**Лабораторная работа №4**

**Основы работы с технологией CUDA. Работа с текстурной памятью.**

**Цифровая обработка сигнала с использованием технологии CUDA/**

**Цель:** изучение основ работы с текстурной памятью на CUDA, реализация цифровой обработки сигналов на CUDA.

При подготовке к лабораторной работе рекомендуется изучить материалы, предоставленные в списке литературы, а также прочие материалы по теме лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект теоретического материала для лабораторной работы, задания и требования к лабораторной работе, а также контрольные вопросы для самопроверки.

1. **Основы работы с текстурной памятью**

Возможности CUDA при работе с текстурной памятью:

* фильтрация текстурных координат;
* билинейная и точечная интерполяция;
* встроенная обработка текстурных координат, в случае, когда значения выходят за допустимые границы;
* обращение по нормализованным или целочисленным координатам;
* возвращение нормализованных значений;
* кеширование данных.

Специфика текстурной памяти:

* быстрая;
* кешируемая в 2-х измерениях;
* только для чтения.

Текстура может быть расположена в:

* линейной памяти на GPU;
* в текстурном массиве (cudaArray).

Шаблон для задания текстурной ссылки имеет следующий вид:

texture<Type, Dim, ReadMode>,

где *Type* – тип элемента (например, float3); *Dim* – размерность текстуры (1, 2 или 3); *ReadMode* – нужна ли нормализация прочитанных значений: cudaReadModeElementType, cudaReadModeNormalizedFloat**.**

Чтение текстур из ядра выполняется с помощью следующих функций:

|  |
| --- |
| #include “cuda\_runtime.h”  template<class T, enum cudaTextureReadMode readMode>  T tex1D ( texture<T, 1, readMode> texRef, float x );  template<class T, enum cudaTextureReadMode readMode>  T tex2D ( texture<T, 2, readMode> texRef, float x, float y );  template<class T, enum cudaTextureReadMode readMode>  T tex3D ( texture<T, 3, readMode> texRef, float x, float y, float z ); |

Пример чтения пикселя в текстуре приведен ниже. Для точного попадания в центр пиксела необходимо добавлять смещение, равное половине пиксела.

|  |
| --- |
| texture<uchar4, 2, cudaReadModeElementType> texName;  uchar4 a = tex2D(texName, texcoord.x + 0.5f, texcoord.y + 0.5f); |

1. **Основы работы с текстурной ссылкой на CUDA**

Доступ к текстурным ссылкам можно получить, зная имя шаблона, с использованием следующих функций:

|  |
| --- |
| const textureReference \* pTexRef = NULL;  cudaGetTExtureReference(&pTexRef, ‘texName’); |

Связать текстурную ссылку и линейную память позволяют следующие функции:

|  |
| --- |
| cudaError\_t cudaBindTexture( size\_t offset,  const struct texture<T, dim, readMode> & tex,  const void \* dev\_ptr, size\_t size)  cudaError\_t cudaBindTexture2D( size\_t offset,  const struct texture<T, dim, readMode> & tex,  const void \* dev\_ptr,  const struct cudaChannelFormatDesc \*desc,  size\_t width, size\_t height,  size\_t pitch\_in\_bytes) |

Фрагмент программного кода, реализующий работу с линейной памятью с использованием текстурной ссылки, приведен на рисунке 4.1.

|  |
| --- |
| texture<float, 1, cudaReadModeElementType> texRef;  \_\_global\_\_ void kernel ( float \* data )  {  int idx = blockIdx.x \* blockDim.x + threadIdx.x;  data [idx] = tex1D( texRef, idx );  }  float \* a, \* aDev;  cudaMalloc ( (void \*\*) &aDev, numBytes );  cudaMemcpy ( aDev, a, numBytes, cudaMemcpyHostToDevice );  cudaBindTexture ( NULL, &texRef, aDev, &texRef.channelDesc, numBytes );  . . . .  kernel<<<blocks, threads>>> ( bDev );  . . . .  cudaThreadSynchronize ();  cudaUnbindTexture( &texRef );  cudaFree( aDev ); |

Рисунок 4.1 – Фрагмент программного кода, использующий текстурную ссылку для передачи данных в ядро

Используя текстуры, рассмотрим применение CUDA в обработке цифровых сигналов. Рассмотрим фильтр негатив. Негатив – фильтр, который вычисляет для заданного цвета дополнение его до белого:

R=1-I.

