Федеральное агентство связи (Россвязь)

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики (ПМиК)

Отчет по лабораторной работе №3

по дисциплине «Программирование Графических Процессоров»

Выполнил: студент группы ИП-712

Алексеев С.В.

Проверил:

Нужнов А.В.

Новосибирск – 2020 г.

**Содержание**

[**Цель работы** 3](#_Toc35679218)

[**Выполнение работы** 3](#_Toc35679219)

[**Вывод** 4](#_Toc35679220)

[**Приложение** 5](#_Toc35679221)

[**Листинг инициализация матрицы:** 7](#_Toc35679222)

# **Цель работы**

Требуется определить для своего устройства зависимость теоретической заполняемости мультипроцессоров от числа нитей в блоке, а также определить достигнутую заполняемость в зависимости от длины вектора для программы инициализации вектора.

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

# **Выполнение работы**

Работа выполняется в среде Micosoft Visual Studio и в Nvidia Nsight Compute. Для моей видеокарты не поддерживается проверка achieved occupancy через nvprof, поэтому использовал Nvidia Nsight Compute.

Модифицируем программу, написанную в предыдущей лабораторной работе. Сделаем так, чтобы размер массива и кол-во нитей в блоке были фиксированными и вводились с клавиатуры в виде параметров командной строки. Размер массива задается в виде числа как степени двойки (например, "10" – размер массива 210 = 1024). Соответственно, необходимы проверки корректности введенных параметров:

* Размер массива: целое число от 1 до 30
* Кол-во нитей в блоке: целое число от 32 до 1024, кратное 32.

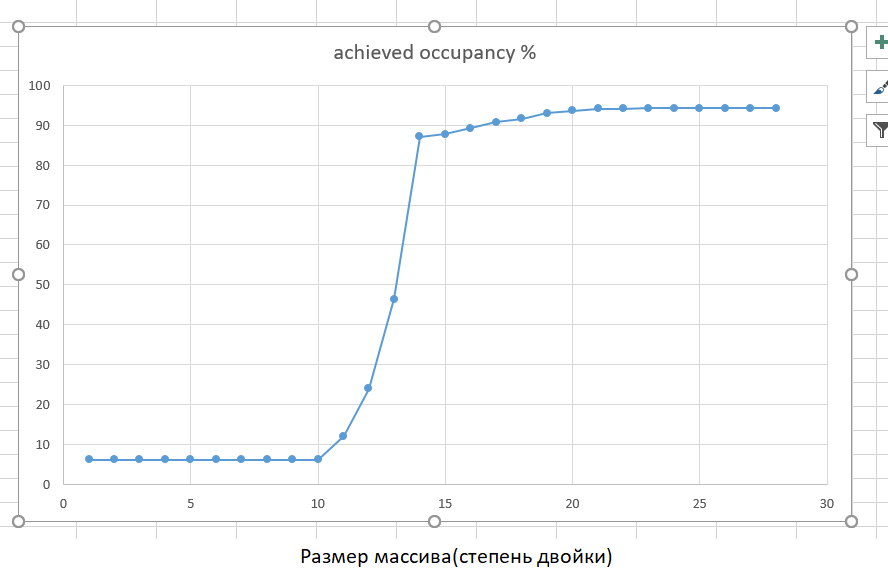


Рисунок 1. Результаты работы программы.

На рис. 1 изображен график результатов работы программы. При фиксированном размере блока – 64 нити – программа была запущена 28 раз, и указывался размер массива от 21 до 228.

# **Вывод**

При относительно небольших размерах массивов (до 210) средняя метрика “achieved occupancy” остается низкой – менее 10%. Затем наблюдается быстрый рост эффективности с максимальным пиком при размере массива 221 – 94%. В дальнейшем рост производительности мало изменяется.

# **Приложение**

Приложение А. Листинг программы.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <algorithm>

#include <cuda.h>

#include <stdio.h>

\_\_global\_\_ void gpuAdd(float\* gpuA, float\* gpuB, float\* gpuC)

{

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

gpuC[id] = gpuA[id] \* gpuB[id];

}

int main(int argc, char\*\* argv)

{

if (argc < 3)

{

printf("Usage: %s [2^size] [tpb]\n", argv[0]);

exit(1);

}

int size = 1 << atoi(argv[1]);

int threads = atoi(argv[2]);

if (size <= 0 || size >= 31)

{

printf("Bad size (%d).\n", size);

exit(1);

}

if (threads <= 0 || threads > 1024 || threads % 32 != 0)

{

printf("Bad threads-per-block (%d).\n", threads);

exit(1);

}

// Создаем обычный огромный массив в оперативной памяти.

float\* cpuFilled = (float\*)malloc(size \* sizeof(float));

float\* cpuEmpty = (float\*)calloc(size, sizeof(float));

float\* cpuResult = (float\*)malloc(size \* sizeof(float));

if (cpuFilled == NULL || cpuEmpty == NULL || cpuResult == NULL)

{

printf("malloc error.\n");

exit(1);

}

// Заполняем этот массив.

