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 5

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

Проверил:
преподаватель Малков Е.А.

Новосибирск 2020

Оглавление

Задание

Задание:

• реализовать алгоритм вычисления интеграла функции, заданной на прямоугольной сетке в трехмерном пространстве, на сфере с использованием текстурной и константной памяти;

• реализовать алгоритм вычисления интеграла функции, заданной на прямоугольной сетке в трехмерном пространстве, на сфере без использованием текстурной и константной памяти (ступенчатую и линейную интерполяцию в узлы квадратуры на сфере реализовать программно);

• сравнить результаты и время вычислений обоими способами.

Цель: изучить преимущества использования константной и текстурной памяти.

Листинг

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <cuda.h>

#define M_PI 3.14159265358979323846
#define COEF 48
#define VERTCOUNT COEF*COEF*2
#define RADIUS 160.0f
#define FGSIZE 320
#define FGSHIFT FGSIZE/2
#define IMIN(A,B) (A<B ? A : B)
#define THREADSPERBLOCK 256
#define BLOCKSPERGRID
IMIN(32,(VERTCOUNT+THREADSPERBLOCK-1)/THREADSPERBLOCK)

typedef float(*ptr_f)(float, float, float);

struct Vertex { //вершина
	float x, y, z;
};

__constant__ Vertex vert[VERTCOUNT];
//текстура<тип, размерность текстуры, без нормализации> тексурная ссылка:
texture<float, 3, cudaReadModeElementType> df_tex;
//указатель на область памяти, предназначенную для работы с текстурой:
```

```
cudaArray* df_Array = 0;

//тестовая функция:
float func(float x, float y, float z) {
	return (0.5 * sqrtf(15.0 / M_PI)) * (0.5 * sqrtf(15.0 / M_PI)) *
		z * z * y * y * sqrtf(1.0f - z * z / RADIUS / RADIUS) / RADIUS /
RADIUS
		/ RADIUS / RADIUS;
}

//проверочная фу-ия:
float check(Vertex* v, ptr_f f) {
	float sum = 0.0f;
	for (int i = 0; i < VERTCOUNT; ++i)
		sum += f(v[i].x, v[i].y, v[i].z);
	return sum;
}

//дискретизация функций на прямоугольной сетке:
void calc_f(float* arr_f, int x_size, int y_size, int z_size, ptr_f f) {
	for (int x = 0; x < x_size; ++x)
		for (int y = 0; y < y_size; ++y)
			for (int z = 0; z < z_size; ++z) {
				arr_f[z_size * (x * y_size + y) + z] = f(x - FGSHIFT, y -
					FGSHIFT, z - FGSHIFT);
				//printf("%f\n", arr_f[z_size * (x * y_size + y) + z]);
			}
}
```

```
//определение узлов квадратуры на сфере в константной памяти.
//Котрольное вычисление квадратуры:
float sum_check = 0;
void init_vertices(Vertex* vert_dev) {
	Vertex* temp_vert = (Vertex*)malloc(sizeof(Vertex) * VERTCOUNT);
	int i = 0;
	for (int iphi = 0; iphi < 2 * COEF; iphi++) {
		for (int ipsi = 0; ipsi < COEF; ipsi++, i++) {
			float phi = iphi * M_PI / COEF;
			float psi = ipsi * M_PI / COEF;
			temp_vert[i].x = RADIUS * sinf(psi) * cosf(phi);
			temp_vert[i].y = RADIUS * sinf(psi) * sinf(phi);
			temp_vert[i].z = RADIUS * cosf(psi);
		}
	}
	sum_check = check(temp_vert, &func) * M_PI * M_PI / COEF / COEF;
	printf("sum check = %f\n", sum_check); //check(temp_vert, &func) * M_PI * M_PI / COEF / COEF);
	cudaMemcpyToSymbol(vert, temp_vert, sizeof(Vertex) * VERTCOUNT, 0, cudaMemcpyHostToDevice);
	cudaMemcpy(vert_dev, temp_vert, sizeof(Vertex) * VERTCOUNT, cudaMemcpyHostToDevice);
	free(temp_vert);
}

