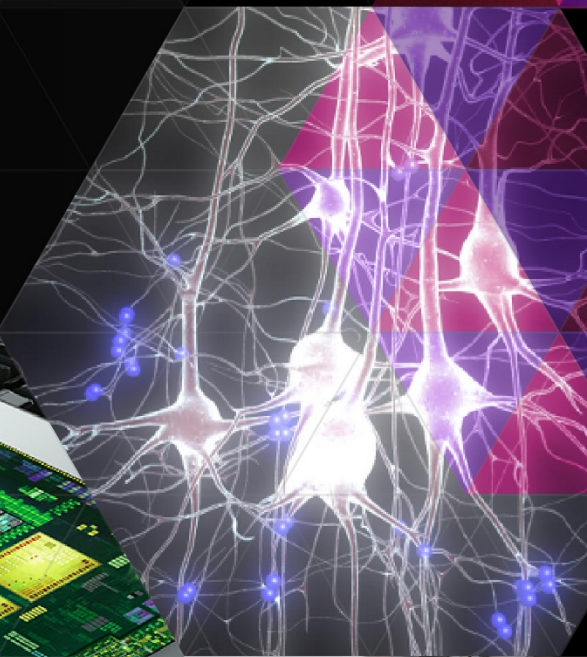
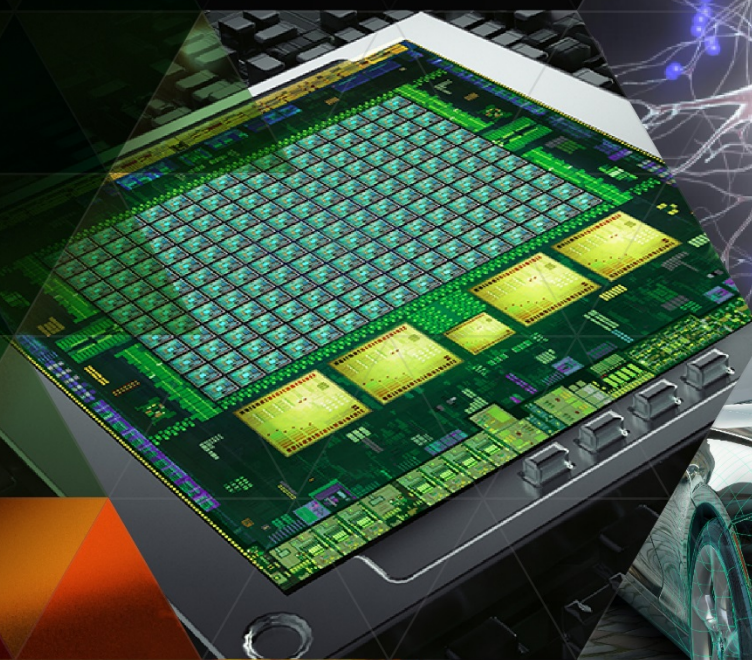




NVIDIA CUDA И OPENACC ЛЕКЦИЯ 7

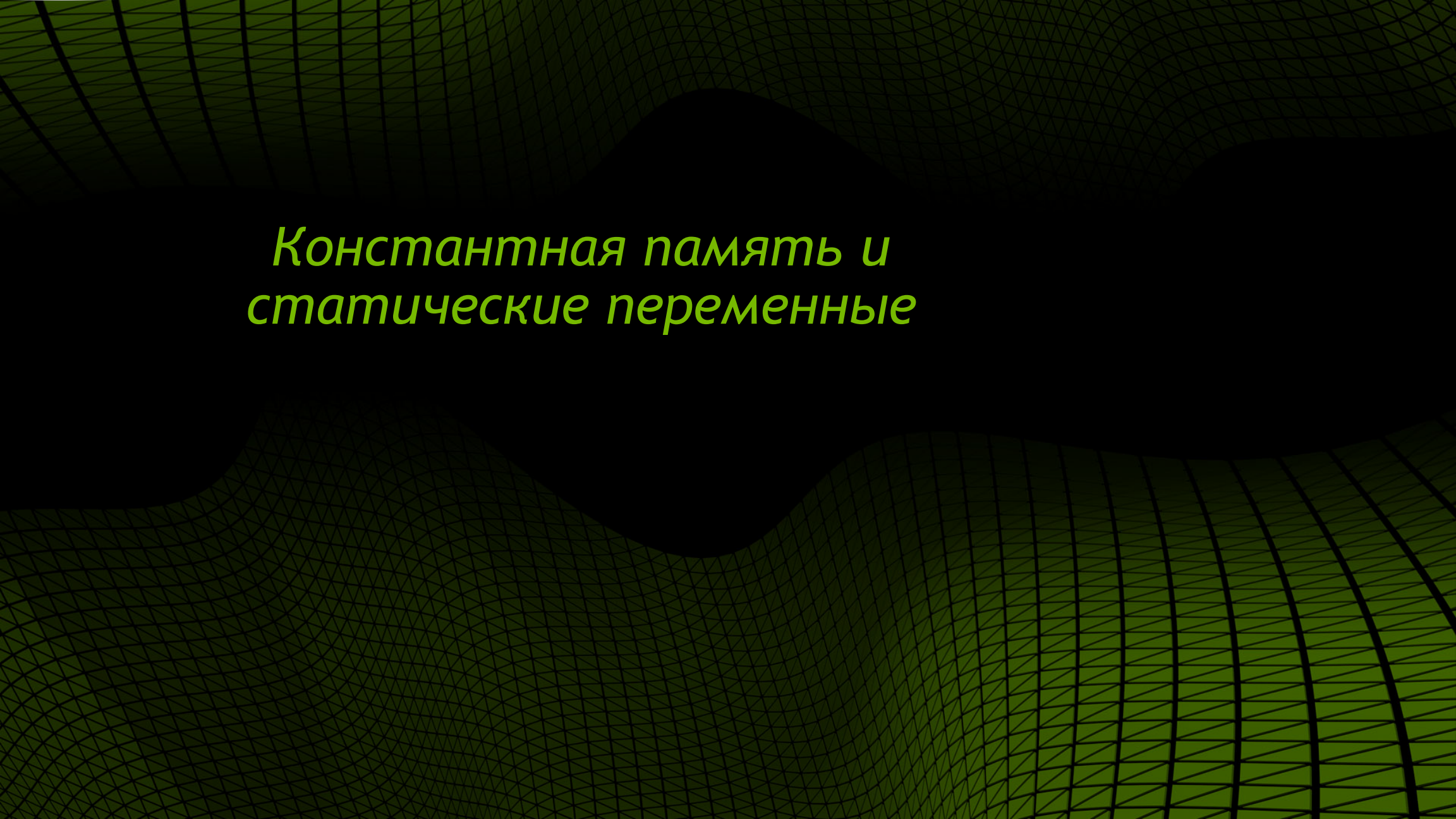
Перепёлкин Евгений



СОДЕРЖАНИЕ

Лекция 7

- ▶ Константная память и статические переменные
- ▶ Текстурная память
- ▶ Примеры
 - ▶ Вычисление свёртки
 - ▶ Интерполяция сеточной функции
 - ▶ Численное решение СЛАУ
 - ▶ Работа с двойной точностью (double)



