Выполнила: Белоусова Е., ИП-911

Задача

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.

Описание работы программы

Проведем расчёты теоретической части.

A	B	C
Max number of threads per MP	2048	GeForce GT 740M
Max number of warp per MP	2048/32 = 64	
Max number of blocks per MP (capability 3.0)	16	
Оптимальное количество нитей в блоке	2048/16 = 128	

Теоретическая заполняемость мультипроцессоров от числа нитей в блоке	128 блоков						
кол-во одновременно выполняемых нитей	32 * 16 = 512	64 * 16 = 1024	96 * 16 = 1536	128 * 16 = 2048	256 * 16 = 4096	512 * 16 = 8192	1024 * 16 = 16384
максимальное кол-во нитей	64 * 32 = 2048	2048	2048	2048	2048	2048	2048
заполняемость теоретическая	25%	50%	75%	100%	200%	400%	800%
заполняемость реальная	19,7%	40,2%	62,2%	80,80%	75,30%	70,60%	66%

Далее для программы инициализации вектора определим достигнутую заполняемость в зависимости от длины вектора с помощью профилировщика.

```
sonya@sonya-S551LB:~/cuda/lab3$ sudo nvprof -m achieved_occupancy ./31
 6240 128
N = 798720
num_of_blocks = 6240
threads_per_block = 128
==10848== NVPROF is profiling process 10848, command: ./31 6240 128
==10848== Profiling application: ./31 6240 128
==10848== Profiling result:
==10848== Metric result:
Invocations                               Metric Name
      Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GT 740M (0)"
    Kernel: initializing_vector(int*)
          1                        achieved_occupancy
      Achieved Occupancy    0.911451    0.911451    0.911451
```

					2^19		384 ядра * 16 блоков	
Кол-во нитей \ Длина вектора	1024	2048	4096	8192	542288		786432	960000
32	24,4	22,3	20,1	18,6	16,3		16,6	16,7
64	24,5	47,6	43	40,3	41,9		41,8	41,7
128	24,6	47,7	79,9	88,8	90,6	...	91,1	91,2
256	24,5	47,9	79,7	85,7	87,9		88	88,2
512	24,6	47,9	80,3	83,1	81,2		81,5	82,1
1024	48,2	48,4	85,6	82,5	67,2		67,9	68,2

Ссылка на онлайн-таблицу с результатами:

https://docs.google.com/spreadsheets/d/1il5DfzW7HeHa3f4XMzi8Y0CfBgFqNf5xn6O3p0vXNFI/edit?usp=sharing

Листинг

```
//2.cu
#include <cuda.h>
#include <cuda_runtime.h>
#include <iomanip>
#include <iostream>
#include <malloc.h>
#include <stdio.h>

using namespace std;

#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 init(int *c, int N) {
int i = threadIdx.x + blockIdx.x * blockDim.x;
 if(i >= N) return;
 c[i] = 0;
}
```

```
int main(int argc,char* argv[])
{
   // char dev;
   // cudaSetDevice(dev);
   // cudaDeviceProp deviceProp;
   // cudaGetDeviceProperties(&deviceProp, dev);
   // printf("  Total amount of constant memory:  %lu bytes\n", deviceProp.totalConstMem);
   // printf("  Total amount of shared memory per block: %lu bytes\n", deviceProp.sharedMemPerBlock);
   // printf("  Total number of registers available per block: %d\n", deviceProp.regsPerBlock);
   // printf("  Warp size: %d\n", deviceProp.warpSize);
   // printf("  Maximum number of threads per multiprocessor:  %d\n",
deviceProp.maxThreadsPerMultiProcessor);
   // printf("  Maximum number of threads per block:  %d\n", deviceProp.maxThreadsPerBlock);
   float elapsedTime;
   int N = 0;
   int *dev_c, *c;
   N = atoi(argv[1]);
   int th = atoi(argv[2]);
   cudaEvent_t start, stop;
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   fprintf(stderr, "%d blocks\n", N / th);
   c = (int*)calloc(N, sizeof(int));
   CUDA_CHECK_RETURN(cudaMalloc((void **)&dev_c, N * sizeof(int)));

   cudaEventRecord(start, 0);
   init<<<N / th, th>>>(dev_c, N);
   cudaEventRecord(stop, 0);
   cudaEventSynchronize(stop);

   CUDA_CHECK_RETURN(cudaGetLastError());
   cudaEventElapsedTime(&elapsedTime, start, stop);
   CUDA_CHECK_RETURN(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));
```

```
    fprintf(stderr, "%d: %.6f ms\n", N, elapsedTime);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    free(c);
    cudaFree(dev_c);
    cout << endl;
    return 0;
}

//lab31.cu
#include <cuda.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#define onekk 1000000

#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 initializing_vector(int *vector){
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    vector[index] = index;
}

int main(int argc, char *argv[]){
```

```
    int num_of_blocks = atoi(argv[1]);
    int threads_per_block = atoi(argv[2]);
    int N = num_of_blocks * threads_per_block;

    printf("N = %d\n",N);
    printf("num_of_blocks = %d\n",num_of_blocks);
    printf("threads_per_block = %d\n",threads_per_block);



    int *a;
    int *a_gpu;

    a = (int*)malloc(N * sizeof(int));

    CUDA_CHECK_RETURN(cudaMalloc((void**)&a_gpu, N * sizeof(int)));

    initializing_vector <<< dim3(num_of_blocks),
    dim3(threads_per_block) >>> (a_gpu);
    CUDA_CHECK_RETURN(cudaGetLastError());
    CUDA_CHECK_RETURN(cudaMemcpy(a, a_gpu, N * sizeof(int), cudaMemcpyDeviceToHost));

    free(a);
    CUDA_CHECK_RETURN(cudaFree(a_gpu));
}
```