Лабораторная работа №6

Работа с текстурной памятью

Цель работы: получить начальные практические навыки работы с текстурной памятью; выполнить цифровую обработку изображения с использованием различных фильтров и функций.

1. Краткие теоретические сведения

CUDA поддерживает работу с текстурами, предоставляя при этом полный набор функциональности, доступной из графических API. 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:

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

- Параметр ***Type*** обозначает тип данных, возвращаемых при чтении из текстуры. В качестве Type можно использовать базовые целочисленные типы, float, а также все их 1/2/3/4-мерные вектора.
- Параметр ***Dim*** задает размерность текстуры и принимает значения от 1 до 3 включительно.
- Параметр ***ReadMode*** определяет нужно ли производить так называемую нормализацию прочитанных из текстуры значений, в тех случаях, когда Type является целочисленным типом (или целочисленным вектором). Нормализацией целочисленных значений называется их отображение в floating-point-значения из отрезка [-1,1] для знаковых типов и из [0,1] для беззнаковых. Фактически нормализация значений – это всего лишь деление на максимально возможное значение данного типа (после преобразования во float). В качестве значения параметра ***ReadMode*** выступает одна из следующих констант: ***cudaReadModeNormalizedFloat*** (для случая, когда необходимо произвести нормализацию) и ***cudaReadModeElementType*** (когда никакой нормализации проводить не нужно).

К изменяемым атрибутам текстур относятся также какой тип текстурных координат используется (нормализованные или нет), способ адресации и тип используемой фильтрации.

В качестве памяти для текстуры можно использовать любую область как линейной памяти (linear memory), так и так называемый CUDA-массив (CUDA array). При этом выбор в качестве памяти для текстуры линейной памяти накладывает следующие ограничения на текстуру:

- текстура в линейной памяти может быть только одномерной;
- текстуры в линейной памяти не поддерживают фильтрацию;
- адресация к текстурам в линейной памяти может происходить только через ненормализованные текстурные координаты (т.е. координата является целым числом от 0 до N-1, где N - размер текстуры);
- не поддерживаются режимы адресации (wrap) - если происходит обращение за пределами текстуры, то возвращается ноль.

Для чтения из текстур, размещенных в линейной памяти, используется функция **tex1Dfetch**.

```
template <class Type>
Type tex1Dfetch ( texture<Type, 1, cudaReadModeElementType> texRef, int x );
```

Для чтения из текстур, размещенных в CUDA-массивах (в ядре) используются функции tex1D(), tex2D(), tex3D(), которые принимают текстурную ссылку и одну, две или три координаты:

```
template <class Type, enum cudaTextureReadMode readMode>
Type tex1D ( texture<Type, 1, readMode> texRef, float x);

template <class Type, enum cudaTextureReadModereadMode>
Type tex2D ( texture<Type, 2, readMode> texRef, float x, float y );

template <class Type, enum cudaTextureReadModereadMode>
Type tex3D ( texture<Type, 3, readMode> texRef, float x, float y, float z);
```

В случае если режим обращения к текстуре выбран ненормализованный, стоит учитывать, что для точного попадания в центр пикселя необходимо добавлять смещение, равное половине пикселя (то есть равное 0.5f или 0.5f/ ширина 0.5f/ высота при нормализованных координатах).

```
uchar4 a = tex2D(texName, texcoord.x + 0.5f, texcoord.y + 0.5f);
```

Текстурные ссылки можно получить, зная имя текстурного шаблона, с помощью следующей функции:

```
const textureReference* pTexRef = NULL;
cudaGetTextureReference(&pTexRef, 'texName');
```

Кроме cudaArray, "привязать" к текстурной ссылке можно и обычную линейную память. Пример связывания текстурной ссылки и линейной памяти:

```
cudaError_t cudaBindTexture (size t * offset, const struct texture<T, dim, readMode> & tex,
const void * dev_ptr, size_t size)

cudaError_t cudaBindTexture2D ( sizet * 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)
```

Основным отличительным качеством текстуры является возможность кеширования данных в двухмерном измерении. Если текстура размещена в линейной памяти, то для выделения и освобождения этой памяти используются функции ***cudaMalloc*** и ***cudaFree***.

