop);

srand(time(0));

int \*da, \*ma, \*rez, \*rezc, \*dac, \*mac;

int num\_of\_blocks, threads\_per\_block=30;

int N=1<<22;

num\_of\_blocks = N / threads\_per\_block;

dac = (int\*)calloc(N,sizeof(int));

mac = (int\*)calloc(N,sizeof(int));

rezc = (int\*)calloc(N,sizeof(int));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&da,N\*sizeof(int)));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&ma,N\*sizeof(int)));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&rez,N\*sizeof(int)));

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

{

dac[i] = rand()%100;

mac[i] = rand()%100;

}

cudaMemcpy(da,dac,N\*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(ma,mac,N\*sizeof(int),cudaMemcpyHostToDevice);

cudaEventRecord(start,0);

summ<<<dim3(num\_of\_blocks),

dim3(threads\_per\_block)>>>(da,ma,rez);

cudaEventRecord(stop,0);

cudaEventSynchronize(stop);

//CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

CUDA\_CHECK\_RETURN(cudaGetLastError());

cudaEventElapsedTime(&elapsedTime,start,stop);

fprintf(stderr,"Func took %g\n",elapsedTime);

cudaEventDestroy(start);

cudaEventDestroy(stop);

CUDA\_CHECK\_RETURN(cudaMemcpy(rezc,rez,N\*sizeof(int),cudaMemcpyDeviceToHost));

free(dac);

free(mac);

free(rezc);

cudaFree(da);

cudaFree(ma);

cudaFree(rez);

return 0;

}

Текст программы для задания 2

#include <cuda.h>

#include <stdio.h>

#include <stdlib.h>

\_\_global\_\_ void init\_matr\_X(int \*A)

{

int i = threadIdx.x + blockIdx.x \* blockDim.x;

int j = threadIdx.y+blockIdx.y\*blockDim.y;

int I = gridDim.x\*blockDim.x;

A[i+j\*I] = (int)(threadIdx.x + blockDim.y \* blockIdx.x);

}

\_\_global\_\_ void init\_matr\_Y(int \*A)

{

int i = threadIdx.x + blockIdx.x \* blockDim.x;

int j = threadIdx.y + blockIdx.y \* blockDim.y;

int J = blockDim.y \* gridDim.y;

A[i+j\*J] = (int)(threadIdx.y + blockDim.y \* blockIdx.x);

}

\_\_global\_\_ void trans(int \*A, int \*B)

{

int i = threadIdx.x + blockIdx.x \* blockDim.x;

int j = threadIdx.y + blockIdx.y \* blockDim.y;

int N = blockDim.x \* gridDim.x;

B[j+i\*N] = A[i+j\*N];

}

int N=1<<10;

int \*dev\_a, \*dev\_b;

main()

{

int num\_of\_blocks, threads\_per\_block = 32;

num\_of\_blocks = N / threads\_per\_block;

cudaMalloc((void \*\*)&dev\_a,sizeof(int)\*N\*N);

cudaMalloc((void \*\*)&dev\_b,sizeof(int)\*N\*N);

dim3 threads = dim3(threads\_per\_block, threads\_per\_block);

dim3 blocks = dim3(num\_of\_blocks, num\_of\_blocks);

init\_matr\_X<<<blocks,threads>>>(dev\_a);

cudaDeviceSynchronize();

init\_matr\_Y<<<blocks,threads>>>(dev\_b);

cudaDeviceSynchronize();

trans<<<blocks,threads>>>(dev\_a, dev\_b);

cudaDeviceSynchronize();

cudaFree(dev\_a);

cudaFree(dev\_b);

return 0;

}