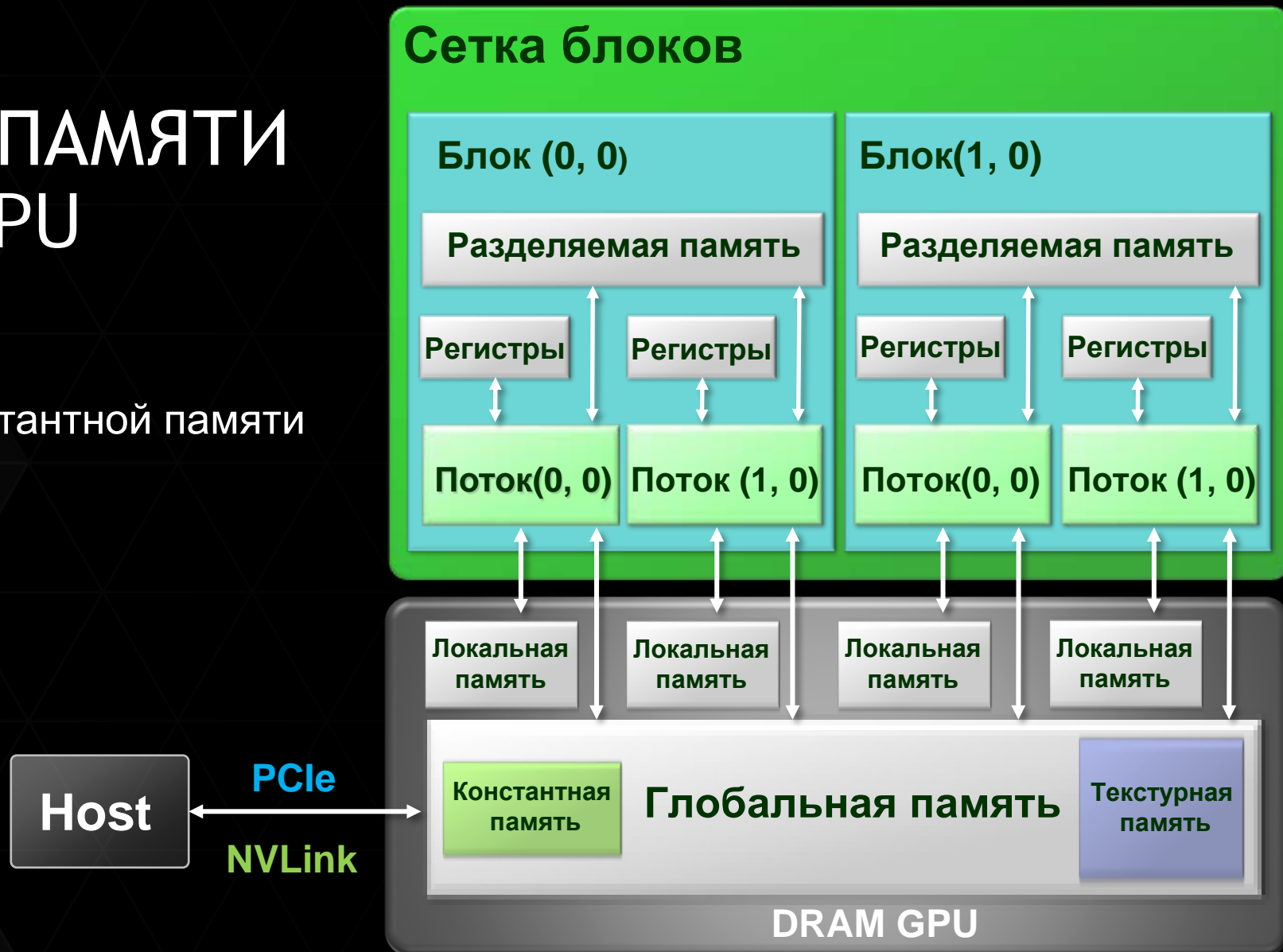
Константная память и статические переменные

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Register (регистровая)	RW	Per-thread	Высокая (on-chip)
Local (локальная)	RW	Per-thread	Низкая (DRAM)
Global (глобальная)	RW	Per-grid	Низкая (DRAM)
Shared (разделяемая)	RW	Per-block	Высокая (on-chip)
Constant (константная)	RO	Per-grid	Высокая (L1 cache)
Texture (текстурная)	RO	Per-grid	Высокая (L1 cache)

ДОСТУП К ПАМЯТИ НА GPU

На SM/SMX 64 КБ константной памяти



КОНСТАНТНАЯ ПАМЯТЬ

Шаблон работы

```
#include <stdio.h>
__constant__ float constData[256];

int main ()
{float src[256]; // «ИСХОДНЫЙ» массив
 float dst[256]; // «КОНЕЧНЫЙ» массив
 for (int i=0;i<256;i++) src[i]=i*i; // заполнение массива src

 cudaMemcpyToSymbol(constData, src, sizeof(src)); // копирование host-device
 cudaMemcpyFromSymbol(dst, constData, sizeof(src)); // device-host

 for (int i=0;i<256;i++) printf ("\n src = %e, dst = %e",src[i],dst[i]);
 return 0;
}
```

СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ

Шаблон работы переменной

```
#include <stdio.h>

__device__ float devData;

int main ()
{float value_src = 3.14f; // «исходная» переменная
  float value_dst;        // «конечная» переменная

  cudaMemcpyToSymbol(devData, &value_src, sizeof(float));
  cudaMemcpyFromSymbol(&value_dst, devData, sizeof(float));

  printf("\n value_src = %e, value_dst = %e", value_src, value_dst);

  return 0;
}
```

СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ

Шаблон работы с массивом

```
#include <stdio.h>
__device__ float *devPointer;

int main ()
{int N = 256;
  float *src = (float*)malloc(N*sizeof(float));
  float *dst = (float*)malloc(N*sizeof(float));
  for (int i=0;i<N;i++) src[i] = i*i;

  cudaMemcpyToSymbol (devPointer, &scr, sizeof (scr));
  cudaMemcpyFromSymbol (&dst, devPointer, sizeof (scr));

  for (int i=0;i<N;i++) printf ("\n scr = %e, dst = %e",scr[i],dst[i]);
  free (scr); free (dst);

  return 0;
}
```