Выделение и освобождения CUDA-массивов производится при помощи функций ***cudaMallocArray*** и ***cudaFreeArray***. При выделении памяти как CUDA-массива в функцию ***cudaMallocArray*** необходимо кроме размера передать ссылку на структуру ***cudaChannelFormatDesc***, используемую для описания структуры текстуры.

Основные функции для работы с блок-линейной памятью:

```
cudaError_t cudaMallocArray ( struct cudaArray **arrayPtr,
const struct cudaChannelFormatDesc *desc, size_t width, size_t height )

cudaError_t cudaFreeArray ( struct cudaArray *array )
```

Данные функции выделяют участок блок-линейной памяти, возвращая в ***arrayPtr*** указатель на контейнер ***cudaArray***. Доступ к нему происходит из ядра через специальные текстурные ссылки (texture reference), которые в графических API назывались сэмплерами (***sampler***). Суть такой двухуровневой абстракции заключается в том, чтобы отделить непосредственно сами данные

(***cudaArray***) и способ их хранения от интерфейса доступа к ним (*texture reference*). Тем самым интерфейс доступа может обладать вышеперечисленными состояниями (размерность массива, режим фильтрации, режим адресации, возвращаемое значение и т. д.), которые будет учитывать текстурный блок.

```
cudaChannelFormatDesc chDesc = cudaCreateChannelDesc<float>(); cudaArray * array;

cudaMallocArray ( &array, &chDesc, width, height );

cudaMemcpy2DToArray ( array, 0, 0, devPtr, pitch, width * sizeof ( float),

height, cudaMemcpyDeviceToDevice );

…….

cudaFreeArray  ( array );
```

В основе типа texture лежит следующая структура:

```
struct textureReference

{
int normalized;
enum cudaTextureFilterMode filterMode;
enum cudaTextureAddressMode addressMode [3];
struct cudaChannelFormatDesc channelDesc;

};


struct cudaChannelFormatDesc
{
int x, y, z, w;
enum cudaChannelFormatKind f;
};
```

Поля структуры и их назначение представлены в таблице 1.

Таблица 1. Поля структуры и их назначение

Поле	**Назначение**
normalized	Задает происходит ли обращение к данной текстуре с использованием нормализованных текстурных координат (если значение этого поля отлично от нуля) или нет.
filterMode	Задает способ фильтрации для текстуры. Допустимыми значениями являются ***cudaFilterModePoint*** и ***cudaFilterModeLinear***. Линейная фильтрация поддерживается только для текстур, возвращающих *floating-point*-значения.
addressMode	Задает режим приведения для каждой из компонент текстурных координат. Возможными значениями являются ***cudaAddressModeClamp*** и ***cudaAddressModeWrap***.
channelDesc	Задает формат значения, возвращаемого при чтении из текстуры. Поля x, y, z и w содержат число бит на соответствующую компоненту, а f задает тип *channelDesc* данных для этих компонент и принимает одно из следующих значений – ***cudaChannelFormatKindSigned***, ***cudaChannelFormatKindUnsigned*** и ***cudaChannelFormatKindFloat***.

Поля ***normalized***, ***addressMode*** и ***filterMode*** могут быть изменены прямо во время выполнения программы. Все они применимы только к текстурам, расположенным в CUDA-массивах. Для получения ***cudaChannelFormatDesc*** по типу T служит следующая функция:

```
template <class T>
struct cudaChannelFormatDesc cudaCreateChannelDesc <T> ();
```