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

{

cpuFilled[i] = float(i);

}

// Создаем три огромных массива на видеокарте.

float\* gpuA;

float\* gpuB;

float\* gpuC;

int test1 = cudaMalloc((void\*\*)&gpuA, size \* sizeof(float));

int test2 = cudaMalloc((void\*\*)&gpuB, size \* sizeof(float));

int test3 = cudaMalloc((void\*\*)&gpuC, size \* sizeof(float));

if (test1 == cudaErrorMemoryAllocation ||

test2 == cudaErrorMemoryAllocation ||

test3 == cudaErrorMemoryAllocation)

{

printf("cudaMalloc error (%d %d %d).\n", test1, test2, test3);

exit(1);

}

// Два из них - копии того же массива из оперативы.

// В третьем будет храниться текущий результат.

cudaMemcpy(gpuA, cpuFilled, size \* sizeof(float),

cudaMemcpyHostToDevice);

cudaMemcpy(gpuB, cpuFilled, size \* sizeof(float),

cudaMemcpyHostToDevice);

// Запускаем цикл по числу элементов от минимума до максимума.

int blockCount = std::max(1, size / threads);

//printf(" %6d blocks x %4d threads: ", blockCount, threads);

if (blockCount < 1 && blockCount > 65535)

{

printf("Bad block count (%d).\n", blockCount);

exit(1);

}

if (threads < 1 && threads > 1024)

{

printf("Bad threads-per-block count (%d).\n", threads);

exit(1);

}

cudaMemcpy(gpuC, cpuEmpty, size \* sizeof(float),

cudaMemcpyHostToDevice);

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaEventSynchronize(start);

gpuAdd <<< dim3(blockCount), dim3(threads) >>> (gpuA, gpuB, gpuC);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

float elapsedTime = 0.0f;

cudaEventElapsedTime(&elapsedTime, start, stop);

cudaEventDestroy(start);

cudaEventDestroy(stop);

//cudaDeviceReset();

cudaMemcpy(cpuResult, gpuC, size \* sizeof(float),

cudaMemcpyDeviceToHost);

// Освобождаем память.

cudaFree(gpuA);

cudaFree(gpuB);

cudaFree(gpuC);

free(cpuFilled);

free(cpuEmpty);

free(cpuResult);

return 0;

}

# **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.

# **Выполнение рботы**

На GPU создаю матрицу используя двумерную инициализацию:

A[threadIdx.y \* n + threadIdx.x] = 10 \* threadIdx.y + threadIdx.x;

Здесь используется одномерный массив, но индексация производится с помощью числа n, обозначающего размер строки матрицы. Т.о. получаем матрицу. В работе матрицы с различными линейными индексациями массива различий не выявлено(при сохранении общего числа нитей одинаковым). То есть dim3(4, 25, 1) и dim3(10, 10, 1) работают примерно с одинаковой скоростью.

# **Листинг инициализация матрицы:**

// Сравнение двух матриц

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

\_\_global\_\_ void createMatrix(int\* A, const int n)

{

// Создание элементов матрицы на GPU

A[threadIdx.y \* n + threadIdx.x] = 10 \* threadIdx.y + threadIdx.x;

}

int main()

{

// кол-во строк и столбцов матрицы

const int n = 10;

// размер матрицы

size\_t size = n \* n \* sizeof(int);

// выделяем память для матрицы на CPU

int\* h\_A = (int\*)malloc(size);

// инициализируем матрицу

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

for (int j = 0; j < n; j++)

h\_A[j \* n + i] = 10 \* j + i;

int\* d\_B = NULL;

// выделяем память для матрицы на GPU

cudaMalloc((void\*\*)&d\_B, size);

// определение размеров сетки и блоков

dim3 threadsPerBlock = dim3(10, 10);

dim3 blocksPerGrid = dim3(1);

// вызов ядра

createMatrix << <blocksPerGrid, threadsPerBlock >> > (d\_B, n);