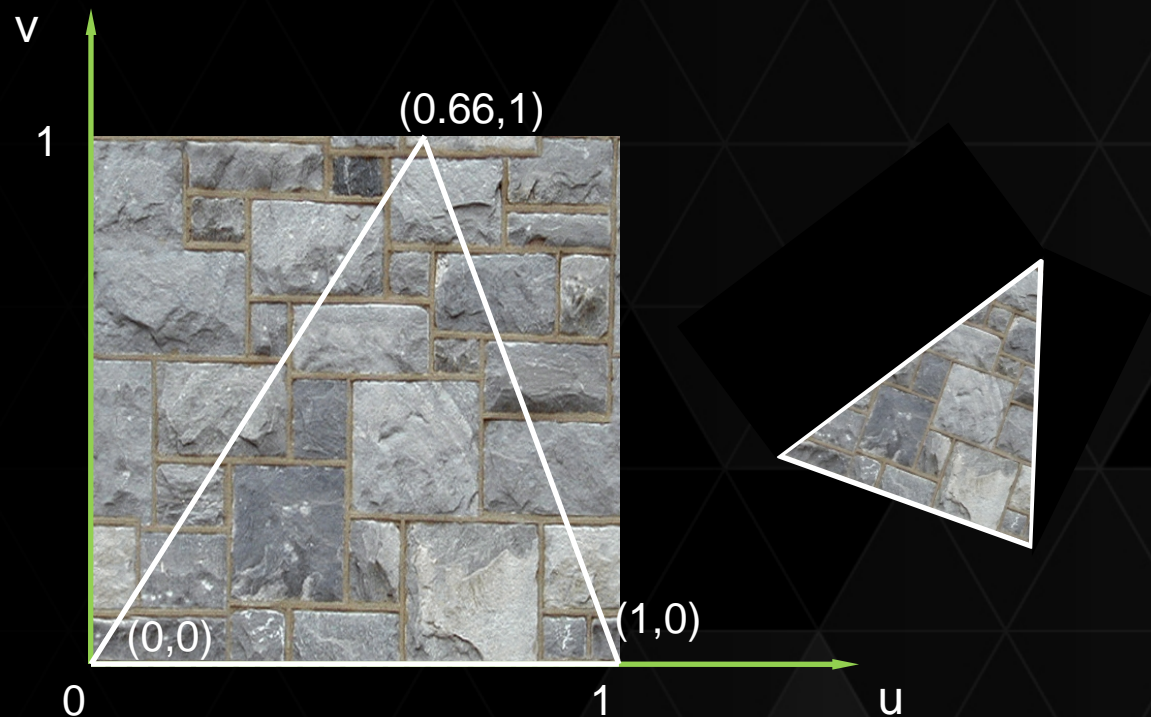
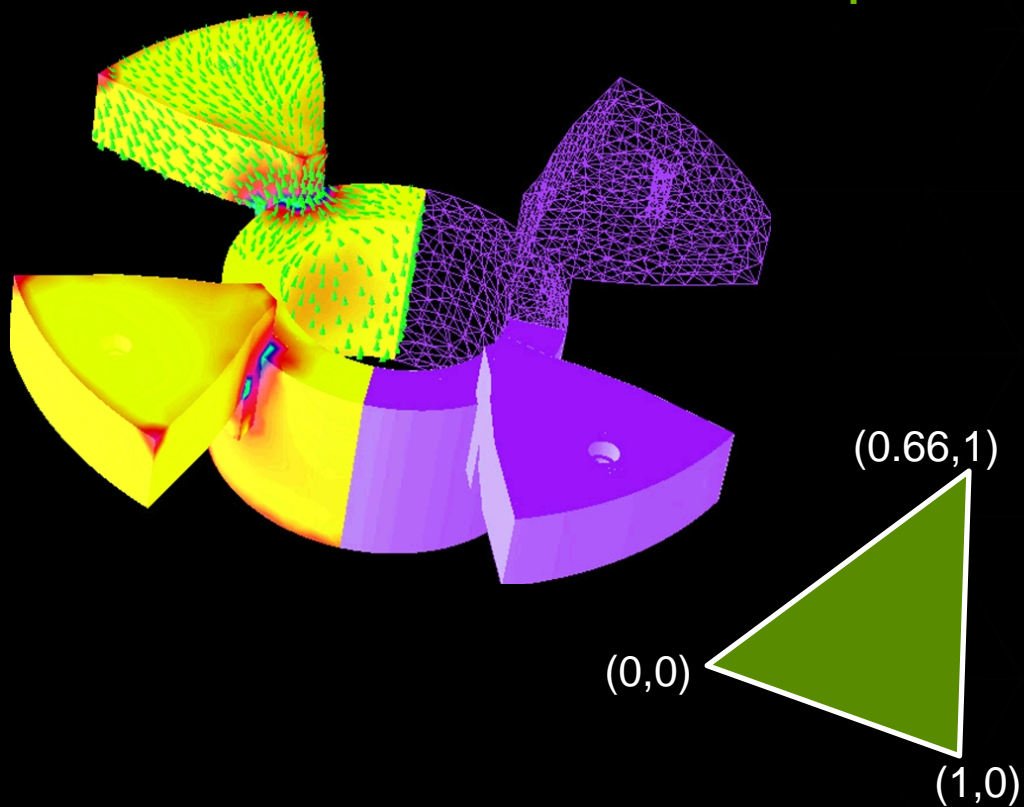
Текстурная память

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Register (регистровая)	RW	Per-thread	Высокая (on-chip)
Local (локальная)	RW	Per-thread	Низкая (DRAM)
Global (глобальная)	RW	Per-grid	Низкая (DRAM)
Shared (разделяемая)	RW	Per-block	Высокая (on-chip)
Constant (константная)	RO	Per-grid	Высокая (L1 cache)
Texture (текстурная)	RO	Per-grid	Высокая (L1 cache)

TEXTURE 3D

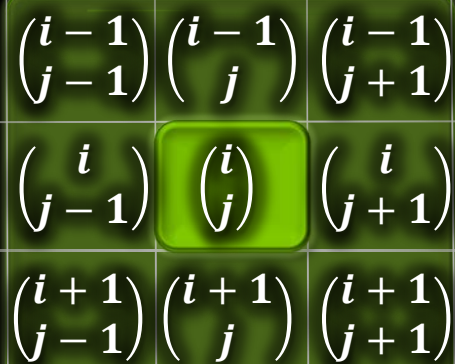
«Раскрашивание» треугольников



ОСОБЕННОСТИ РАБОТЫ ТЕКСТУРНОГО КЭША

«Локализованное» кэширование

При обращении к элементу
массива $\binom{i}{j}$ кэшируется его
2D- окрестность



The diagram shows a 3x3 grid of elements in a 2D array. The elements are labeled with binomial coefficients $\binom{i}{j}$. The central element, $\binom{i}{j}$, is highlighted in green. The surrounding elements are $\binom{i-1}{j-1}$, $\binom{i-1}{j}$, $\binom{i-1}{j+1}$ in the top row; $\binom{i}{j-1}$, $\binom{i}{j+1}$ in the middle row; and $\binom{i+1}{j-1}$, $\binom{i+1}{j}$, $\binom{i+1}{j+1}$ in the bottom row.

$\binom{i-1}{j-1}$	$\binom{i-1}{j}$	$\binom{i-1}{j+1}$
$\binom{i}{j-1}$	$\binom{i}{j}$	$\binom{i}{j+1}$
$\binom{i+1}{j-1}$	$\binom{i+1}{j}$	$\binom{i+1}{j+1}$

ТЕКСТУРНАЯ ПАМЯТЬ

Особенности работы

- ▶ Латентность больше, чем у прямого обращения в память
 - ▶ Дополнительные стадии в конвейере:
 - ▶ Преобразование адресов
 - ▶ Фильтрация
 - ▶ Преобразование данных
- ▶ Наличие кэша
 - ▶ Разумно использовать, если:
 - ▶ Объем данных не влезает в shared память
 - ▶ Паттерн доступа хаотичный
 - ▶ Данные переиспользуются разными потоками

ТЕКСТУРНАЯ ПАМЯТЬ

Особенности работы

- ▶ Существует два разных API's доступа к Texture и Surface memory
 - ▶ Texture reference API, поддерживаемое на всех видеокартах
 - ▶ Texture Object API, поддерживаемое только на видеокартах с CC 3.x
- ▶ В отличие от Texture Object API доступ через Texture reference API имеет ограничения
- ▶ Память
 - ▶ Linear memory
 - ▶ CUDA Array

TEXTURE REFERENCE API

cudaArray

- ▶ Данные представлены в виде 1D/2D/3D- массивов
- ▶ Элементами (texel) 1D/2D/3D- массивов являются 1/2/4-компонентные векторы
 - ▶ signed/unsigned 8/16/32-bit integers
 - ▶ 16/32-bit float
- ▶ Обращение через функции tex1D/tex2D/tex3D