Прежде, чем ядро сможет читать из текстуры, соответствующая текстурная ссылка должна быть "привязана" (*bind*) к выделенной памяти (в зависимости от того к какому именно типу памяти идет привязка). По окончании работы, необходимо вызвать функцию ***cudaUnbindTexture***.

```
cudaError_t cudaBindTexture ( size_t * offset, const struct textureReference * texRef,
const void * devPtr, struct cudaChannelFormatDesc * desc, size_t size = UINT_MAX const X );

cudaError_t cudaBindTextureToArray ( const struct textureReference * texRef,
const struct cudaArray * array,    const struct cudaChannelFormatDesc * desc );
```

или

```
 cudaError_t cudaBindTextureToArray ( const struct textureReference *texref,
const struct cudaArray *array,const struct cudaChannelFormatDesc *desc);
```

Этот вызов низкоуровневый и требует ручного задания переменной типа ***textureReference*** и дескриптора канала:

```
textureReference texref;
texref.addressMode[0] = cudaAddressModewrap;
texref.addressMode[1] = cudaAddressModewrap;
texref.addressMode[2] = cudaAddressModeWrap;
texref.channelDesc = cudaCreateChannelDesc<uchar4>();
texref.filterMode = cudaFilterModeLinear;
texref.normalized = cudaReadModeElementType;

cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar4>();
```

В то время, как вызов

```
cudaError_t cudaBindTextureToArray ( const struct texture <T, dim, readMode>
& tex, const struct cudaArray *array);
```

использует шаблоны для задания текстурных ссылок и наследует дескриптор канала от переданного *cudaArray*:

```
texture<uchar4, 2, cudaReadModeElementType> texName;
cudaError_t cudaUnbindTexture ( const struct textureReference * texRef );
```

Для копирования памяти между CUDA-массивами, линейной памятью и памятью CPU используются следующие функции:

```
cudaError_t cudaMemcpyFromArray ( void * dst, const struct cudaArray * srcArray,
size_t srcX, size_t srcY, size_t count, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpy2DFromArray  ( void * dst, size_t dpitch, const struct cudaArray * srcArray,
size_t srcX, size_t srcY, size_t width,  size_t height, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyToArray ( struct cudaArray * dstArray, size_t dstX, size_t dstY,
const void * src, size_t count, enum cudaMemcpyKind kind );
```

cudaError_t cudaMemcpy2DToArray (struct cudaArray * dstArray, size_t dstX, size_t dstY,
const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind);

cudaError_t cudaMemcpyArrayToArray (struct cudaArray * dstArray, size_t dstX, size_t dstY,
const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t count, enum cudaMemcpyKind kind);

cudaError_t cudaMemcpy2DArrayToArray (struct cudaArray * dstArray, size_t dstX, size_t dstY,
const struct cudaArray * srcArray, size_t srcX, size_t srcY, size_t width, size_t height,
enum cudaMemcpyKind kind);

Ниже приведен пример работы с текстурной памятью и изображением.

```
#define N 1024
// намеренно выбран тип uint4, встроенный в nvcc, так как он имеет максимально возможный
// размер для текстурной выборки; это позволяет сократить количество выборок
	uint4* gpu_memory;
	texture<uint4, 1, cudaReadModeElementType> texture;

void host_function()
{
	uint cpu_buffer[N];

	...

	cudaMalloc((void**)&gpu_memory, N * sizeof(uint4)); // выделим память в GPU

	// настройка параметров текстуры texture:
	texture.addressMode[0] = cudaAddressModeWrap; // режим Wrap
	texture.addressMode[1] = cudaAddressModeWrap;
	texture.filterMode = cudaFilterModePoint; // ближайшее значение
	texture.normalized = false; // не использовать нормализованную адресацию
	cudaBindTexture(0, texture, gpu_memory, N) // отныне эта память будет считаться
текстурной
                                                                        //копируем
данные на GPU:
	cudaMemcpy(gpu_memory, cpu_buffer, N * sizeof(uint4), cudaMemcpyHostToDevice);
}
	...

// __global__ означает, что device_kernel - ядро, которое нужно распараллелить
__global__ void device_kernel()
{
	uint4 a = tex1Dfetch(texture, 0); // можно выбирать данные только таким способом !
	uint4 b = tex1Dfetch(texture, 1);
	int c = a.x * b.y;

	...
}
```

Стоит отметить, что CUDA имеет широкий ряд возможностей 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.