Программный код, реализующий фильтр негатив приведен на рисунке 4.2.

|  |
| --- |
| # include <time.h>  # include <stdlib.h>  # include <stdio.h>  # include <string.h>  # include <cuda.h>  # include <ctime>  #include <cuda\_runtime.h>  #include "../../../common/inc/helper\_image.h"  texture<unsigned char, 2, cudaReadModeElementType> g\_Texture;  unsigned int width=512 , height =512;  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // функция ядра  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \_\_global\_\_ void negative\_kernel(unsigned char \* dest, int width, int height)  {  int tidx = threadIdx.x + blockIdx.x \* blockDim.x;  int tidy = threadIdx.y + blockIdx.y \* blockDim.y;  // проверка, что текущие индексы не выходят за границы изображения  if (tidx < width && tidy < height)  {  unsigned char c = tex2D(g\_Texture, tidx + 0.5f, tidy + 0.5f );  // преобразование для негатива  //unsigned char r = 255 - c;  dest[tidx + tidy \* width] = 255 - c; //r;  }  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // загрузка и сохранение изображения  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  void loadImage(char \*file, unsigned char\*\* pixels, unsigned int \* width, unsigned int \* height)  {  size\_t file\_length = strlen(file);  if (!strcmp(&file[file\_length - 3], "pgm"))  {  if (sdkLoadPGM<unsigned char>(file, pixels, width, height) != true)  {  printf("Failed to load PGM image file: %s\n", file);  exit(EXIT\_FAILURE);  }  }  return;  }  void saveImage(char \*file, unsigned char\* pixels, unsigned int width, unsigned int height)  {  size\_t file\_length = strlen(file);  if (!strcmp(&file[file\_length - 3], "pgm"))  {  sdkSavePGM(file, pixels, width, height);  }  return;  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // main  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  int main ( int argc , char \*\* argv )  {  /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* setup work \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*/  unsigned char \* d\_result\_pixels ;  unsigned char \* h\_result\_pixels ;  unsigned char \* h\_pixels = NULL ;  unsigned char \* d\_pixels = NULL ;  char \* src\_path = "lena.pgm";  char \* d\_result\_path = "lena\_d.pgm";  loadImage(src\_path, &h\_pixels, &width, &height);  int image\_size = sizeof ( unsigned char ) \* width \* height ;  h\_resultPixels = ( unsigned char \*) malloc (image\_size );  cudaMalloc (( void \*\*)& d\_pixels , image\_size );  cudaMalloc (( void \*\*)& d\_result\_pixels , image\_size);  cudaMemcpy ( d\_pixels , h\_pixels , image\_size , cudaMemcpyHostToDevice);  int n = 16;  dim3 block(n, n);  dim3 grid(width / n, height / n);  cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar1>();  size\_t offset = 0;  cudaError\_t error = cudaBindTexture2D ( 0, &g\_Texture, d\_pixels, &desc, width, height, width \* sizeof(unsigned char) );  if (cudaSuccess != error){  printf("ERROR: Failed to bind texture.\n");  exit(-1);  }  else{  printf("Texture was successfully binded\n");  }  /\* CUDA method \*/  negative\_kernel <<< grid, block >>>(d\_result\_pixels, width, height);  cudaMemcpy ( h\_result\_pixels , d\_result\_pixels , image\_size ,cudaMemcpyDeviceToHost );  saveImage(d\_result\_path, h\_result\_pixels, width, height);  cudaUnbindTexture ( &g\_Texture );  } |