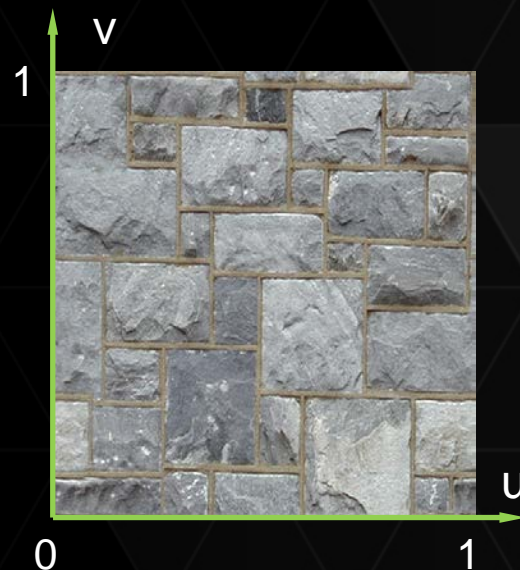
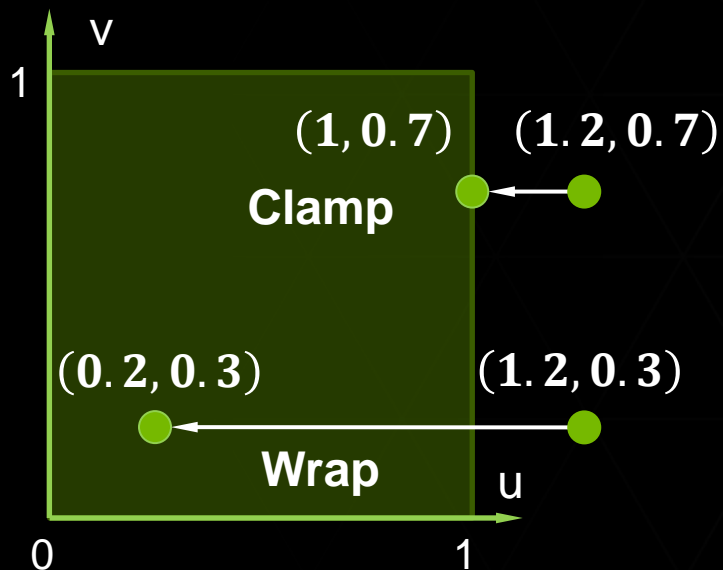
TEXTURE REFERENCE API

cudaArray

- ▶ Текстурные координаты
 - ▶ Целочисленные индексы $(0 \dots N - 1)$
 - ▶ Нормализованные координаты $(0 \dots 1 - \frac{1}{N})$
- ▶ Преобразование адресов на границах

- ▶ Clamp

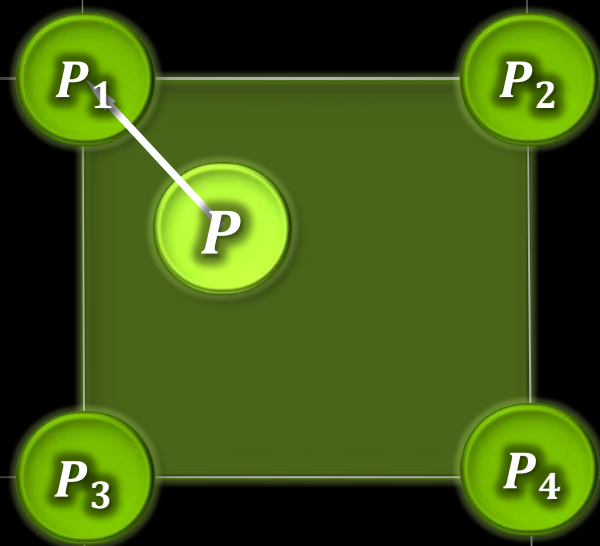
- ▶ Wrap



TEXTURE REFERENCE API

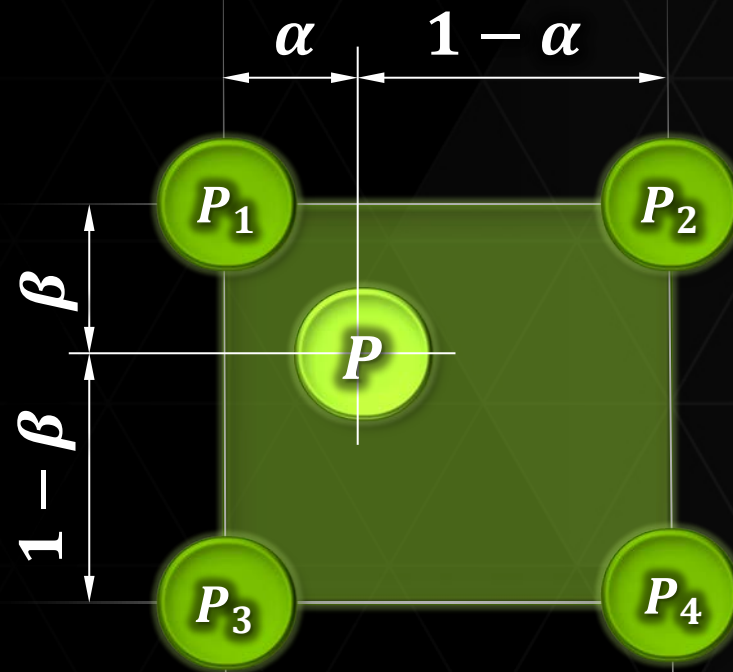
Фильтрация данных

► Point



$$P = P_1$$

► Linear



$$P(\alpha, \beta) = (1 - \beta)[(1 - \alpha)P_1 + \alpha P_2] + \beta[(1 - \alpha)P_3 + \alpha P_4]$$

TEXTURE REFERENCE API

cudaArray

- ▶ Преобразование данных

- ▶ **cudaReadModeElementType**

- ▶ Исходный массив содержит данные в integer, возвращаемое значение во floating point представлении (доступный диапазон значений отображается в интервал $[0,1]$ или $[-1,1]$). Например, элемент unsigned 8-bit со значением 0xff перейдет в 1