2. Пример

```
#include <cuda.h>
#include <cuda_runtime.h>
#include "./helper_image.h" // библиотека для работы с текстурами

typedef unsigned int uint;  // для сокращения
typedef unsigned char uchar; // времени написания

texture<uchar, 2, cudaReadModeElementType> g_Texture; // задание текстуры
uint width = 512, height = 512; // ширина и высота изображения


// Реализация тестового ядра
__global__ void Test_Kernel(uchar *pDst, float g, uint w, uint 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 c = tex2D(g_Texture, tidx + 0.5f, tidy + 0.5f);

		// !!! НЕОБХОДИМО УБРАТЬ КОММЕНТАРИИ У ТРЕБУЕМОЙ ФУНКЦИИ

		// преобразование цвета для гамма-коррекции
		//float r = powf(c, g);

		// преобразование для негатива
		float r = 1.0f - c;

		//преобразование для яркости
		//float r = c*0.3f;
		pDst[tidx + tidy *w] = (int)r;
	}
}

// Функции для загрузки и сохранения 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;
}
//

int main(int argc, char ** argv)
{
	// создание указателей на входные и выходные данные
	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"; // входное изображение
	char * d_result_path = "lena_d.pgm"; // выходное изображение

	loadImage(src_path, &h_pixels, &width, &height); // загрузка изображения

	// вычисление размера изображения
	int image_size = sizeof(unsigned char) * width * height;

	// выделение памяти для записи результата на CPU
	h_result_pixels = (unsigned char *)malloc(image_size);

	// выделение памяти для входых данных на GPU
	cudaMalloc((void **)& d_pixels, image_size);

	// копирование входных данных с CPU на GPU
	cudaMalloc((void **)& d_result_pixels, image_size);   cudaMemcpy(d_pixels, h_pixels,
image_size, cudaMemcpyHostToDevice);

	int n = 8; // !!! РАЗМЕР БЛОКА (НЕОБХОДИМО ПОДОБРАТЬ НАИЛУЧШЕЕ ЗНАЧЕНИЕ)
	dim3 block(n, n); // задание блока
	dim3 grid(width / n, height / n); // задание грида
	cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar1>();
	size_t offset = 0;

	// Проверка на успешную привязку текстуры
	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");
	}

	// Выполнение ядра на GPU

	Test_Kernel <<< grid, block >>> (d_result_pixels, 1.2, width, height);

	// копирование результата в память CPU
	cudaMemcpy(h_result_pixels, d_result_pixels, image_size, cudaMemcpyDeviceToHost);
	// сохранение изображения
	saveImage(d_result_path, h_result_pixels, width, height);
	// отвязка текстуры
	cudaUnbindTexture(&g_Texture);
}
```

Используя текстуры, рассмотрим применение CUDA в обработке цифровых сигналов. Рассмотрим серию простых фильтров, которые преобразуют цвет каждого пикселя по определенному закону: гамма-коррекция, фильтр яркости, негатив.

Негатив – это самый простой фильтр, который вычисляет для заданного цвета дополнение его до белого и выражается преобразованием:

$$R = 1 - I$$

Фильтр яркости отбрасывает цветовую информацию, оставляя только информацию о яркости пикселя. Как правило, яркость связывают с цветом, используя скалярное произведение с вектором весовых коэффициентов:

$$lum = \{0.3, 0.59, 0.11\}$$

$$R = dot(lum, I)$$

Гамма-коррекция – это коррекция функции яркости в зависимости от характеристик устройства вывода, которая может быть выражена формулой:

$$R = I^{y}$$

Значение гаммы, равное 1, соответствует "идеальному" монитору, который имеет линейную зависимость отображения от белого к черному. Но таких мониторов не бывает. Зависимость, в особенности для электроннолучевых устройств, нелинейна. Изменяя показатель гамма-коррекции, можно повысить контрастность, разборчивость темных участков изображения, не делая при этом чрезмерно контрастными или яркими светлые детали снимка.