//копирование данных с хоста в текстуру:
void init_texture(float* df_h) {
```

```
	const cudaExtent volumeSize = make_cudaExtent(FGSIZE,
		FGSIZE, FGSIZE);
	cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
	cudaMalloc3DArray(&df_Array, &channelDesc, volumeSize);
	cudaMemcpy3DParms cpyParams = { 0 };
	cpyParams.srcPtr = make_cudaPitchedPtr((void*)df_h,
		volumeSize.width * sizeof(float), volumeSize.width,
		volumeSize.height);
	cpyParams.dstArray = df_Array;
	cpyParams.extent = volumeSize;
	cpyParams.kind = cudaMemcpyHostToDevice;
	cudaMemcpy3D(&cpyParams);
	//конфигурация текстуры, параметры фильтрации
	df_tex.normalized = false;
	df_tex.filterMode = cudaFilterModeLinear;
	df_tex.addressMode[0] = cudaAddressModeClamp;
	df_tex.addressMode[1] = cudaAddressModeClamp;
	df_tex.addressMode[2] = cudaAddressModeClamp;
	//привящка текстуры к CUDA массиву
	cudaBindTextureToArray(df_tex, df_Array, channelDesc);
}

//освобожднение ресурсов:
void release_texture() {
	cudaUnbindTexture(df_tex);
	cudaFreeArray(df_Array);
}
```

```
//функция ядра для вычисление квадратуры:
//(кэширование фильтрованных значений функции в узлах)
__global__ void kernelTexture(float* a) { //взято из лекции
	__shared__ float cache[THREADSPERBLOCK];
	int tid = threadIdx.x + blockIdx.x * blockDim.x;
	int cacheIndex = threadIdx.x;
	float x = vert[tid].x + FGSHIFT + 0.5f;
	float y = vert[tid].y + FGSHIFT + 0.5f;
	float z = vert[tid].z + FGSHIFT + 0.5f;
	cache[cacheIndex] = tex3D(df_tex, z, y, x);
	__syncthreads();
	//суммирование посредством редукции
	for (int s = blockDim.x / 2; s > 0; s >>= 1) {
		if (cacheIndex < s)
			cache[cacheIndex] += cache[cacheIndex + s];
		__syncthreads();
	}
	if (cacheIndex == 0)
		a[blockIdx.x] = cache[0];
}

//трилинейная интерполяция:
__device__ float interpolate1D(float a, float b, float t) {
	return a * (1 - t) + b * t;
}
__device__ float interpolate2D(float v1, float v2, float v3, float v4, float x, float y) {
```

```
        float s = interpolate1D(v1, v2, x);
        float t = interpolate1D(v3, v4, x);
        return interpolate1D(s, t, y);
}
__device__ float interpolate3D(float* arr_f, Vertex v) {
        int gx = int(v.x);
        int gy = int(v.y);
        int gz = int(v.z);
        float tx = v.x - (float)gx;
        float ty = v.y - (float)gy;//float ty = v.z - (float)gz;
        float tz = v.z - (float)gz;

        if (gx + 1 >= FGSIZE || gy + 1 >= FGSIZE || gz + 1 >= FGSIZE)
                return 0.0f;
        float c000 = arr_f[FGSIZE * (gx * FGSIZE + gy) + gz];
        float c001 = arr_f[FGSIZE * ((gx + 1) * FGSIZE + gy) + gz];
        float c010 = arr_f[FGSIZE * (gx * FGSIZE + (gy + 1)) + gz];
        float c011 = arr_f[FGSIZE * ((gx + 1) * FGSIZE + (gy + 1)) + gz];

        float c100 = arr_f[FGSIZE * (gx * FGSIZE + gy) + (gz + 1)];
        float c101 = arr_f[FGSIZE * ((gx + 1) * FGSIZE + gy) + (gz + 1)];
        float c110 = arr_f[FGSIZE * (gx * FGSIZE + (gy + 1)) + (gz + 1)];
        float c111 = arr_f[FGSIZE * ((gx + 1) * FGSIZE + (gy + 1)) + (gz + 1)];

        float e = interpolate2D(c000, c001, c010, c011, tx, ty);
        float f = interpolate2D(c100, c101, c110, c111, tx, ty);
```

