Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

ЛАБОРАТОРНАЯ РАБОТА №7

По дисциплине: «Программирование графических процессоров»

Выполнили:

Студенты 3 курса группы ИП-211
Оганесян А.С.
Лацук А.Ю.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2025

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 реализовать программно);
- сравнить результаты и время вычислений обоими способами.

Цель: изучить преимущества использования константной и текстурной памяти.

Выполнение работы:

<u>Подход 1:</u> С использованием текстурной и константной памяти

- Данные функции загружаются в 3D текстуру CUDA (через cudaArray), что позволяет использовать аппаратную оптимизацию при обращении к данным.
- Параметры сферы (центр и радиус) передаются через __constant__ память, что обеспечивает быструю трансляцию и широковещательный доступ из всех потоков.
- Ядро CUDA выполняет параметризацию сферы в сферических координатах (θ, φ) и производит выборку значений из текстуры через tex3D(...).
- Использована аппаратная точечная интерполяция (cudaFilterModePoint), которая работает быстрее, чем линейная.
- Время работы измеряется с помощью cudaEvent_t.

```
#include <iostream>
#include <cmath>
#include <vector>
#include <cuda_runtime.h>

__constant__ float d_center[3];
__constant__ float d_radius;
```

```
texture<float, 3, cudaReadModeElementType> tex3DRef;

__global__ void integrateOnSphereKernel(float *partialSums, int thetaSteps, int
phiSteps)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalPoints = thetaSteps * phiSteps;
    if (tid >= totalPoints)
        return;

    int i = tid / phiSteps;
    int j = tid % phiSteps;

    float theta = M_PI * (i + 0.5f) / thetaSteps;
    float phi = 2.0f * M_PI * (j + 0.5f) / phiSteps;

    float x = d_center[0] + d_radius * sinf(theta) * cosf(phi);
    float y = d_center[1] + d_radius * sinf(theta) * sinf(phi);
    float z = d_center[2] + d_radius * cosf(theta);

    float val = tex3D(tex3DRef, x, y, z);

    float dtheta = M_PI / thetaSteps;
    float dphi = 2.0f * M_PI / phiSteps;
    float dS = d_radius * d_radius * sinf(theta) * dtheta * dphi;

    partialSums[tid] = val * dS;
}

float integrateOnSphere(float *h_volume, int nx, int ny, int nz,
                        float x0, float y0, float z0, float R,
                        int thetaSteps, int phiSteps)
{
    cudaExtent volumeSize = make_cudaExtent(nx, ny, nz);
    cudaArray *d_array;
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
    cudaMalloc3DArray(&d_array, &desc, volumeSize);

    cudaMemcpy3DParms copyParams = {0};
    copyParams.srcPtr = make_cudaPitchedPtr(h_volume, nx * sizeof(float), nx, ny);
    copyParams.dstArray = d_array;
    copyParams.extent = volumeSize;
    copyParams.kind = cudaMemcpyHostToDevice;
    cudaMemcpy3D(&copyParams);

    tex3DRef.normalized = true;
    tex3DRef.filterMode = cudaFilterModeLinear;
    tex3DRef.addressMode[0] = cudaAddressModeClamp;
    tex3DRef.addressMode[1] = cudaAddressModeClamp;
    tex3DRef.addressMode[2] = cudaAddressModeClamp;
```

```
    cudaBindTextureToArray(tex3DRef, d_array, desc);

    float h_center[3] = {x0, y0, z0};
    cudaMemcpyToSymbol(d_center, h_center, sizeof(float) * 3);
    cudaMemcpyToSymbol(d_radius, &R, sizeof(float));

    int totalPoints = thetaSteps * phiSteps;
    float *d_partialSums;
    cudaMalloc(&d_partialSums, sizeof(float) * totalPoints);

    int blockSize = 256;
    int gridSize = (totalPoints + blockSize - 1) / blockSize;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    integrateOnSphereKernel<<<gridSize, blockSize>>>(d_partialSums, thetaSteps,
phiSteps);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    std::cout << "GPU kernel execution time: " << milliseconds << " ms" << std::endl;

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    std::vector<float> h_partialSums(totalPoints);
    cudaMemcpy(h_partialSums.data(), d_partialSums, sizeof(float) * totalPoints,
cudaMemcpyDeviceToHost);

    float result = 0.0f;
    for (float val : h_partialSums)
        result += val;

    cudaUnbindTexture(tex3DRef);
    cudaFreeArray(d_array);
    cudaFree(d_partialSums);

    return result;
}

void generateConstantVolume(std::vector<float> &volume, int nx, int ny, int nz, float
value)
{
    volume.resize(nx * ny * nz, value);
}
```

```
int main()
{
    const int nx = 64, ny = 64, nz = 64;

    std::vector<float> volume;
    generateConstantVolume(volume, nx, ny, nz, 1.0f); // f(x, y, z) = 1

    float x0 = nx / 2.0f;
    float y0 = ny / 2.0f;
    float z0 = nz / 2.0f;
    float R = 10.0f;

    int thetaSteps = 180;
    int phiSteps = 360;

    float integral = integrateOnSphere(volume.data(), nx, ny, nz, x0, y0, z0, R,
thetaSteps, phiSteps);

    std::cout << "--- TEXTURE OPTIMIZED ---" << std::endl;
    std::cout << "Computed integral: " << integral << std::endl;
    std::cout << "Expected (4πR²): " << 4.0 * M_PI * R * R << std::endl;

    return 0;
}
```