Рисунок 4.2 – Программный код, реализующий фильтр негатив

Результаты работы программы, реализующей фильтр негатив (рисунок 4.2) представлен на рисунке 4.3.

|  |  |
| --- | --- |
| а) исходное изображение | б) изображение после применения фильтра негатив |

Рисунок 4.3 – Результаты работы фильтра негатив

1. **Основы работы с текстурным массивом на CUDA**

Основные функции для работы с текстурным массивом представлены ниже:

1. выделение массива памяти:

|  |
| --- |
| cudaError\_t cudaMallocArray(struct cudaArray \*\*arrayPtr,  const struct cudaChannelFormatDesc \*desc,size\_t width, size\_t height),  cudaError\_t cudaFreeArray(struct cudaArray \*array); |

1. привязка массива памяти к текстурной ссылки:

|  |
| --- |
| cudaError\_t cudaBindTextureToArray(const struct textureReference  \*texref, const struct cudaArray \* array,  const struct cudaChannelFormatDesc \*desc),  cudaError\_t cudaBindTextureToArray(const struct  texture<T, dim, readMode> & tex, const struct cudaArray \* array), |

Фрагмент программного кода для работы с текстурным массивом приведен на рисунке 4.4.

|  |
| --- |
| texture<float, 1, cudaReadModeElementType> texRef;  cudaArray \* array;  cudaChannelFormatDesc chDesc = cudaCreateChannelDesc ( 32, 0, 0, 0,  cudaChannelFormatKindFloat );  cudaMallocArray ( &array, &chDesc, width, height );  cudaMemcpyToArray ( array, wOffs, hOffs, src, numBytes,  cudaMemcpyHostToDevice );  cudaBindTextureToArray ( &texRef, array, &chDesc );  . . . . . . . . .  cudaUnbindTexture ( &texRef );  cudaFreeArray( array ); |

Рисунок 4.4 – Фрагмент программного кода, демонстрирующий работу с CudaArray

1. **Основные понятия цифровой обработки сигнала**

**Свертка**. Даны две вещественные функции f(x) и g(x), интегрируемые на R, то свертка представляет собой функцию вида:

.

В дискретном виде: .

Цифровая дельта-функция имеет вид:



Ядро свертки (импульсная характеристика фильтра) h(n) (рисунок 4.5) - выходной сигнал [динамической системы](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) как реакция на входной сигнал в виде [дельта-функции](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D1%8C%D1%82%D0%B0-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F).

Свертка данной функции с ядром h – это линейная комбинация откликов системы на входные значения f[i], которая представлена на рисунке 4.6.

