Выполнила: Белоусова Е., ИП-911

Задача

• 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 использованиятекстурной 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.

Описание работы программы

Для выполнения первого пункта задания объявим несколько символьных констант для числа Пи, количества узлов на сфере, количества ячеек в сетке, радиус сферы, размеры сетки) и макросов. Объявим текстуру и глобальную переменную для использования константной памяти.

Переменная для использования константной памяти есть массив для хранения данных о вершинах. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название. Для размещения данных в константной памяти предусмотрен спецификатор __constant__. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается. Для записи с хоста в константную память используется функция cudaMemcpyToSymbol.

В функции init_vertices() происходит определение узлов квадратуры на сфере в константной памяти. Для этого используются функции для перехода от сферической системы координат к декартовым. Далее проверяется общая сумма и данные копируются с хоста в константную память.

В функции calc_f() происходит дискретизация функция на прямоугольной сетке. Т.к. изначально центр сферы находился у нуле, сферу необходимо сдвинуть в центр сетки.

В функции init_texture() происходит копирование данных с хоста в текстуру и конфигурация текстуры. Необходимо настроить текстуру, то есть организовать хранилище и ссылку на него.

Работа с текстурами в CUDA идет при помощи так называемых текстурных ссылок (*texture reference*).

Такая ссылка задает некоторую область в памяти, из которой будет производиться чтение. Перед чтением необходимо "привязать" (*bind*) текстурную ссылку к соответствующей области выделенной памяти.

Текстурная ссылка фактически является объектом, обладающим набором свойств (атрибутов), такими как размерность, размер, тип хранимых данных и т.п. Некоторые из этих свойств можно изменять, другие же являются неизменяемыми и задаются всего один раз.

Текстурная ссылка (*texture reference*) задается при помощи следующей конструкции:

```
texturer<Type, Dim, ReadMode> texRef;
```

Настроим ее, задав поля структуры (нормализация, способ фильтрации для текстуры, режим приведения).
Для организации хранилища необходимо выбрать область с padding, то есть такую, чтобы строки как бы выравнивались на границе слова, для быстрого доступа. Memory access is most efficient if aligned correctly. Mis-aligned access requires extra memory load. cudaMallocPitch & cudaMemcpy3D takes care of necessary memory padding for memory alignement for 3D arrays.
Свяжем текстурную ссылку с массивом.

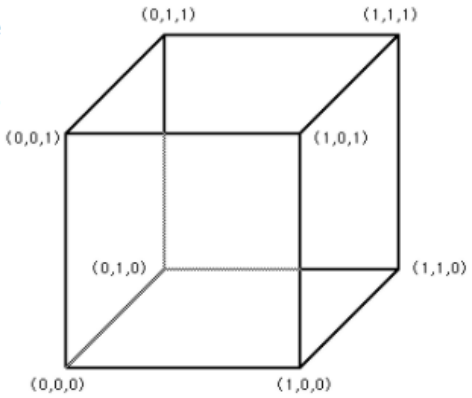
В функции ядра каждая нить работает со своей вершиной. Вычисляем индекс вершины и идентификатор внутри блока. Разделяемая память используется для хранения результатов после чтения данных с текстуры. Далее происходит суммирование посредством редукции.

Программно реализуем ступенчатую и линейную интерполяции.

Трилинейная интерполяция (http://paulbourke.net/miscellaneous/interpolation/ && https://en.wikipedia.org/wiki/Trilinear_interpolation)

Trilinear interpolation is the name given to the process of linearly interpolating points within a box (3D) given values at the vertices of the box. Perhaps its most common application is interpolating within cells of a volumetric dataset.

Consider a unit cube with the lower/left/base vertex at the origin as shown here on the right.
The values at each vertex will be denoted V_{000}, V_{100}, V_{010}, ...etc....V_{111}



The value at position (x,y,z) within the cube will be denoted V_{xyz} and is given by

$$
\begin{aligned}
V_{xyz} = {} & V_{000}\,(1 - x)\,(1 - y)\,(1 - z) + \\
& V_{100}\,x\,(1 - y)\,(1 - z) + \\
& V_{010}\,(1 - x)\,y\,(1 - z) + \\
& V_{001}\,(1 - x)\,(1 - y)\,z + \\
& V_{101}\,x\,(1 - y)\,z + \\
& V_{011}\,(1 - x)\,y\,z + \\
& V_{110}\,x\,y\,(1 - z) + \\
& V_{111}\,x\,y\,z
\end{aligned}
$$