Листинг 1 – texture.cu

<u>Подход 2:</u> Без использования текстурной и константной памяти

- Используется обычная float* память для хранения 3D данных функции в глобальной памяти устройства.
- Значения функции извлекаются вручную, в зависимости от режима:
    - NEAREST: ближайший сосед по индексу;
    - LINEAR: программная трилинейная интерполяция.
- Для каждой точки на сфере вычисляется значение функции, далее умножается на площадь элементарного участка dS.
- Суммирование значений выполняется на хосте.
- Также используется cudaEvent_t для точного измерения времени выполнения ядра.

```
#include <iostream>
#include <cmath>
#include <vector>
#include <cuda_runtime.h>
```

```
enum InterpMode
{
    NEAREST = 0,
    LINEAR = 1
};

__global__ void integrateOnSphereKernel(
    const float *__restrict__ volume, int nx, int ny, int nz,
    float *partialSums,
    float x0, float y0, float z0, float R,
    float dtheta, float dphi,
    int thetaSteps, int phiSteps,
    InterpMode interpMode)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalPoints = thetaSteps * phiSteps;
    if (tid >= totalPoints)
        return;

    int i = tid / phiSteps;
    int j = tid % phiSteps;

    float theta = M_PI * (i + 0.5f) / thetaSteps;
    float phi = 2.0f * M_PI * (j + 0.5f) / phiSteps;

    float x = x0 + R * sinf(theta) * cosf(phi);
    float y = y0 + R * sinf(theta) * sinf(phi);
    float z = z0 + R * cosf(theta);

    float val = 0.0f;

    if (interpMode == NEAREST)
    {
        int xi = __float2int_rn(x);
        int yi = __float2int_rn(y);
        int zi = __float2int_rn(z);

        if (xi >= 0 && xi < nx &&
            yi >= 0 && yi < ny &&
            zi >= 0 && zi < nz)
        {
            int idx = zi * nx * ny + yi * nx + xi;
            val = volume[idx];
        }
    }
    else if (interpMode == LINEAR)
    {
        int x0i = floorf(x), x1i = x0i + 1;
        int y0i = floorf(y), y1i = y0i + 1;
        int z0i = floorf(z), z1i = z0i + 1;
```

```
        float xd = x - x0i;
        float yd = y - y0i;
        float zd = z - z0i;

        auto at = [&](int xi, int yi, int zi) -> float
        {
            if (xi < 0 || xi >= nx ||
                yi < 0 || yi >= ny ||
                zi < 0 || zi >= nz)
                return 0.0f;
            return volume[zi * nx * ny + yi * nx + xi];
        };

        float c000 = at(x0i, y0i, z0i);
        float c001 = at(x0i, y0i, z1i);
        float c010 = at(x0i, y1i, z0i);
        float c011 = at(x0i, y1i, z1i);
        float c100 = at(x1i, y0i, z0i);
        float c101 = at(x1i, y0i, z1i);
        float c110 = at(x1i, y1i, z0i);
        float c111 = at(x1i, y1i, z1i);

        float c00 = c000 * (1 - xd) + c100 * xd;
        float c01 = c001 * (1 - xd) + c101 * xd;
        float c10 = c010 * (1 - xd) + c110 * xd;
        float c11 = c011 * (1 - xd) + c111 * xd;

        float c0 = c00 * (1 - yd) + c10 * yd;
        float c1 = c01 * (1 - yd) + c11 * yd;

        val = c0 * (1 - zd) + c1 * zd;
    }

    float dS = R * R * sinf(theta) * dtheta * dphi;
    partialSums[tid] = val * dS;
}

float integrateOnSphere(
    const std::vector<float> &h_volume, int nx, int ny, int nz,
    float x0, float y0, float z0, float R,
    int thetaSteps, int phiSteps,
    InterpMode interpMode)
{
    int N = nx * ny * nz;
    float *d_volume;
    cudaMalloc(&d_volume, sizeof(float) * N);
    cudaMemcpy(d_volume, h_volume.data(), sizeof(float) * N, cudaMemcpyHostToDevice);

    int totalPoints = thetaSteps * phiSteps;
```

```
    float *d_partialSums;
    cudaMalloc(&d_partialSums, sizeof(float) * totalPoints);

    float dtheta = M_PI / thetaSteps;
    float dphi = 2.0f * M_PI / phiSteps;

    int blockSize = 256;
    int gridSize = (totalPoints + blockSize - 1) / blockSize;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    integrateOnSphereKernel<<<gridSize, blockSize>>>(
        d_volume, nx, ny, nz,
        d_partialSums,
        x0, y0, z0, R,
        dtheta, dphi,
        thetaSteps, phiSteps,
        interpMode);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    std::cout << "GPU kernel execution time: " << milliseconds << " ms" << std::endl;

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    std::vector<float> h_partialSums(totalPoints);
    cudaMemcpy(h_partialSums.data(), d_partialSums, sizeof(float) * totalPoints,
cudaMemcpyDeviceToHost);

    float result = 0.0f;
    for (float val : h_partialSums)
        result += val;

    cudaFree(d_volume);
    cudaFree(d_partialSums);
    return result;
}

void generateConstantVolume(std::vector<float> &volume, int nx, int ny, int nz, float
value)
{
    volume.resize(nx * ny * nz, value);
}
```

```
int main()
{
    const int nx = 64, ny = 64, nz = 64;
    std::vector<float> volume;
    generateConstantVolume(volume, nx, ny, nz, 1.0f); // f(x,y,z) = 1

    float x0 = nx / 2.0f;
    float y0 = ny / 2.0f;
    float z0 = nz / 2.0f;
    float R = 10.0f;

    int thetaSteps = 180;
    int phiSteps = 360;

    std::cout << "--- NEAREST ---\n";
    float integral_nearest = integrateOnSphere(volume, nx, ny, nz, x0, y0, z0, R,
thetaSteps, phiSteps, NEAREST);
    std::cout << "Computed integral: " << integral_nearest << std::endl;

    std::cout << "--- LINEAR ---\n";
    float integral_linear = integrateOnSphere(volume, nx, ny, nz, x0, y0, z0, R,
thetaSteps, phiSteps, LINEAR);
    std::cout << "Computed integral: " << integral_linear << std::endl;

    std::cout << "Expected (4πR²): " << 4.0 * M_PI * R * R << std::endl;

    return 0;
}
```