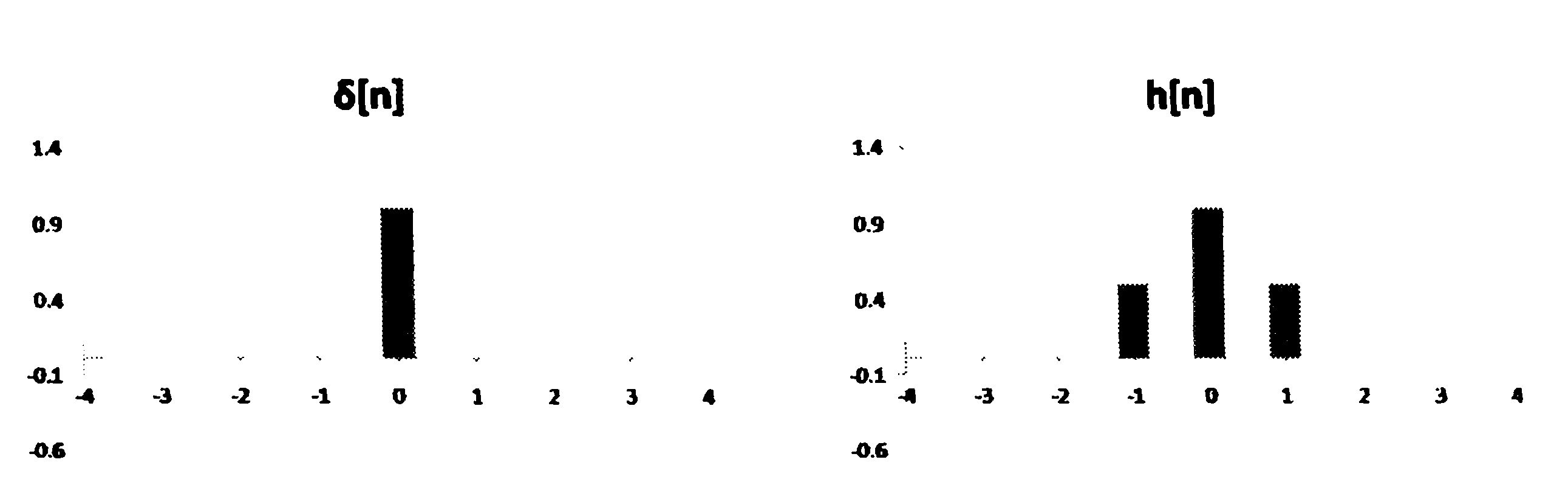


Рисунок 4.5 – Ядро свертки (импульсная характеристика фильтра) h(n)(справа) и дельта-функция (слева)