- ▶ **cudaReadModeNormolizedFloat**

- ▶ Возвращаемое значение то же, что и во внутреннем представлении

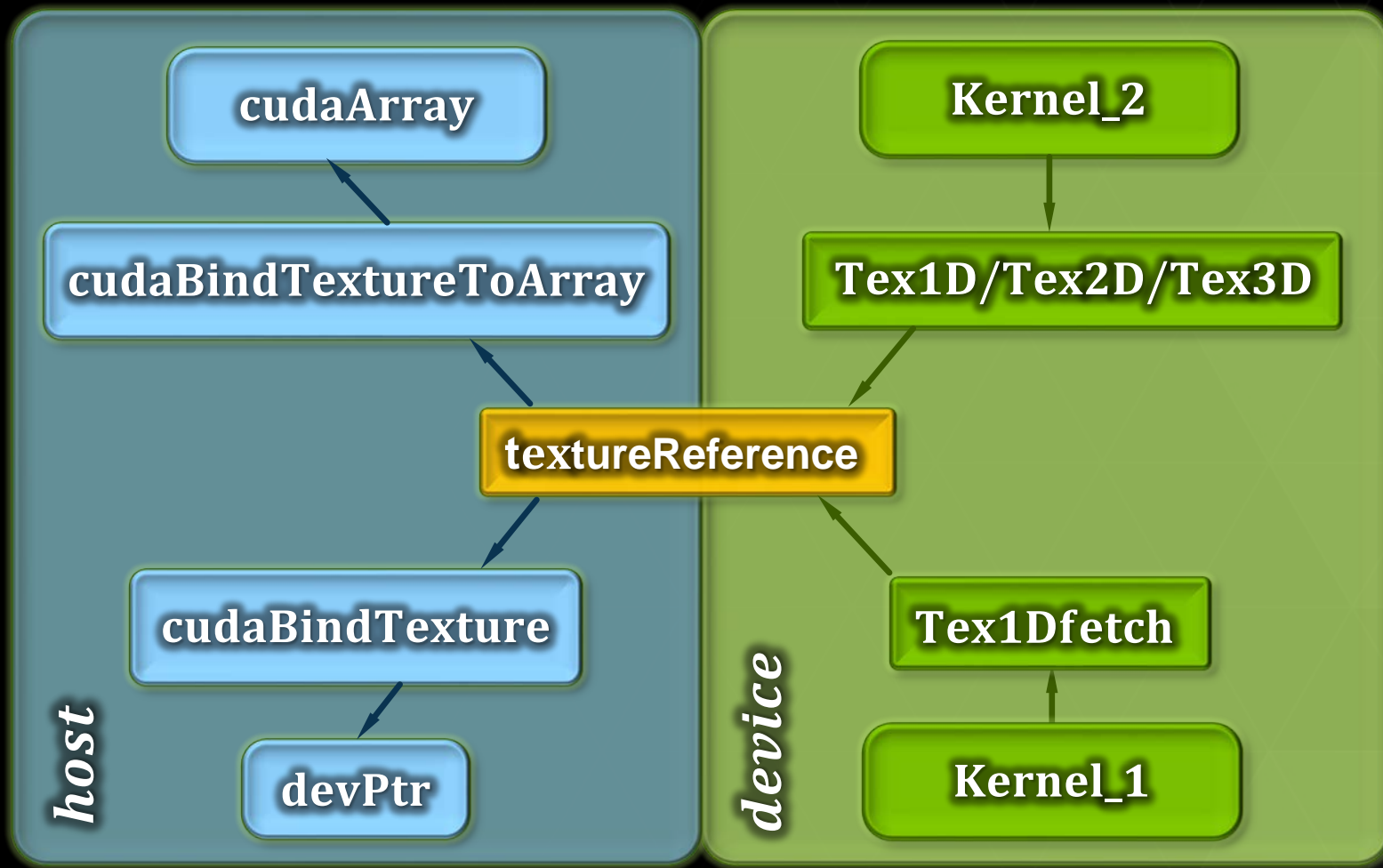
TEXTURE REFERENCE API

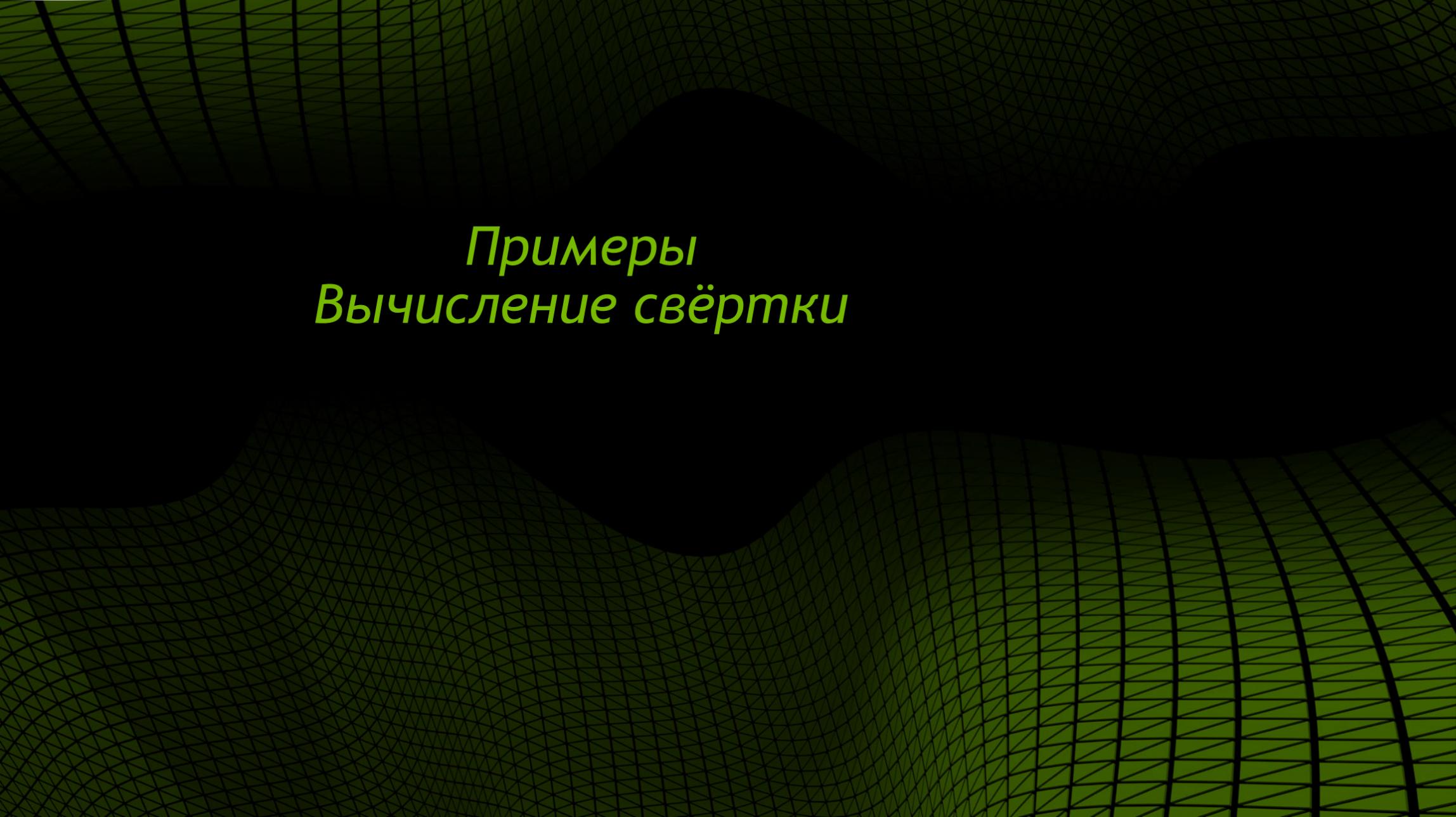
Linear memory

- ▶ Ограничения при использовании «линейной» памяти
 - ▶ Только для одномерных массивов
 - ▶ Нет фильтрации
 - ▶ Доступ по целочисленным координатам
 - ▶ Обращение по адресу вне допустимого диапазона возвращает ноль