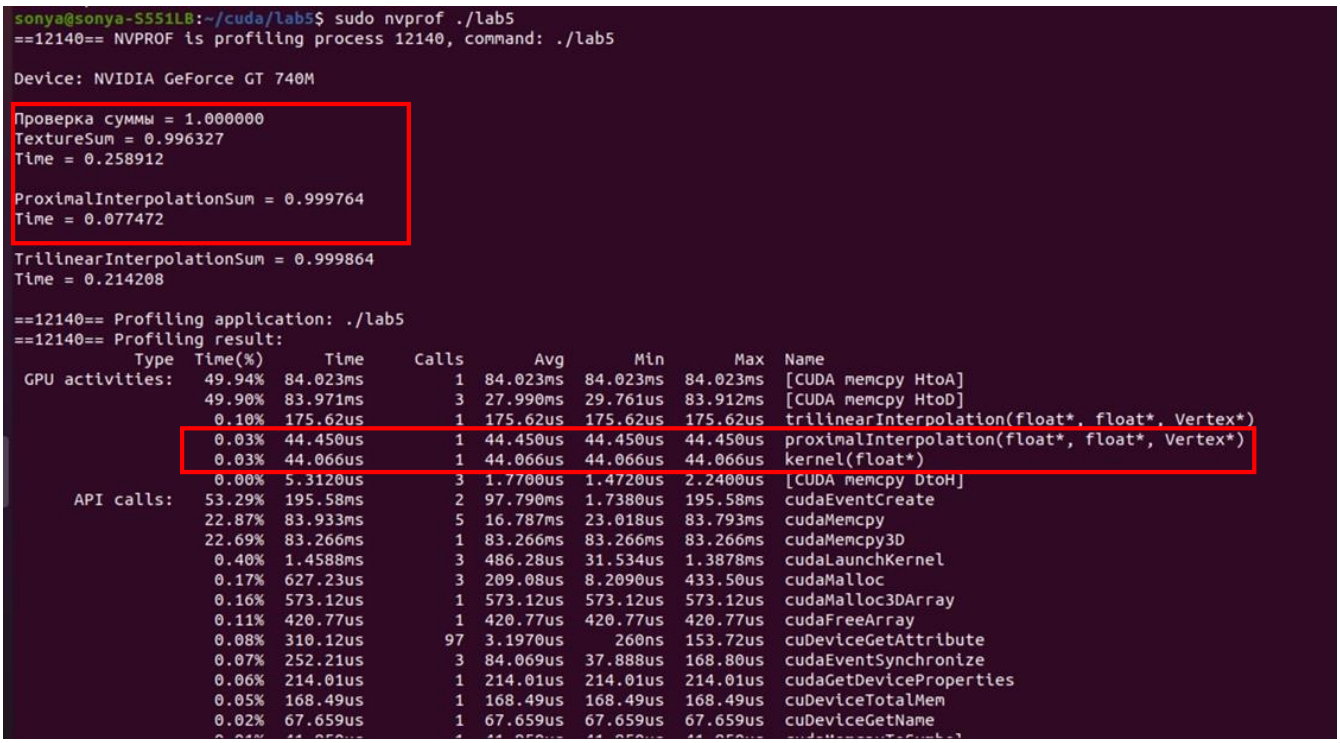
Результаты вычисления интеграла с программной трилинейной интерполяцией:

```
sonya@sonya-S551LB:~/cuda/lab5$ sudo nvprof ./lab5
[sudo] пароль для sonya:
==4708== NVPROF is profiling process 4708, command: ./lab5

Device: NVIDIA GeForce GT 740M

Проверка суммы = 1.000000
TextureSum = 0.999999
ProximalInterpolationSum = 0.999764
TrilinearInterpolationSum = 0.999864
==4708== Profiling application: ./lab5
==4708== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.64%  100.51ms         3  33.502ms  29.826us  100.45ms  [CUDA memcpy HtoD]
                   48.21%  93.833ms         1  93.833ms  93.833ms  93.833ms  [CUDA memcpy HtoA]
                    0.09%  167.92us         1  167.92us  167.92us  167.92us  trilinearInterpolation(float*, float*, Vertex*)
                    0.03%  66.244us         1  66.244us  66.244us  66.244us  kernel(float*)
                    0.02%  45.187us         1  45.187us  45.187us  45.187us  proximalInterpolation(float*, float*, Vertex*)
                    0.00%  5.4080us         3  1.8020us  1.4720us  2.3360us  [CUDA memcpy DtoH]
      API calls:   74.04%  564.86ms         2  282.43ms  1.8040us  564.86ms  cudaEventCreate
                   13.17%  100.48ms         5  20.095ms  23.342us  100.33ms  cudaMemcpy
```