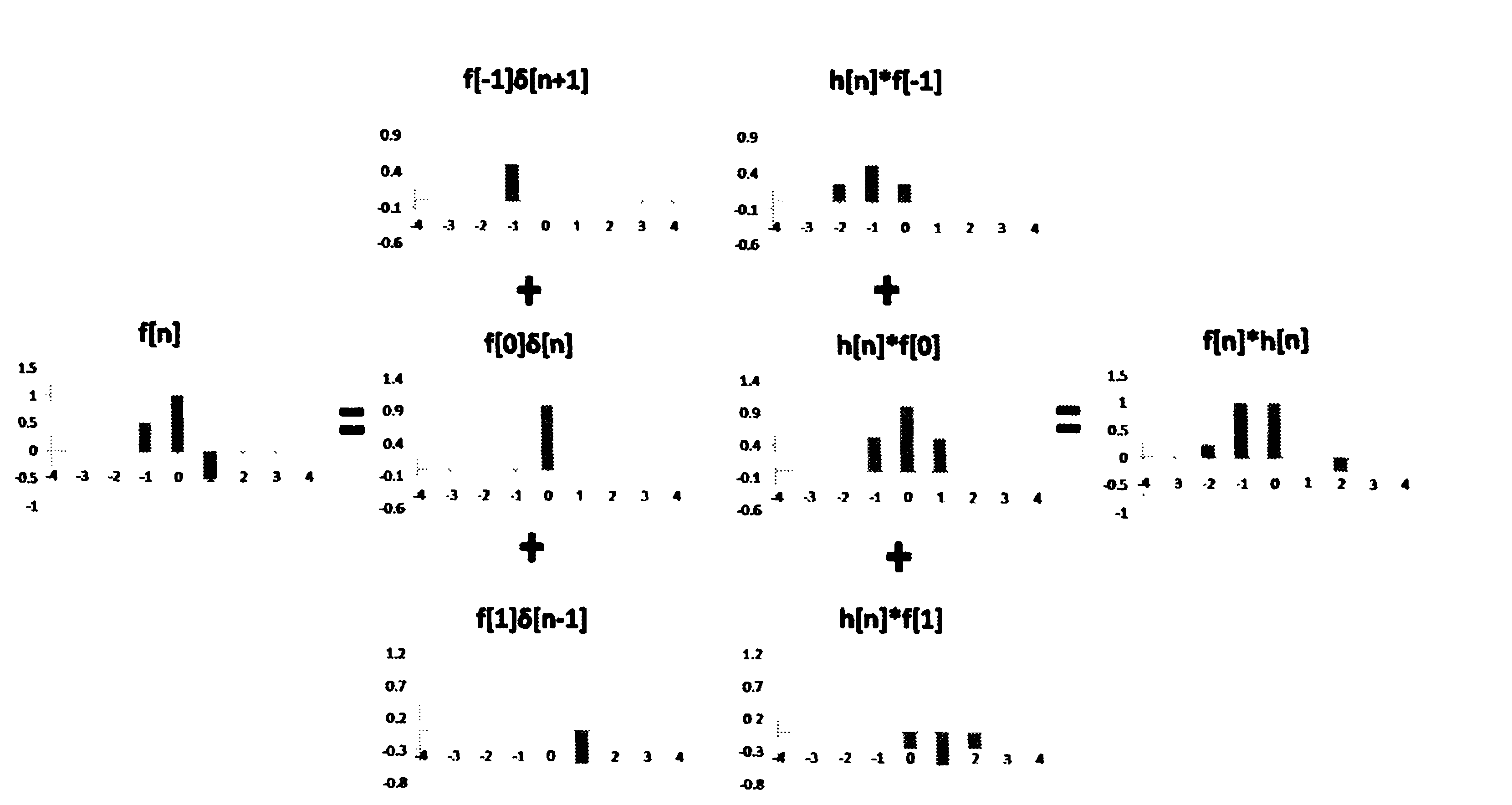


Рисунок 4.6 – Свертка функции f c ядром h

Фильтрация, при которой сигнал усредняется в некоторой окрестности радиуса R с равными весами (см. рисунок 4.7), называется **box blur** (размытие коробкой).