```
	return interpolate1D(e, f, tz);
}
__global__ void kernelTrilinear(float* a, float* arr, Vertex* arr_vrt) {
	__shared__ float cache[THREADSPERBLOCK];
	int tid = threadIdx.x + blockIdx.x * blockDim.x;
	int cacheIndex = threadIdx.x;
	Vertex vrt;
	vrt.x = arr_vrt[tid].x + FGSHIFT;
	vrt.y = arr_vrt[tid].y + FGSHIFT;
	vrt.z = arr_vrt[tid].z + FGSHIFT;
	cache[cacheIndex] = interpolate3D(arr, vrt);
	__syncthreads();
	for (int s = blockDim.x / 2; s > 0; s >>= 1) {
		if (cacheIndex < s)
			cache[cacheIndex] += cache[cacheIndex + s];
		__syncthreads();
	}
	if (cacheIndex == 0)
		a[blockIdx.x] = cache[0];
}

//ступенчатая интерполяция:
__device__ float getDistance(Vertex v0, Vertex v) {
	return sqrtf((v0.x - v.x) * (v0.x - v.x) + (v0.y - v.y) * (v0.y - v.y) + (v0.z - v.z) *
(v0.z - v.z));
}
__device__ float interpolateStep(float* arr_f, Vertex v) {
	int gx = int(v.x);
```

```
	int gy = int(v.y);
	int gz = int(v.z);

	if (gx + 1 >= FGSIZE || gy + 1 >= FGSIZE || gz + 1 >= FGSIZE)
		return 0.0f;

	float fgx = float(gx);
	float fgy = float(gy);
	float fgz = float(gz);

	Vertex coords[8] = { {fgx, fgy, fgz},		{fgx + 1, fgy, fgz},
					{fgx, fgy, fgz + 1},	{fgx + 1, fgy, fgz + 1},
					{fgx, fgy + 1, fgz},	{fgx + 1, fgy + 1, fgz},
					{fgx, fgy + 1, fgz + 1}, {fgx + 1, fgy + 1, fgz +
1} };

	float value = arr_f[FGSIZE * (gx * FGSIZE + gy) + gz];
	Vertex vrt;
	vrt.x = coords[8].x;
	vrt.y = coords[8].y;
	vrt.z = coords[8].z;
	float distance = getDistance(vrt, v);
	float tmp = 0;
	for (int i = 1; i < 8; i++) {
		Vertex vrt1;
		vrt1.x = coords[i].x;
		vrt1.y = coords[i].y;
		vrt1.z = coords[i].z;
```

```
            tmp = getDistance(vrt1, v);
            if (tmp < distance) {
                distance = tmp;
                value = arr_f[FGSIZE * (int(coords[i].x) * FGSIZE +
int(coords[i].y)) + int(coords[i].z)];
            }
        }