// выделяем память для матрицы B, чтобы

// скопировать из GPU на CPU

int\* h\_B = (int\*)malloc(size);

// копируем матрицу из GPU на CPU

cudaMemcpy(h\_B, d\_B, size,

cudaMemcpyDeviceToHost);

// проверяем совпадение матрицы А и матрицы В

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

for (int j = 0; j < n; j++)

if (h\_A[j \* n + i] != h\_B[j \* n + i]) {

printf("h\_A[%d] != h\_B[%d]\n", j \* n

+ i, j \* n + i);

}

// освобождаем память на GPU

cudaFree(d\_B);

// освобождаем память на CPU

free(h\_A);

free(h\_B);

return 0;

}

# Листинг транспонирование матрицы

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

// размер блока

#define BLOCK\_SIZE 16

// тип, который будут иметь элементы матриц

#define BASE\_TYPE float

// Ядро

// Функция транспонирования матрицы

\_\_global\_\_ void matrixTranspose(const BASE\_TYPE\* A, BASE\_TYPE\* AT, int rows, int cols)

{

// Индекс элемента в исходной матрице

int iA = cols \* (blockDim.y \* blockIdx.y + threadIdx.y) + blockDim.x \* blockIdx.x + threadIdx.x;

// Индекс соответствующего элемента в транспонированной матрице

int iAT = rows \* (blockDim.x \* blockIdx.x + threadIdx.x) + blockDim.y \* blockIdx.y + threadIdx.y;

AT[iAT] = A[iA];

}

// Функция вычисления числа, которое больше числа а и кратное числу b

int toMultiple(int a, int b)

{

int mod = a % b;

if (mod != 0)

{

mod = b - mod;

return a + mod;

}

return a;

}

int main()

{

// Объекты событий

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// Количество строк и столбцов матрицы

int rows = 1000;

int cols = 2000;

// Меняем количество строк и столбцов матрицы

// на число, кратное размеру блока (16)

rows = toMultiple(rows, BLOCK\_SIZE);

printf("rows = %d\n", rows);

cols = toMultiple(cols, BLOCK\_SIZE);

printf("cols = %d\n", cols);

size\_t size = rows \* cols \* sizeof(BASE\_TYPE);

// Выделение памяти под матрицы на хосте

// Исходная матрица

BASE\_TYPE\* h\_A = (BASE\_TYPE\*)malloc(size);

// Транспонированная матрица

BASE\_TYPE\* h\_AT = (BASE\_TYPE\*)malloc(size);

// Инициализация матрицы

for (int i = 0; i < rows \* cols; ++i)

{

h\_A[i] = rand() / (BASE\_TYPE)RAND\_MAX;

}

// Выделение глобальной памяти на девайсе

// для исходной матрицы

BASE\_TYPE\* d\_A = NULL;

cudaMalloc((void\*\*)&d\_A, size);

// Выделение глобальной памяти на девайсе для транспонированной матрицы

BASE\_TYPE\* d\_AT = NULL;

cudaMalloc((void\*\*)&d\_AT, size);

// Копируем матрицу из CPU на GPU

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

// Определяем размер блока и сетки

dim3 threadsPerBlock = dim3(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 blocksPerGrid = dim3(cols / BLOCK\_SIZE, rows / BLOCK\_SIZE);

// Начать отсчет времени

cudaEventRecord(start, 0);

// Запуск ядра

matrixTranspose << <blocksPerGrid,

threadsPerBlock >> > (d\_A, d\_AT, rows, cols);

// Окончание работы ядра, остановка времени

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

float KernelTime;

cudaEventElapsedTime(&KernelTime, start, stop);

printf("KernelTime: %.2f milliseconds\n",

KernelTime);

// Копируем матрицу из GPU на CPU

cudaMemcpy(h\_AT, d\_AT, size,

cudaMemcpyDeviceToHost);

// Проверка правильности работы ядра

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

for (int j = 0; j < cols; j++)

{

if (h\_A[i \* cols + j] != h\_AT[j \* rows + i]) {

fprintf(stderr, "Result verification failed at element[% d, % d]!\n", i, j);

exit(EXIT\_FAILURE);

}

printf("Test PASSED\n");

// Освобождаем память на GPU

cudaFree(d\_A);

// Освобождаем память на GPU

cudaFree(d\_AT);

// Освобождаем память на CPU

free(h\_A);

free(h\_AT);

// Удаляем объекты событий

cudaEventDestroy(start);

cudaEventDestroy(stop);

printf("Done\n");

return 0;

}

}