Представленный выше код реализует фильтр негатив (результат работы представлен на рисунке 1). В ядре **Test_Kernel** с помощью добавления/удаления комментариев можно выбрать выполняемую ядром функцию (гамма коррекция, негатив, преобразование яркости).

Для успешной компиляции и выполнения программы необходимы следующие условия:

- Изображения lena.pgm и lena_d.pgm должны находиться в одной директории с файлом kernel.cu.
- Прилагаемые с лабораторной работой библиотеки должны также находиться в одной директории с файлом kernel.cu.

а) исходное изображение

б) изображение после применения фильтра негатив

Рисунок 1. Результаты работы фильтра негатив

3. Обработка цифровых сигналов

Фильтрация и свертка

Если даны две вещественные функции f(x) и g(x), интегрируемые на R, то с математической точки зрения свертка представляет собой функцию вида

$$(f * g)(t) = \int_R f(\tau)g(t-\tau)d\tau.$$

Или в дискретной форме:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]g[n-k].$$

Для простоты изложения дальше будем рассматривать только дискретный случай. Пусть дана цифровая дельта-функция

$$\delta[n] = \begin{cases} 0, n \neq 0 \\ 1, n = 0 \end{cases}.$$

Пусть дана линейная система h[n], которая преобразовывает единичный импульс, например, так, как показано на рисунке 2. Эту линейную систему назовем ядром свертки.

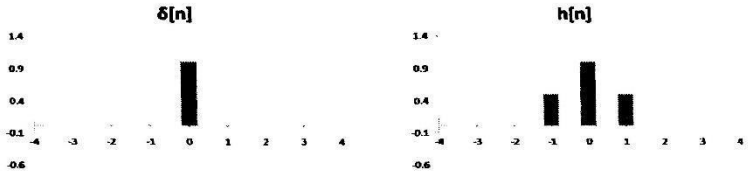

Рисунок 2. Ядро свертки (импульсная характеристика фильтра) h(n)(справа) и дельта-функция (слева)

Любой сигнал можно разложить на сумму таких единичных импульсов, сдвинутых во времени и умноженных на некоторый коэффициент (рис. 3).

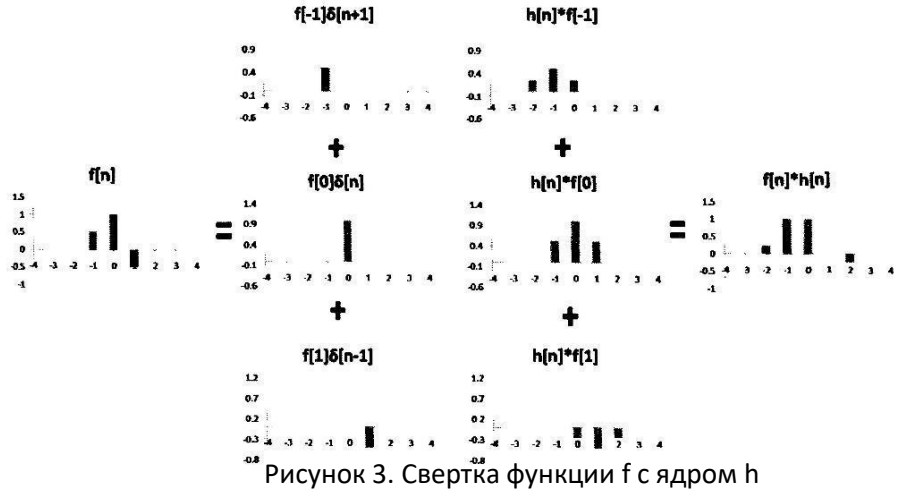

Рисунок 3. Свертка функции f с ядром h

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

Gaussian Blur

Вместо усреднения с одинаковыми коэффициентами можно использовать веса, пропорциональные расстоянию от текущего пикселя. Для одномерного сигнала Гауссово размытие задает веса по следующей формуле:

а) одномерный случай:

$$W(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}};$$

б) двумерный случай:

$$W(x) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Такая фильтрация обладает важным свойством сепарабельности (избирательности): для фильтрации изображения можно сначала произвести фильтрацию по горизонтали, а затем – отдельно по вертикали. Размытие, по сути, является низкочастотным фильтром. Можно рассмотреть некоторую случайную величину ξ (шумовая функция), равномерно распределенную и имеющую нулевое математическое ожидание:

$$E_o = \sum \zeta_i \equiv 0,$$

где под ξi понимается *i*-е случайное значение.

Тогда если изображение все одноцветное (например, все белое), но в каждом пикселе произошло случайное отклонение ξ, то размытие приведет к тому, что шум будет просуммирован в некоторой окрестности каждого пикселя, и тем самым при увеличении радиуса окрестности можно эффективно подавить шум.

Естественно, это не срабатывает на реальных фотографиях, *gaussian blur* приводит к размытию изображения, потери четкости, размытию на краях и потери мелких деталей.

```
__global__ void Gaussian_Kernel(unsigned char * pDst, float radius, float sigma_sq, int w,
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 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_Texture, tidx + 0.5f + ic, tidy + 0.5f)* weight;
			weight_sum += weight;
		}
		//нормализация полученных результатов
		r /= weight_sum;
		pDst[tidx + tidy*w] = (int)r;
	}
}
```

Билинейная фильтрация

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

$$f(k+\alpha) = (1-\alpha) f(k) + \alpha f(k+1).$$

В двумерном случае:

$$f(m+\alpha, n+\alpha) = (1-\beta)((1-\alpha) f(m,n) + \beta f(m+1,n))$$
$$+ \beta((1-\alpha) f(m,n+1) + \alpha f(m+1,n+1))$$

```
__global__ void Bilinear_Kernel(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 = (b - a)*t + a;
		dest[tidx + tidy*w] = (int)(linear);
	}
}
```

Особенностью данного ядра можно назвать то, что выборки производятся по столбцам, а запись по строкам. Это сделано для того, чтобы избежать проблем с коалесингом при чтении (рассчитываем на способность текстуры эффективно закешировать данные) и обеспечить коалесинг на запись. Таким образом, сначала изображение растягивается по высоте и поворачивается на 90^0. Если подать на вход данному алгоритму изображение еще раз, то оно промасштабируется по ширине и опять повернется на 90^0.

а) б) в)

Рисунок 2. 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.

Bilateral Blur

Bilateral, или, как его еще называют, *K* ближайших соседей (от англ. *K Nearest Neighbors*), фильтр устроен таким образом, чтобы удалять белый шум, и, по сути, является модификацией размытия Гаусса. Если $I(x)$ – это исходно немодифицированное изображение, то результатом фильтрации назовем $R(x)$ такое, что:

$$R(\mathbf{x}) = \frac{1}{C(\mathbf{x})} \int_{\Omega(\mathbf{x})} I(\mathbf{y}) e^{-\frac{|\mathbf{y}-\mathbf{x}|^2}{r^2}} e^{-\frac{|I(\mathbf{y})-I(\mathbf{x})|^2}{h^2}} d\mathbf{y},$$