Результаты вычисления интеграла с программной ступенчатой интерполяцией:



```
sonya@sonya-S551LB:~/cuda/lab5$ sudo nvprof ./lab5
==12140== NVPROF is profiling process 12140, command: ./lab5

Device: NVIDIA GeForce GT 740M

Проверка суммы = 1.000000
TextureSum = 0.996327
Time = 0.258912

ProximalInterpolationSum = 0.999764
Time = 0.077472

TrilinearInterpolationSum = 0.999864
Time = 0.214208

==12140== Profiling application: ./lab5
==12140== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.94%  84.023ms         1  84.023ms  84.023ms  84.023ms  [CUDA memcpy HtoA]
                   49.90%  83.971ms         3  27.990ms  29.761us  83.912ms  [CUDA memcpy HtoD]
                    0.10%  175.62us         1  175.62us  175.62us  175.62us  trilinearInterpolation(float*, float*, Vertex*)
                    0.03%  44.450us         1  44.450us  44.450us  44.450us  proximalInterpolation(float*, float*, Vertex*)
                    0.03%  44.066us         1  44.066us  44.066us  44.066us  kernel(float*)
                    0.00%  5.3120us         3  1.7700us  1.4720us  2.2400us  [CUDA memcpy DtoH]
      API calls:   53.29%  195.58ms         2  97.790ms  1.7380us  195.58ms  cudaEventCreate
                   22.87%  83.933ms         5  16.787ms  23.018us  83.793ms  cudaMemcpy
                   22.69%  83.266ms         1  83.266ms  83.266ms  83.266ms  cudaMemcpy3D
                    0.40%  1.4588ms         3  486.28us  31.534us  1.3878ms  cudaLaunchKernel
                    0.17%  627.23us         3  209.08us  8.2090us  433.50us  cudaMalloc
                    0.16%  573.12us         1  573.12us  573.12us  573.12us  cudaMalloc3DArray
                    0.11%  420.77us         1  420.77us  420.77us  420.77us  cudaFreeArray
                    0.08%  310.12us        97  3.1970us     260ns  153.72us  cuDeviceGetAttribute
                    0.07%  252.21us         3  84.069us  37.888us  168.80us  cudaEventSynchronize
                    0.06%  214.01us         1  214.01us  214.01us  214.01us  cudaGetDeviceProperties
                    0.05%  168.49us         1  168.49us  168.49us  168.49us  cuDeviceTotalMem
                    0.02%  67.659us         1  67.659us  67.659us  67.659us  cuDeviceGetName
```

