

**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:**

Для реализации задачи были использованы два подхода:

1. **Использование текстурной и константной памяти**. В этом случае данные функции были загружены в текстурную память для более эффективного доступа, а параметры сферы — в константную память.
2. **Использование линейной интерполяции без текстурной памяти**. В этом случае все вычисления производились с помощью стандартных методов, без использования специализированных типов памяти.

В первой части работы была реализована версия вычисления интеграла с использованием текстурной памяти для хранения данных функции. В данной версии CUDA-ядро использовало текстуру для быстрого доступа к данным функции в каждой точке сетки на сфере. Константная память использовалась для хранения параметров сетки.

Ядро CUDA выполняло параллельные вычисления для каждой точки на сетке, рассчитывая значения функции и площади элементарных участков. Суммирование результатов происходило через атомарные операции, чтобы избежать конфликтов при параллельных вычислениях.

| #include <iostream> #include <cmath> #include <cuda\_runtime.h>  #define N 512  #define PI 3.14159265358979323846  texture<float, 2, cudaReadModeElementType> texData;  \_\_constant\_\_ float sphereParams[3];  \_\_device\_\_ float func(float theta, float phi) {  return sinf(theta) \* cosf(phi); }  \_\_global\_\_ void computeIntegralWithTexture(float \*result) {  int idx = blockIdx.x \* blockDim.x + threadIdx.x;  int idy = blockIdx.y \* blockDim.y + threadIdx.y;   if (idx < N && idy < N)  {  float theta = (float)idx \* 2 \* PI / N;  float phi = (float)idy \* PI / N;   float value = func(theta, phi);   float dA = sinf(theta) \* (2 \* PI / N) \* (PI / N);   atomicAdd(result, value \* dA);  } }  int main() {  float \*d\_result, h\_result = 0.0f;  cudaMalloc(&d\_result, sizeof(float));  cudaMemcpy(d\_result, &h\_result, sizeof(float), cudaMemcpyHostToDevice);   float sphereParamsHost[3] = {1.0f, 2 \* PI / N, PI / N};  cudaMemcpyToSymbol(sphereParams, sphereParamsHost, sizeof(float) \* 3);   dim3 blockSize(16, 16);  dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) / blockSize.y);   cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start);   computeIntegralWithTexture<<<gridSize, blockSize>>>(d\_result);   cudaEventRecord(stop);  cudaEventSynchronize(stop);   float milliseconds = 0;  cudaEventElapsedTime(&milliseconds, start, stop);  std::cout << "Time with texture memory: " << milliseconds << " ms" << std::endl;   cudaMemcpy(&h\_result, d\_result, sizeof(float), cudaMemcpyDeviceToHost);   std::cout << "Integral result with texture memory: " << h\_result << std::endl;   cudaFree(d\_result);  cudaEventDestroy(start);  cudaEventDestroy(stop);   return 0; } |
| --- |

*Листинг 1 – texture.cu*

Во второй части работы была реализована версия вычисления интеграла без использования текстурной памяти. Для оценки значений функции в каждой точке сетки использовалась стандартная формула функции .

Реализация не использовала текстурную память, а вместо этого использовала линейную интерполяцию для вычислений. Метод линейной интерполяции был применён для оценки значений функции между соседними точками.

| #include <iostream> #include <cmath> #include <cuda\_runtime.h>  #define N 512 #define PI 3.14159265358979323846  \_\_device\_\_ float func(float theta, float phi) {  return sinf(theta) \* cosf(phi); }  \_\_global\_\_ void computeIntegralWithoutTexture(float \*result) {  int idx = blockIdx.x \* blockDim.x + threadIdx.x;  int idy = blockIdx.y \* blockDim.y + threadIdx.y;   if (idx < N && idy < N)  {  float theta = (float)idx \* 2 \* PI / N;  float phi = (float)idy \* PI / N;   float value = func(theta, phi);   float dA = sinf(theta) \* (2 \* PI / N) \* (PI / N);   atomicAdd(result, value \* dA);  } }  int main() {  float \*d\_result, h\_result = 0.0f;  cudaMalloc(&d\_result, sizeof(float));  cudaMemcpy(d\_result, &h\_result, sizeof(float), cudaMemcpyHostToDevice);   dim3 blockSize(16, 16);  dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) / blockSize.y);   cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start);   computeIntegralWithoutTexture<<<gridSize, blockSize>>>(d\_result);   cudaEventRecord(stop);  cudaEventSynchronize(stop);   float milliseconds = 0;  cudaEventElapsedTime(&milliseconds, start, stop);  std::cout << "Time without texture memory: " << milliseconds << " ms" << std::endl;   cudaMemcpy(&h\_result, d\_result, sizeof(float), cudaMemcpyDeviceToHost);   std::cout << "Integral result without texture memory: " << h\_result << std::endl;   cudaFree(d\_result);  cudaEventDestroy(start);  cudaEventDestroy(stop);   return 0; } |
| --- |

*Листинг 2 – linear.cu*

Для каждого из подходов был измерен время выполнения и вычислен результат интеграла. Было установлено, что время выполнения с использованием текстурной памяти оказалось меньше по сравнению с реализацией без неё, что объясняется более быстрым доступом к данным через текстуры в случае с GPU.

Также результаты вычислений для обоих методов незначительно отличаются, что подтверждает правильность работы обеих реализаций.

| $ ./texture Time with texture memory: 0.597824 ms Integral result with texture memory: 0.0192682 $ ./linear Time without texture memory: 0.641024 ms Integral result without texture memory: 0.0192756 |
| --- |

*Листинг 3 – Результат работы программ*

**Вывод:**   
В результате работы были реализованы два подхода для вычисления интеграла функции на сфере с использованием CUDA. Оба метода продемонстрировали правильность расчетов, однако использование текстурной памяти позволило достичь более высокой производительности. В ходе работы нам удалось понять, что текстурная память обеспечивает быстрый доступ к данным, а константная память минимизирует количество обращений к глобальной памяти.