где $\Omega(P)$ – окрестность пикселя p, (как правило, мы будем рассматривать окрестности размера NxN где $N = 2r + 1$), а $C(x)$ – нормализующий коэффициент. Параметр h отражает зашумленность данного блока и должен быть оценен независимо. Такое преобразование можно рассматривать как свертку, в которой весовые коэффициенты равны весовым коэффициентам Гаусса, модифицированным таким образом, чтобы отражать не только пространственную близость двух пикселей на изображении, но и их близость в цветовом смысле.

```
__global__ void Bilateral_Kernel(uchar * pDst, int radius, float inv_sigma_sq, float area,
	float weight_threshold, uint w, uint h)
{
	int tidx = threadIdx.x + blockIdx.x * blockDim.x;
	int tidy = threadIdx.y + blockIdx.y * blockDim.y;
		if (tidx < w && tidy < h)
		{
			float r = 0;
			float weight_sum =  0.0f;
			float weight = 0.0f;
			float weight_threshold_counter = 0.0f;
			float c00 = tex2D(g_Texture, tidx + 0.5f, tidy + 0.5f);
			for (int ir = -radius; ir <= radius; ir++)
				for (int ic = -radius; ic <= radius; ic++)
				{
					float c = tex2D(g_Texture, tidx + 0.5f + ic, tidy + 0.5f +
ir);
					uint param = 0; // !!! ПОПРОБУЙТЕ ИЗМЕНЯТЬ ДАННЫЙ ПАРАМЕТР
					weight = exp(-((ir*ir) + (ic*ic)) * inv_sigma_sq - param);
// !!! ПАРАМЕТР ИСПОЛЬЗУЕТСЯ ЗДЕСЬ
					weight_sum += weight;
					weight_threshold_counter += (weight >= weight_threshold ?
1.0f : 0.0f);
					r += (c * weight);
				}
			r /= weight_sum;
			r = (c00 - weight_threshold_counter/area)*r +
weight_threshold_counter/area;
			pDst[tidx + tidy * w] = (int)r;
		}
}
```

Стоит обратить внимание, что на каждом шаге вычисления веса полученный вес сравнивается с неким пороговым значением (пороговое значение – параметр алгоритма), и ведется счет, сколько весовых коэффициентов превзошло данный порог. При выходе из цикла происходит линейная интерполяция между оригинальным значением пикселя и результатом свертки.

Рассмотрим предельный случай:

1) пороговое значение равно нулю, следовательно, после интерполяции получаем точно результат свертки;

2) пороговое значение равно единице, следовательно, после интерполяции получаем точно исходный пиксель.

Чем больше весовых коэффициентов пройдут это нехитрое сравнение, тем более однородной (по цвету) является область, а значит, свертка эффективно подавит шум и не размоет границ. Если же в данной области проходит граница, то многие цвета будут разными, значит, и весовые коэффициенты будут меньше. Следовательно, отношение счетчика к общему количеству пикселей в области будет близко к нулю, то есть нам бы хотелось сохранить с большим весом старое значение.

Интерполяция Ланкзоса

Фильтр Ланкзоса – это оконная версия *sinc*-интерполяции. Функция *sinc* не бесконечно убывает к нулю, но не равна ему, поэтому плохо применима на практике. Можно аппроксимировать ее, используя некоторое "окно", то есть некоторую функцию, которая обращается в нуль вне отрезка [-*r*, *r*], что позволяет балансировать точность интерполяции и ее вычислительную сложность.

Импульсная характеристика фильтра Ланкзос – это нормализованная *sinc* (*x*)-функция, взвешенная с окном Ланкзоса. Само окно Ланкзоса – это центральная область *sinc*(*x*/*a*) для интервала $x \in (-a, a)$.

Таким образом, Ланкзос-фильтр на своем интервале представляется произведением двух *sinc*()-функций, а сам процесс масштабирования выражается через свертку со следующим ядром:

$$L(x) = \begin{cases} \sin c(x)\sin c(x/a), & -a < x < a, x \neq 0 \\ 1 & x = 0 \\ 0 & \textit{иначе} \end{cases}.$$

```
__device__ float Lanczos(float x, float r)
{
	const float m_pi = 3.14159265f;
	float result = 0.0f;
	if (x >= -r && x <= r) {
		float a = x*m_pi;
		float b = (r*(sin(a / r)) * (sin(a)) / (a*a));
		result = (x == 0.0f) ? 1.0f : b;
	}
	return result;
}

__global__ void Lanczos_Kernel(uchar * pDst, float factor, float blur, float radius, float
	support, float scale, uint w, uint 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, weight = 0.0f;
		float center = tidx / factor;
		uint start = (uint)fmax(center - support + 0.5f, (float)0);
		uint stop = (uint)fmin(center + support + 0.5f, (float)w);
		float nmax = stop - start;
		float s = start - center;
		for (int n = 0; n < nmax; ++n, ++s) {
			weight = Lanczos(s*scale, radius);
			weight_sum += weight;
			r += (tex2D(g_Texture, tidy + 0.5f, start + n + 0.5f)* weight);
		}
		if (weight_sum != 0.0f)
		{//нормализация полученных результатов
			r /= weight_sum;
		}
		pDst[tidx + tidy*w] = int(r);
	}
}
```

4. Задание

1. Создание и отладка работы программы.

Требуется написать и отладить работу программы, в которой будут реализованы следующие ядра:

- Test_Kernel;
- Bilinear_Kernel;
- Gaussian_Kernel;
- Bilateral_Kernel;
- Lanczos_Kernel.

Результатом являются 7 изображений:

1) lena_gam (Test_Kernel – гамма коррекция);
2) lena_neg (Test_Kernel – негатив);
3) lena_br (Test_Kernel – преобразование яркости);
4) lena_bilinear;
5) lena_gaussian;
6) lena_bilateral;
7) lena_lanczos.

Замечание: 7 необработанных изображений формируются вручную путем копирования и соответствующего переименования файла lena.pgm.

2. Изменение входных параметров.

Требуется разобраться во входных параметрах ядер: Bilinear_Kernel, Gaussian_Kernel, Bilateral_Kernel, Lanczos_Kernel; запустить каждое упомянутое выше ядро с 2 различными вариантами входных параметров и предоставить соответствующие изображения **(разница на изображениях должна быть очевидна)**.

Результатом являются 8 изображений:

1) lena_bilinear_1;
2) lena_bilinear_2;
3) lena_gaussian_1;
4) lena_gaussian_2;
5) lena_bilateral_1;
6) lena_bilateral_2;
7) lena_lanczos_1;
8) lena_lanczos_2.

3. Оценка времени работы и выбор значения размера блока.

Требуется выбрать (на Ваш взгляд) самое медленное и самое быстрое (в плане времени выполнения) ядро. Необходимо засечь время выполнения обоих ядер и вывести результаты на консоль, также надо вывести размер блока. Далее необходимо увеличить/уменьшить размер блока и продемонстрировать полученные результаты для нескольких значений размеров блока. Выбрать такое значение размера блока, при котором достигается минимальное время выполнения ядер, объяснить свой выбор.

Замечание: засекать только время исполнения ядра (без учета копирований в глобальную память GPU и прочих пересылок).

4. Добавление своего ядра (фильтра). НЕ ОБЯЗАТЕЛЬНО

Выполнение данного задания предусматривает поощрение баллами со стороны преподавателя (обговаривается заранее).

Требуется добавить реализацию любого фильтра изображения, продемонстрировать результат его работы.

Замечание: Можно воспользоваться 1 ссылкой из списка источников ниже, перейти в пункт «**3.4. Imaging Reference**» (находится слева), выбрать из заинтересовавший пример. Найти все примеры можно в директории: C:\ProgramData\NVIDIA Corporation\CUDA Samples\<Ваша версия>\ 3_Imaging. Второй вариант: Пуск->Все Программы->Nvidia Corporation->Browse Cuda Samples->3_Imaging.

5. Источники и литература

1. http://docs.nvidia.com/cuda/cuda-samples/index.html
2. http://www.nvidia.ru/object/cuda_education_ru1.html
3. http://www.nvidia.ru/object/cuda-openacc-online-course-ru.html
4. А.В. Боресков, А.А. Харламов. Основы работы с технологией Cuda. – М: ДМК Пресс, 2010. – 232 с.
5. 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.