        return value;
}
__global__ void kernelStepped(float* a, float* arr, Vertex* arr_vrt) {
        __shared__ float cache[THREADSPERBLOCK];
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        int cacheIndex = threadIdx.x;
        Vertex vrt;
        vrt.x = arr_vrt[tid].x + FGSHIFT;
        vrt.y = arr_vrt[tid].y + FGSHIFT;
        vrt.z = arr_vrt[tid].z + FGSHIFT;
        cache[cacheIndex] = interpolateStep(arr, vrt);
        __syncthreads();
        for (int s = blockDim.x / 2; s > 0; s >>= 1) {
            if (cacheIndex < s)
                cache[cacheIndex] += cache[cacheIndex + s];
            __syncthreads();
        }
        if (cacheIndex == 0)
            a[blockIdx.x] = cache[0];
}
```

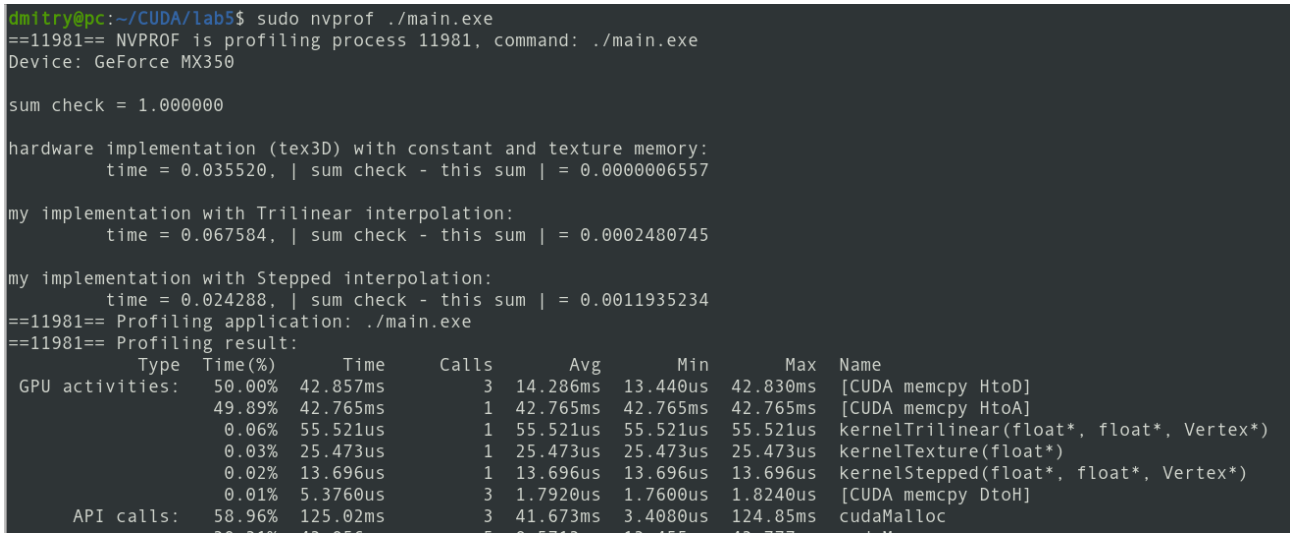
```
int main(void) {
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, 0);
	printf("Device: %s\n\n", deviceProp.name);

	//память, выделяемая на хосте:
	float* arr_ho = (float*)malloc(sizeof(float) * FGSIZE * FGSIZE * FGSIZE);
	float* sum_ho = (float*)malloc(sizeof(float) * BLOCKSPERGRID);

	//память, выделяемая на девайсе:
	float* sum_de, * arr_de;
	cudaMalloc((void**)&sum_de, sizeof(float) * BLOCKSPERGRID);
	cudaMalloc((void**)&arr_de, sizeof(float) * FGSIZE * FGSIZE * FGSIZE);
	Vertex* vert_de;
	cudaMalloc((void**)&vert_de, sizeof(Vertex) * VERTCOUNT);
	init_vertices(vert_de);
	calc_f(arr_ho, FGSIZE, FGSIZE, FGSIZE, &func);
	init_texture(arr_ho);

	cudaMemcpy(arr_de, arr_ho, sizeof(float) * FGSIZE * FGSIZE * FGSIZE,
		cudaMemcpyHostToDevice);

	//cuda-события:
	float elapsedTime;
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
```

```
        cudaEventCreate(&stop);


        for (int test_num = 0; test_num < 3; test_num++) {
                switch (test_num) {
                        case 0:
                                cudaEventRecord(start, 0);
                                kernelTexture <<< BLOCKSPERGRID,
THREADSPERBLOCK >>> (sum_de);
                                cudaEventRecord(stop, 0);
                                break;
                        case 1:
                                cudaEventRecord(start, 0);
                                kernelTrilinear <<< BLOCKSPERGRID,
THREADSPERBLOCK >>> (sum_de, arr_de, vert_de);
                                cudaEventRecord(stop, 0);
                                break;
                        case 2:
                                cudaEventRecord(start, 0);
                                kernelStepped <<< BLOCKSPERGRID,
THREADSPERBLOCK >>> (sum_de, arr_de, vert_de);
                                cudaEventRecord(stop, 0);
                                break;
                        default:
                                break;
                }
                cudaEventSynchronize(stop);
```