TEXTURE REFERENCE API

cudaArray и linear memory





*Примеры
Вычисление свёртки*

ВЫЧИСЛЕНИЕ СВЁРТКИ

cudaArray, tex2D

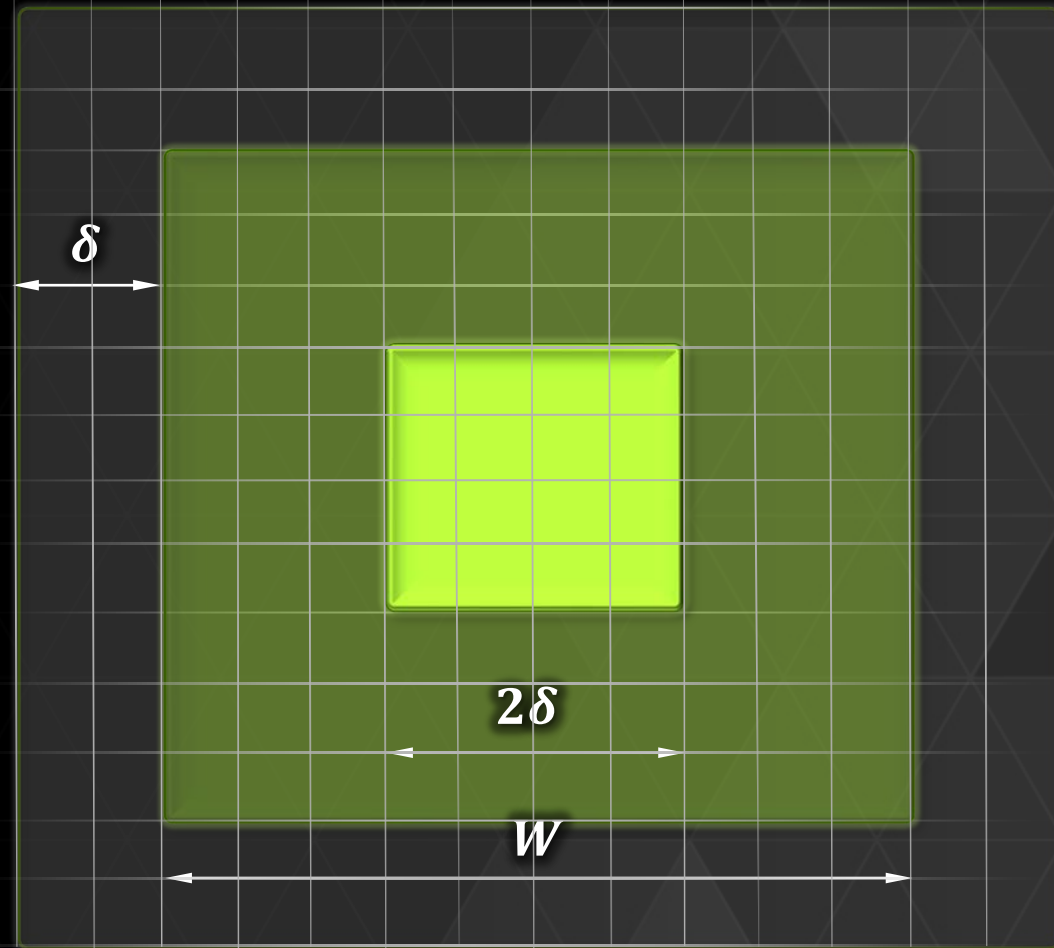
$$\text{Conv}_{n,m} = \frac{1}{N} \sum_{i=-\delta}^{\delta} \sum_{j=-\delta}^{\delta} K_{i,j} S_{i+n,j+m},$$

$$N = \sum_{i=-\delta}^{\delta} \sum_{j=-\delta}^{\delta} K_{i,j}, \quad \begin{aligned} n &= \delta \dots H - 1 + \delta, \\ m &= \delta \dots W - 1 + \delta, \end{aligned}$$

Возьмем

$$K_{i,j} = e^{-\frac{i^2+j^2}{\delta^2}}, \quad W = H,$$

$$S_{i,j} = \sin \frac{2\pi i}{W + 2\delta} \sin \frac{2\pi j}{H + 2\delta}.$$



ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция-ядро «Conv_Glb»

```
#define BLOCK_SIZE 16 // размер ядра (2*delta)
__global__ void Conv_Glb (float *dConv, float *dS, int W, int H, int delta )
{int idx = blockIdx.x*blockDim.x+threadIdx.x + delta;
  int idy = blockIdx.y*blockDim.y+threadIdx.y + delta;

  float norm = 0.0f; float cov = 0.0f;

  for(int ix = -delta; ix <= delta; ix++)
    for(int iy = -delta; iy <= delta; iy++)
    {float K = expf( -(ix*ix + iy*iy) / (delta*delta) );
      cov += K * dS[idx + ix + (idy + iy) *(W+BLOCK_SIZE)];
      norm += K;
    }
  dConv[idx + idy * (W+BLOCK_SIZE)] = cov/norm;
}
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция «main»

```
int main ()
{float timerValueGPU, timerValueCPU;
 cudaEvent_t start, stop;
 cudaEventCreate(&start); cudaEventCreate(&stop);

int delta = BLOCK_SIZE/2; // delta сдвиг
int W = 512;  int H = 512;
float PI = 3.1415;
float norm,cov;
int idx,idy,ix,iy;

int size = (W+BLOCK_SIZE)*(H+BLOCK_SIZE);
unsigned int mem_size = sizeof(float)*size;
float *hS, *hConv, *dS, *dConv, *hdConv;
hS      = (float*) malloc (mem_size);
hConv   = (float*) malloc (mem_size); // CPU вариант
hdConv  = (float*) malloc (mem_size); //GPU вариант
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция «main»

```
cudaMalloc((void**)&dConv, mem_size);
cudaMalloc((void**)&dS, mem_size);

for(idy = 0; idy < H+BLOCK_SIZE; idy++)
{for(idx = 0; idx < W+BLOCK_SIZE; idx++)
{hs[idx+idy*(BLOCK_SIZE+W)] =
    sinf(idx*2.0f*PI/(W+BLOCK_SIZE))*sinf(idy*2.0f*PI/(H+BLOCK_SIZE));
    hConv[idx+idy*(BLOCK_SIZE+W)] = 0.0f;
}
}

dim3 nThreads(BLOCK_SIZE, BLOCK_SIZE);
dim3 nBlocks(W/nThreads.x, H/nThreads.y);
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция «main»

```
//----- GPU вариант -----  
cudaEventRecord(start, 0);  
cudaMemcpy (dS, hS, mem_size, cudaMemcpyHostToDevice);  
cudaMemset (dConv, 0, mem_size );  
  
Conv_Glb <<<nBlocks, nThreads>>> (dConv,dS,W,H,delta);  
  
cudaMemcpy (hdConv, dConv, mem_size, cudaMemcpyDeviceToHost);  
  
cudaEventRecord (stop, 0); cudaEventSynchronize(stop);  
cudaEventElapsedTime(&timerValueGPU, start, stop);  
printf("\n GPU calculation time %f msec\n",timerValueGPU);
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция «main»

```
//----- CPU вариант -----  
cudaEventRecord (start, 0);  
for(idy = 0; idy < H; idy++)  
{for(idx = 0; idx < W; idx++)  
  {norm = 0.0f; cov = 0.0f;  
   for(ix = -delta; ix <= delta; ix++)  
     for(iy = -delta; iy <= delta; iy++)  
       {float K = expf(-(ix*ix+iy*iy)/(delta*delta));  
        cov += K*hs[idx+delta+ix+(idy+delta+iy)*(W+BLOCK_SIZE)]; norm += K;  
       }  
   hConv[idx+delta+(idy+delta)*(W+BLOCK_SIZE)] = cov/norm;  
  }  
}  
cudaEventRecord (stop, 0); cudaEventSynchronize (stop);  
cudaEventElapsedTime (&timerValueCPU, start, stop);  
printf ("\n CPU calculation time %f msec\n", timerValueCPU);  
printf ("\n Rate %f x\n", timerValueCPU/timerValueGPU);
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «GLOBAL»

Функция «main»