Листинг

```
#include <cuda.h>

#include <cuda_runtime.h>

#include <math.h>

#include <stdio.h>

#include <stdlib.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)

#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);                                                          \
   }                                                                   \
 }
typedef float (*ptr_f)(float, float, float);

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

__constant__ Vertex vert[VERTCOUNT];

texture<float, 3, cudaReadModeElementType> df_tex;
cudaArray *df_Array = 0;

// Функция в разложении по ортонормированному базису в Гильбертовом пространстве
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;
}

// Проверка суммы по функции в декартовых координат вершины
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;
}

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);
}

void init_vertices(Vertex *vertex_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);
    }
  }
  printf("Проверка суммы = %f\n",
         check(temp_vert, &func) * M_PI * M_PI / COEF / COEF);
  // Функция для копирования данных с host'а в текстурную память
  CUDA_CHECK_RETURN(cudaMemcpyToSymbol(vert, temp_vert, sizeof(Vertex) * VERTCOUNT, 0,
cudaMemcpyHostToDevice));
  CUDA_CHECK_RETURN(cudaMemcpy(vertex_dev, temp_vert, sizeof(Vertex) * VERTCOUNT,
cudaMemcpyHostToDevice));
  free(temp_vert);
}
```

```
void init_texture(float *df_h) {
 const cudaExtent volumeSize = make_cudaExtent(FGSIZE, FGSIZE, FGSIZE);
 // Формат дескриптора канала // float = 2
 cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();

 cudaMalloc3DArray(&df_Array, &channelDesc, volumeSize);
 cudaMemcpy3DParms cpyParams = {0};
 // Адрес исходной памяти
 cpyParams.srcPtr =
    make_cudaPitchedPtr((void *)df_h, volumeSize.width * sizeof(float),
                        volumeSize.width, volumeSize.height);
 // df_h - Указатель на выделенную память
 // volumeSize.width * sizeof(float) - шаг выделенной памяти в байтах
 // volumeSize.width - логическая ширина(высота) размещения в элементах
 // Адрес целевой памяти
 cpyParams.dstArray = df_Array;
 // Запрошенный размер экземпляра памяти
 cpyParams.extent = volumeSize;
 // Тип копирования
 cpyParams.kind = cudaMemcpyHostToDevice;
 cudaMemcpy3D(&cpyParams);

 df_tex.normalized =
    false; // Указывает, нормализовано ли чтение текстуры или нет
 df_tex.filterMode =
    cudaFilterModeLinear; // cudaFilterModePoint | cudaFilterModeLinear
 // Режим текстурной адресации для 3-х измерений
 df_tex.addressMode[0] = cudaAddressModeClamp; // Clamp to edge address mode
 df_tex.addressMode[1] = cudaAddressModeClamp;
 df_tex.addressMode[2] = cudaAddressModeClamp;
 // Привязывает массив к текстуре
 cudaBindTextureToArray(df_tex, df_Array, channelDesc);
```

```
}

void release_texture() {
 cudaUnbindTexture(df_tex);
 cudaFreeArray(df_Array);
}

__global__ void kernel(float *a) {
 // Использование разделяемой памяти для кеширования фильтрованных значений
 // функции
 __shared__ float cache[THREADSPERBLOCK];
 // Индекс потока
 int tid = threadIdx.x + blockIdx.x * blockDim.x;
 int cacheIndex = threadIdx.x;
 // Получаем координаты вершин в которых нужно посчитать значение функции
 float x, y, z;
 x = vert[tid].x + FGSHIFT + 0.5f;
 y = vert[tid].y + FGSHIFT + 0.5f;
 z = vert[tid].z + FGSHIFT + 0.5f;

 cache[cacheIndex] = tex3D(df_tex, z, y, x);

 __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];
}
```

```
__device__ float getDistance(Vertex a, Vertex b) {
 return sqrtf((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y) + (a.z - b.z) * (a.z - b.z));
}

__device__ float interpolateStep(float *arr_f, float z, float y, float x) {

 int gx = x;
 int gy = y;
 int gz = z;

 //за пределы сетки
 if (gx + 1 >= FGSIZE || gy + 1 >= FGSIZE || gz + 1 >= FGSIZE)
  return 0.0;

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

 //углы куба
 Vertex angle[8] = {{fgx, fgy, fgz},     {fgx + 1, fgy, fgz},     {fgx, fgy + 1, fgz},     {fgx + 1, fgy + 1, fgz},
                    {fgx, fgy, fgz + 1},  {fgx + 1, fgy, fgz + 1},  {fgx, fgy + 1, fgz + 1},  {fgx + 1, fgy + 1, fgz + 1}};

 // arr_f[z_size * (x * y_size + y) + z]
 float value = arr_f[FGSIZE * (gx * FGSIZE + gy) + gz];
 Vertex vrt {angle[0].x, angle[0].y, angle[0].z};
 Vertex v {x, y, z};
 float distance = getDistance(vrt, v);
 float tmp = 0;

 //минимальное расстояние к точке
 for (int i = 1; i < 8; i++) {
```

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

  return value;
}

__global__ void proximalInterpolation(float *a, float *arr, Vertex *v)
{
  __shared__ float cache[THREADSPERBLOCK];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float x, y, z;
  x = vert[tid].x + FGSHIFT + 0.5f;
  y = vert[tid].y + FGSHIFT + 0.5f;
  z = vert[tid].z + FGSHIFT + 0.5f;

  cache[cacheIndex] = interpolateStep(arr, z, y, x);