Листинг 2 – no_texture.cu

```
albert@DESKTOP-70OAJI4:/mnt/c/Users/User/Documents/GitHub/OS/6_t
erm/7$ ./texture
GPU kernel execution time: 0.099072 ms
--- TEXTURE OPTIMIZED ---
Computed integral: 1256.61
Expected (4πR²): 1256.64

albert@DESKTOP-70OAJI4:/mnt/c/Users/User/Documents/GitHub/OS/6_t
erm/7$  ./no_texture
--- NEAREST ---
GPU kernel execution time: 0.08928 ms
Computed integral: 1256.61
--- LINEAR ---
```

```
GPU kernel execution time: 0.082624 ms
Computed integral: 1256.61
Expected (4πR²): 1256.64
```

Листинг 3 – Результат работы программ

Сравнение результатов:

Подход	**Интерполяция**	**Время выполнения (ms)**	**Результат интеграла**
С текстурной и константной памятью	Point	0.09907	1256.61
Без специальных типов памяти	Nearest	0.08928	1256.61
Без специальных типов памяти	Linear	0.08262	1256.61

Вывод:

В процессе выполнения работы были реализованы и протестированы два подхода к вычислению поверхностного интеграла функции, заданной на трехмерной сетке, по сфере: с использованием текстурной и константной памяти CUDA, а также без использования этих типов памяти с программной реализацией интерполяции.

В первом случае данные функции были размещены в 3D текстуре, что позволило воспользоваться преимуществами кэширования и аппаратной поддержки точечной интерполяции. Параметры сферы (центр и радиус) были переданы в __constant__ память, что обеспечило быстрый и единообразный доступ к ним из всех потоков. Теоретически это должно было привести к увеличению производительности.

Однако экспериментальные результаты показали, что реализация с текстурной и константной памятью оказалась незначительно медленнее, чем реализация без них — как в режиме ступенчатой (NEAREST), так и линейной (LINEAR) интерполяции. Это может быть связано с дополнительными накладными расходами на загрузку данных в cudaArray, а также с тем, что аппаратная реализация доступа к текстуре не всегда выигрывает на современных архитектурах GPU, особенно при регулярных и простых шаблонах чтения данных.

Тем не менее, по точности все реализации показали одинаковый результат, что подтверждает корректность выполненных вычислений.

Таким образом, можно сделать вывод, что при оптимальной ручной реализации интерполяции использование текстурной памяти не всегда оправдано с точки зрения производительности. Тем не менее, при работе с более сложными функциями, резкими градиентами и необходимостью аппаратной интерполяции текстурная память может дать преимущество.