```
		cudaMemcpy(sum_ho, sum_de, sizeof(float) * BLOCKSPERGRID,
cudaMemcpyDeviceToHost);
		float s = 0.0f;
		for (int i = 0; i < BLOCKSPERGRID; i++) {
			s += sum_ho[i];
		}
		switch (test_num) {
			case 0:
				printf("\nhardware implementation (tex3D) with constant
and texture memory:\n");
				break;
			case 1:
				printf("\nmy implementation with Trilinear interpolation:\
n");
				break;
			case 2:
				printf("\nmy implementation with Stepped interpolation:\
n");
				break;
		}
		s = s * M_PI * M_PI / COEF / COEF;
		//printf("\t sum = %f\n", s);
		cudaEventElapsedTime(&elapsedTime, start, stop);
		printf("\t time = %f, | sum check - this sum | = %.10f\n",
			elapsedTime, fabs(sum_check - s));
	}

	cudaFree(sum_ho);
```

```
        release_texture();
        free(arr_ho);
        return 0;
}
```

Скриншоты



```
dmitry@pc:~/CUDA/lab5$ sudo nvprof ./main.exe
==11981== NVPROF is profiling process 11981, command: ./main.exe
Device: GeForce MX350

sum check = 1.000000

hardware implementation (tex3D) with constant and texture memory:
        time = 0.035520, | sum check - this sum | = 0.0000006557

my implementation with Trilinear interpolation:
        time = 0.067584, | sum check - this sum | = 0.0002480745

my implementation with Stepped interpolation:
        time = 0.024288, | sum check - this sum | = 0.0011935234
==11981== Profiling application: ./main.exe
==11981== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.00%  42.857ms         3  14.286ms  13.440us  42.830ms  [CUDA memcpy HtoD]
                   49.89%  42.765ms         1  42.765ms  42.765ms  42.765ms  [CUDA memcpy HtoA]
                    0.06%  55.521us         1  55.521us  55.521us  55.521us  kernelTrilinear(float*, float*, Vertex*)
                    0.03%  25.473us         1  25.473us  25.473us  25.473us  kernelTexture(float*)
                    0.02%  13.696us         1  13.696us  13.696us  13.696us  kernelStepped(float*, float*, Vertex*)
                    0.01%  5.3760us         3  1.7920us  1.7600us  1.8240us  [CUDA memcpy DtoH]
      API calls:   58.96%  125.02ms         3  41.673ms  3.4080us  124.85ms  cudaMalloc
```

Рисунок 1 — запуск программы через nvprof

Об используемом GPU

Device name: GeForce MX350

Global memory available on device: 2099904512 (2002 MByte)

Shared memory available per block: 49152

Count of 32-bit registers available per block: 65536

Warp size in threads: 32

Maximum pitch in bytes allowed by memory copies: 2147483647

Maximum number of threads per block: 1024

Maximum size of each dimension of a block[0]: 1024

Maximum size of each dimension of a block[1]: 1024

Maximum size of each dimension of a block[2]: 64

Maximum size of each dimension of a grid[0]: 2147483647

Maximum size of each dimension of a grid[1]: 65535

Maximum size of each dimension of a grid[2]: 65535

Clock frequency in kilohertz: 1468000

totalConstMem: 65536

Major compute capability: 6

Minor compute capability: 1

Number of multiprocessors on device: 5

Count of cores: 640

Id current device: 0

Id nearest device to 1.3: 0

Заключение

Как видно из результатов тестирования, самая меньшую погрешность имеет аппаратная реализация трилинейной интерполяции. Также оно несколько медленее написаной ступенчатой интерполяции и быстрее написанной трилинейной интерполяции.