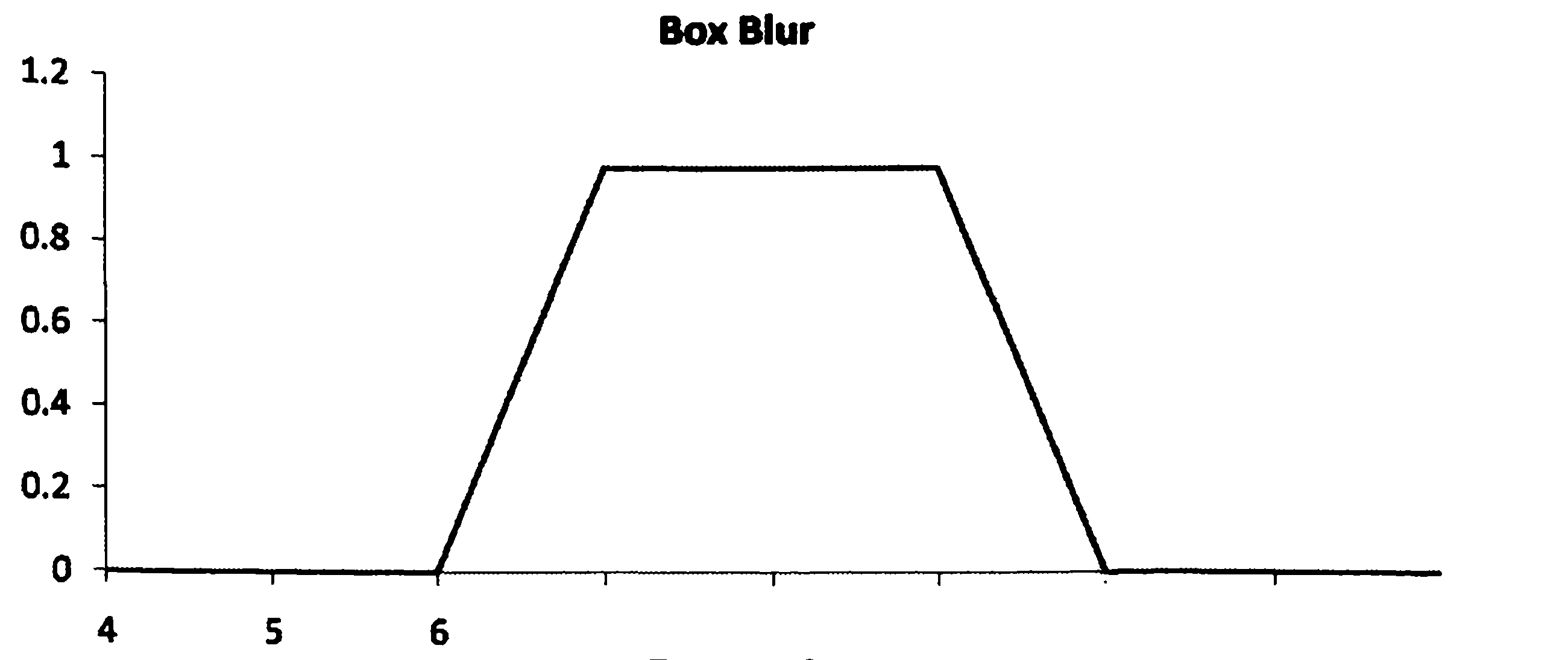


Рисунок 4.7 – Ядро фильтра Box Blur

Ядро на CUDA, которое производит такое размытие, приведено на рисунке 4.8.

|  |
| --- |
| texture<unsigned char, 2, cudaReadModeElementType> g\_BoxBlur;  \_\_global\_\_ void BoxBlur\_kernel(uchar\* pDst, uint radius, uint32 w, uint32 h)  {  int tidx = threadIdx.x + blockIdx.x \* blockDim.x;  int tidy = threadIdx.y + blockIdx.y \* blockDim.y;  // проверка, что текущие индексы не выходят за границы изображения  if ( tidx < w && tidy < h)  {  unsigned char r = 0;  for(int ir = -radius; ir <= radius; ir++ )  for(int ic = -radius; ic <= radius; ic++ )  {  r += tex2D(g\_BoxBlur, tidx + 0.5f+ic, tidy + 0.5f+ir);  }  //нормализация полученных результатов  r /= ((2\*radius+1)\*(2\*radius+1));  pDst[tidx+tidy\*w] = r;  }  } |

Рисунок 4.8 – Ядро, реализующее фильтр Box Blur

**Гауссово размытие** – это фильтрация, при которой сигнал усредняется в некоторой окрестности радиуса R с весами, вычисляемыми по следующим формулам:

а) одномерный случай:

;

б) двумерный случай:

.

Ядро на CUDA, которое реализует гауссово размытие, приведено на рисунке 4.9.

|  |
| --- |
| texture<unsigned char, 2, cudaReadModeElementType> g\_Gaussian;  \_\_global\_\_ void GaussianK (uchar\* pDst, uint radius, float sigma\_sq, uint32 w, uint32 h)  {  int tidx = threadIdx.x + blockIdx.x \* blockDim.x;  int tidy = threadIdx.y + blockIdx.y \* blockDim.y;  // проверка, что текущие индексы не выходят за границы изображения  if ( tidx < w && tidy < h)  {  float r = 0;  float weight\_sum = 0.0f;  float weight = 0.0f;  for(int ic = -radius; ic <= radius; ic++ )  {  weight = exp(-(ic\*ic)/ sigma\_sq);  r += tex2D(g\_ Gaussian, tidx + 0.5f+ic, tidy + 0.5f)\* weight;  weight\_sum += weight;  }  //нормализация полученных результатов  r /= weight\_sum;  pDst[tidx+tidy\*w] = (int)r;  } |

Рисунок 4.9 – Ядро, реализующее фильтр Гауссово размытие

**Билинейная фильтрация.** Даны два значения f(k) и f(k+1), то промежуточное значение можно приблизить с помощью линейной функции вида:

.