```
free(hS);  
free (hConv);  
free (hdConv);  
cudaFree(dS);  
cudaFree (dConv);  
cudaEventDestroy ( start );  
cudaEventDestroy ( stop );  
  
return 0;  
}
```

Результаты

```
GPU calculation time 192.6 ms  
CPU calculation time 2498.7 ms  
Rate 12.97 x
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «TEXTURE»

Функция-ядро «Conv_Tex»

```
texture <float, 2, cudaReadModeElementType> S_TexRef;

__global__ void Conv_Tex (float *dConv, int W, int H, int delta)
{int idx = blockIdx.x*blockDim.x+threadIdx.x + delta;
  int idy = blockIdx.y*blockDim.y+threadIdx.y + delta;

  float norm = 0.0f; float cov = 0.0f;

  for(int ix = -delta; ix <= delta; ix++)
  for(int iy = -delta; iy <= delta; iy++)
  {float K = expf( -(ix*ix + iy*iy) / (delta*delta) );
    cov += K * tex2D(S_TexRef, idx+ix+0.5f, idy+iy+0.5f); norm += K;
  }
  dConv[idx + idy * (W+BLOCK_SIZE)] = cov/norm;
}
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «TEXTURE»

Фрагмент функции «main»

```
int main ()
{...
    cudaArray *ContS; // указатель на контейнер в текстурной памяти

    // определение вида элементов, содержащихся в контейнере
    // одномерные векторы (32,0,0,0), с компонентой 32 бита, типа float

    cudaChannelFormatDesc DescS =
        cudaCreateChanelDesc(32,0,0,0,cudaChannelFormatKindFloat);

    // выделение двумерного контейнера в текстурной памяти
    cudaMallocArray(&ContS, &DescS, W+BLOCK_SIZE,H+BLOCK_SIZE);
    ...
}
```

ВЫЧИСЛЕНИЕ СВЁРТКИ. ВАРИАНТ «TEXTURE»

Фрагмент функции «main»

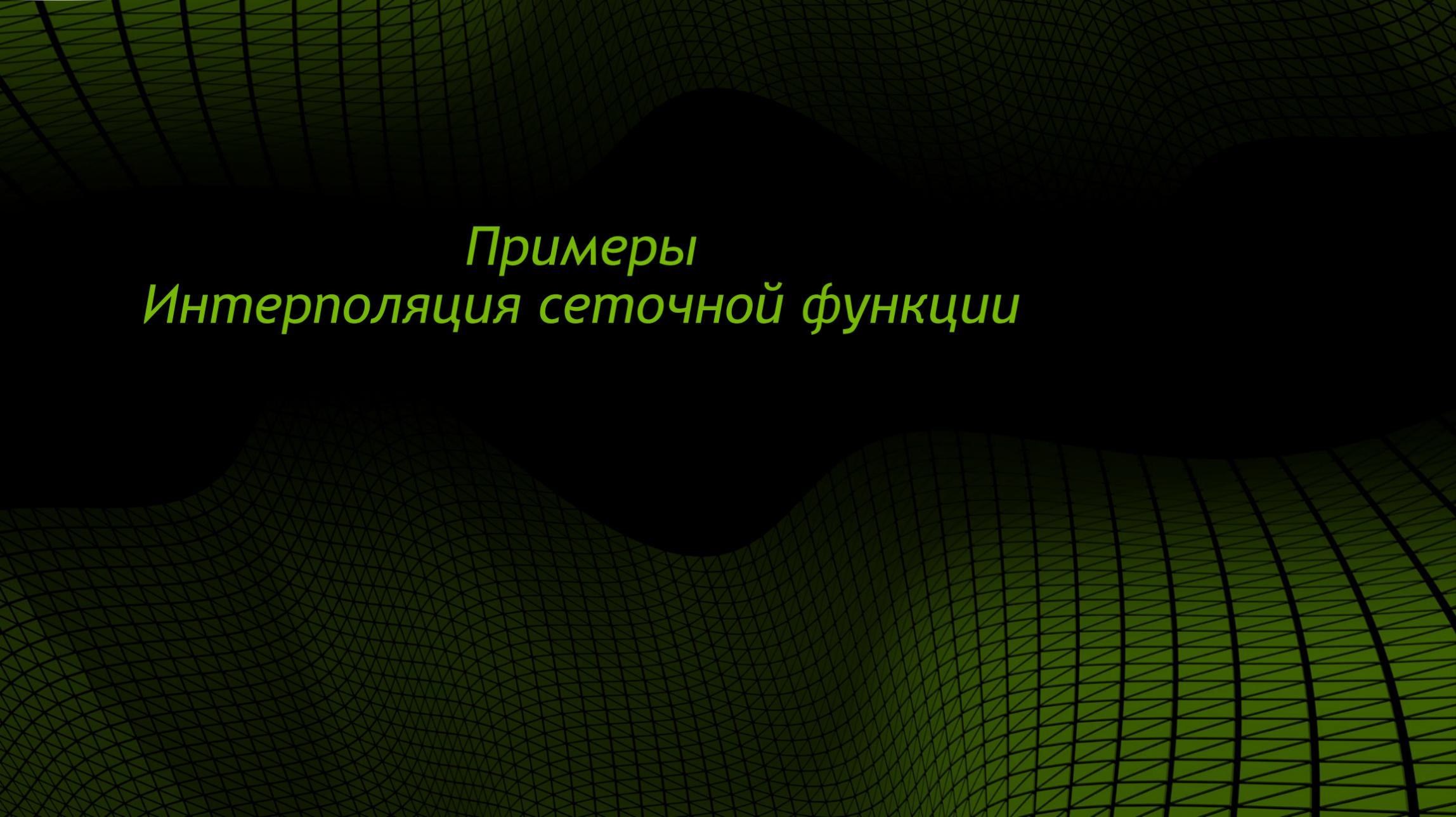
```
// ----- GPU вариант -----  
cudaEventRecord(start, 0); cudaMemcpy (dConv, 0, mem_size );  
// копирование функции сигнала S с host на device в текстурную память  
cudaMemcpyToArray(ContS, 0, 0, hS, mem_size, cudaMemcpyHostToDevice);  
// «прикручивание» текстурной ссылки к контейнеру  
cudaBindTextureToArray(S_TexRef, ContS);  
  