  __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];
}

__device__ float interpolate1D(float a, float b, float x) {
  return a * (1 - x) + b * x;
}

__device__ float interpolate2D(float a1, float b1, float a2, float b2, float x, float y) {

  float v1 = interpolate1D(a1, b1, x);
  float v2 = interpolate1D(a2, b2, x);

  return interpolate1D(v1, v2, y);
}

__device__ float interpolate3D(float *arr_f, float z, float y, float x) {

  int gx = x;
  int gy = y;
  int gz = z;
  float tx = x - (float)gx;
  float ty = z - (float)gz;
  float tz = 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 trilinearInterpolation(float *a, float *arr, Vertex *v) {
  __shared__ float cache[THREADSPERBLOCK];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float x = v[tid].x + FGSHIFT;
  float y = v[tid].y + FGSHIFT;
  float z = v[tid].z + FGSHIFT;

  cache[cacheIndex] = interpolate3D(arr, z, y, x);
  __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(int argc, char *argv[]) {
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, 0);
  printf("\nDevice:\t%s\n\n", deviceProp.name);

  Vertex *vert_dev;
  float elapsedTime;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float *arr = (float *)malloc(sizeof(float) * FGSIZE * FGSIZE * FGSIZE);
  float *sum = (float *)malloc(sizeof(float) * BLOCKSPERGRID);
  float *sum_dev, *arr_dev;

  CUDA_CHECK_RETURN(cudaMalloc((void **)&sum_dev, sizeof(float) * BLOCKSPERGRID));
  CUDA_CHECK_RETURN(cudaMalloc((void **)&arr_dev, sizeof(float) * FGSIZE * FGSIZE * FGSIZE));
  CUDA_CHECK_RETURN(cudaMalloc((void **)&vert_dev, sizeof(Vertex) * VERTCOUNT));

  init_vertices(vert_dev);
  calc_f(arr, FGSIZE, FGSIZE, FGSIZE, &func);
  init_texture(arr);

  CUDA_CHECK_RETURN(cudaMemcpy(arr_dev, arr, sizeof(float) * FGSIZE * FGSIZE * FGSIZE,
          cudaMemcpyHostToDevice));
  /* Texture Kernel */
  cudaEventRecord(start, 0);
  kernel<<<BLOCKSPERGRID, THREADSPERBLOCK>>>(sum_dev);
  CUDA_CHECK_RETURN(cudaGetLastError());
  cudaEventRecord(stop, 0);
```

```
 cudaEventSynchronize(stop);


 CUDA_CHECK_RETURN(cudaMemcpy(sum, sum_dev, sizeof(float) * BLOCKSPERGRID,
cudaMemcpyDeviceToHost));


 float s = 0.0f;
 for (int i = 0; i < BLOCKSPERGRID; i++) {
   s += sum[i];
 }
 cudaEventElapsedTime(&elapsedTime, start, stop);
 fprintf(stderr, "TextureSum = %f\n",s * M_PI * M_PI / COEF / COEF);


/* Proximal Interpolation */
 cudaEventRecord(start, 0);
 proximalInterpolation<<<BLOCKSPERGRID, THREADSPERBLOCK>>>(sum_dev, arr_dev, vert_dev);
 CUDA_CHECK_RETURN(cudaGetLastError());
 cudaEventRecord(stop, 0);
 cudaEventSynchronize(stop);


 CUDA_CHECK_RETURN(cudaMemcpy(sum, sum_dev, sizeof(float) * BLOCKSPERGRID,
cudaMemcpyDeviceToHost));


 s = 0.0f;
 for (int i = 0; i < BLOCKSPERGRID; i++) {
   s += sum[i];
 }
 cudaEventElapsedTime(&elapsedTime, start, stop);
 fprintf(stderr, "ProximalInterpolationSum = %f\n", s * M_PI * M_PI / COEF / COEF);


/* Trilinear Kernel */
 cudaEventRecord(start, 0);
 trilinearInterpolation<<<BLOCKSPERGRID, THREADSPERBLOCK>>>(sum_dev, arr_dev, vert_dev);
 CUDA_CHECK_RETURN(cudaGetLastError());
```

```
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  CUDA_CHECK_RETURN(cudaMemcpy(sum, sum_dev, sizeof(float) * BLOCKSPERGRID,
cudaMemcpyDeviceToHost));

  s = 0.0f;
  for (int i = 0; i < BLOCKSPERGRID; i++) {
    s += sum[i];
  }
  cudaEventElapsedTime(&elapsedTime, start, stop);
  fprintf(stderr, "TrilinearInterpolationSum = %f\n", s * M_PI * M_PI / COEF / COEF);
  cudaFree(sum);
  release_texture();
  free(arr);
  return 0;
}
```