В двумерном случае:

.

Ядро, выполняющее такое преобразование, приведено на рисунке 4.10. Его особенность заключается в том, что выборки производятся по столбцам, а запись – по строкам, поэтому сначала изображение растягивается по высоте и поворачивается на 90 градусов (рисунок 4.11 б)). Если подать на вход данное изображение еще раз, то оно промасштабируется по ширине и повернется на 90 градусов (рисунок 4.11в)).

|  |
| --- |
| texture<unsigned char, 2, cudaReadModeElementType> g\_Texture;  \_\_global\_\_ void Bilinear(unsigned char \* dest, float factor, unsigned int w, unsigned int h)  {  int tidx = threadIdx.x + blockIdx.x \* blockDim.x;  int tidy = threadIdx.y + blockIdx.y \* blockDim.y;  // проверка, что текущие индексы не выходят за границы изображения  if ( tidx < w && tidy < h){  float center = tidx/factor;  unsigned int start = (unsigned int)center;  unsigned int stop = start + 1.0f;  float t = center - start;  unsigned char a = tex2D(g\_Texture, tidy + 0.5f, start + 0.5f);  unsigned char b = tex2D(g\_Texture, tidy + 0.5f, stop + 0.5f);  float linear = lerp(a, b, t);  dest[tidx+tidy\*w] = (int)(linear );  }  } |

Рисунок 4.10 – Ядро, реализующее билинейную фильтрацию (lerp – функция для расчета промежуточного значения)

|  |  |  |
| --- | --- | --- |
| а) | б) | в) |

Рисунок 4.11 – Результаты работы билинейного фильтра: а) исходное изображение; б) изображение промасштабировано по высоте и повернуто на 90 градусов; в) изображение промасштабировано по ширине и повернуто на 90 градусов.

**Лабораторные задания** (№ варианта = (№ студента в списке)%2)

**Задание 1**: изучить основы работы с текстурной ссылкой на CUDA. Написать программу на Cи, реализующую цифровой фильтр с использованием текстурной ссылки и CUDA runtime API, в соответствии с вариантом задания. Измерить время работы программы. Написать программу для верификации результатов. **Результаты занести в отчёт.**

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| **0** | Дано двухмерное изображение. Написать программу, реализующую фильтр Box Blur, применительно к исходному изображению. Радиус фильтра – параметр программы. |
| **1** | Дано двухмерное изображение. Написать программу, реализующую фильтр Gaussian Blur, применительно к исходному изображению. Радиус фильтра – параметр программы. |

**Задание 2**: изучить основы работы с текстурным массивом на CUDA. Написать программу на Cи, реализующую цифровой билинейный фильтр для масштабирования изображения с использованием текстурного массива и CUDA runtime API.

**Задание 3**: изучить основы работы CUDA Occupancy calculator. Получить оптимальные теоретические параметры программной модели CUDA для используемого GPU.

**Контрольные вопросы**

1. Где расположена текстурная память?
2. Основные особенности текстурной памяти?
3. Где может располагаться текстура?
4. Основные функции работы с текстурным массивом?
5. Что такое текстурная ссылка?
6. Основные функции работы с текстурными ссылками?
7. Какова основная цель использования текстурной памяти при выполнении задач цифровой обработки сигналов?

**Требования к сдаче работы**

1. При домашней подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчёт результаты выполнения.
2. Продемонстрировать выполнение лабораторных заданий.
3. Ответить на контрольные вопросы.
4. Показать преподавателю отчет.

**Литература**

1. <http://www.nvidia.ru/object/cuda-parallel-computing-ru.html>
2. А.В. Боресков, А.А. Харламов. Основы работы с технологией Cuda. – М: ДМК Пресс, 2010. – 232 с.
3. А. В. Боресков и др. Предисл.: В. А. Садовничий. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие. Издательство Московского университета, 2012. – 336 с.