Conv_Tex <<<nBlocks, nThreads>>> ( dConv, W, H, delta);  
cudaMemcpy (hdConv, dConv, mem_size, cudaMemcpyDeviceToHost);  
  
cudaEventRecord (stop, 0); cudaEventSynchronize(stop);  
cudaEventElapsedTime (&timerValueGPU, start, stop);  
printf("\n GPU calculation time %f msec\n", timerValueGPU);  
...  
cudaFreeArray (ContS);  
return 0;  
}
```

РЕЗУЛЬТАТЫ

CPU Core2 Duo P8600 2.4 ГГц MS VS 1-ядро

GPU «1» Geforce GT 9650 / GPU «2» Geforce GTX 260

Variants	: Global	Texture
GPU «1» calculation time:	192 ms	103 ms
GPU «2» calculation time:	17 ms	15 ms
CPU calculation time	: 2500 ms	2500 ms
Rate «1»	: 13 x	24 x
Rate «2»	: 147 x	167 x



*Примеры
Интерполяция сеточной функции*

ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

Режим фильтрации и переадресации

Сеточная функция

$$f(x, y) = e^{-\frac{(x-x_0)^2}{a^2} - \frac{(y-y_0)^2}{b^2}},$$

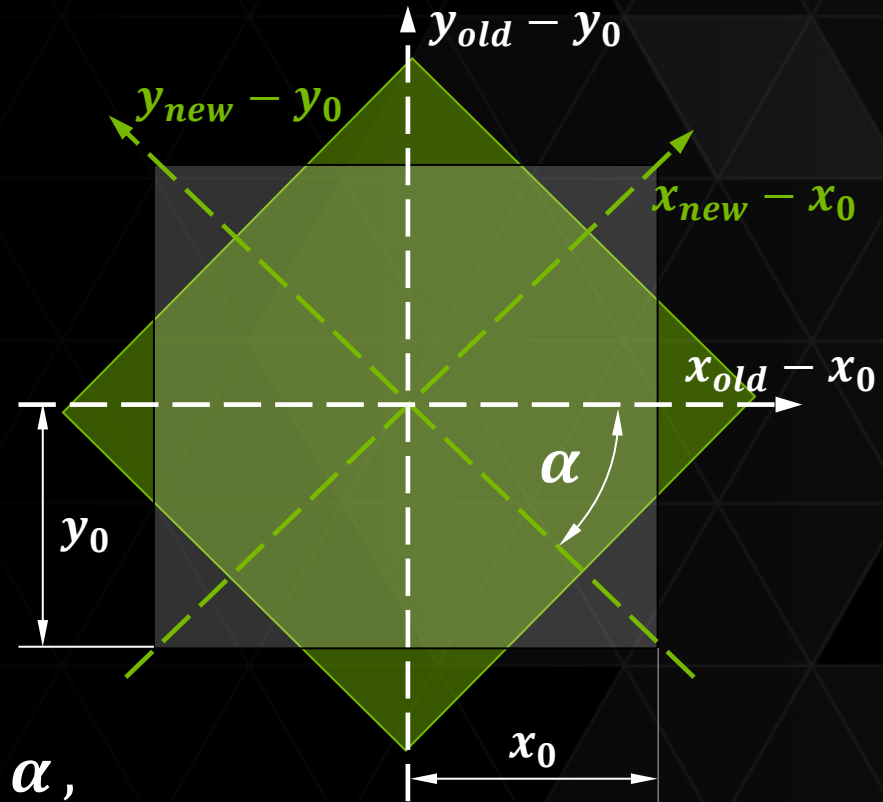
$$n = 0 \dots H - 1, \quad m = 0 \dots W - 1,$$

$$x_m = \frac{m}{W}, \quad y_n = \frac{n}{H}, \quad f_{m,n} = f(x_m, y_n),$$

Поворот и сдвиг системы координат

$$x_{new} - x_0 = (x_{old} - x_0) \cos \alpha - (y_{old} - y_0) \sin \alpha,$$

$$y_{new} - y_0 = (x_{old} - x_0) \sin \alpha + (y_{old} - y_0) \cos \alpha.$$



ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

Функция-ядро «TransformKernel»

```
texture <float, 2, cudaReadModeElementType> texRef;
```

```
__global__ void TransformKernel (float *output, int W, int H, float theta)  
{int x = blockIdx.x * blockDim.x + threadIdx.x;  
  int y = blockIdx.y * blockDim.y + threadIdx.y;
```

```
  float u = x/(float)W; // переход к нормализованным текстурным координатам  
  float v = y/(float)H;
```

```
  u -= 0.5f; v -= 0.5f; // преобразование координат: поворот и сдвиг  
  float tu = u*cosf(theta)-v*sinf(theta)+0.5f;  
  float tv = v*cosf(theta)+u*sinf(theta)+0.5f;
```

```
  output[x+y*W] = tex2D(texRef, tu, tv);
```

```
}
```

ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

Фрагмент функции «main»

```
int main()
{...
    int W = 512; int H = 512;
    int size = W*H; // размер сетки

    unsigned int mem_size = sizeof(float)*size; // выделение памяти на host
    float *h_old = (float*) malloc (mem_size);
    float *h_new = (float*) malloc (mem_size);

    for(n = 0; n < H; n++) // задание изначальной сеточной функции
    {v = n/(float)H-0.5f;
        for(m = 0; m < W; m++)
        {u = m/(float)W-0.5f;
            h_old[m+n*W]= expf(-u*u/(a*a)-v*v/(b*b));
        }
    }
}
```

ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

Фрагмент функции «main»

```
cudaChannelFormatDesc channelDesc =  
    cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);  
cudaArray *cuArray;  
cudaMallocArray(&cuArray, &channelDesc, W, H);  
  
cudaMemcpyToArray (cuArray,0,0,h_old,mem_size,cudaMemcpyHostToDevice);  
// задание параметров текстуры  
texRef.addressMode[0] = cudaAddressModeWrap; // переадресация Wrap по u  
texRef.addressMode[1] = cudaAddressModeWrap; // переадресация Wrap по v  
texRef.filterMode = cudaFilterModeLinear;    // интерполяция Linear  
texRef.normalized = true;                    // нормализованные координаты  
  
cudaBindTextureToArray(texRef, cuArray, channelDesc);  
  
float *output;  
cudaMalloc((void**)&output, mem_size);
```

ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

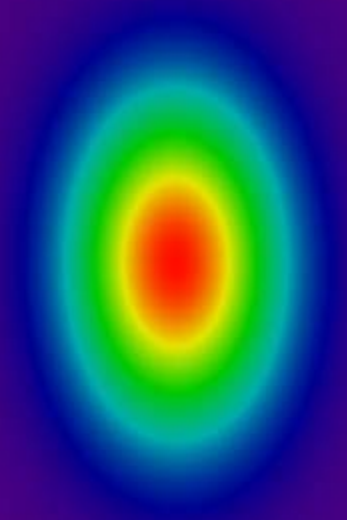
Фрагмент функции «main»

```
dim3 dimBlock (16, 16);  
dim3 dimGrid (W/dimBlock.x, H/dimBlock.y);  
  
TransformKernel <<<dimGrid, dimBlock>>> (output, W, H, angle);  
  
cudaMemcpy (h_new, output, mem_size, cudaMemcpyDeviceToHost);  
  
free (h_old); free (h_new);  
cudaFreeArray (cuArray);  
cudaFree (output);  
  
return 0;  
}
```

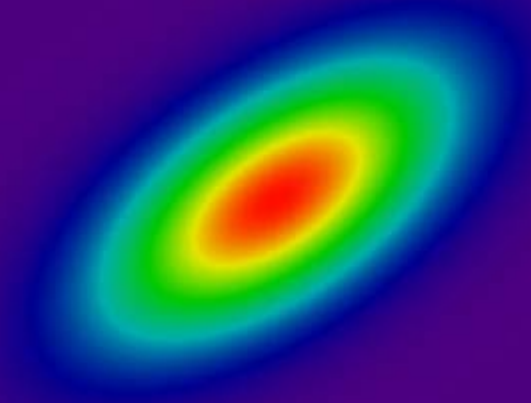
ИНТЕРПОЛЯЦИЯ СЕТОЧНОЙ ФУНКЦИИ

Графики линий уровня функции до поворота «old» и после поворота «new»

old



new





*Примеры
Численное решение СЛАУ*

ПРЕДЫДУЩИЙ ВАРИАНТ

Лекция 4. Функция-ядро «Solve»

```
__global__ void Solve ( double *dA, double *dF,  
                        double *dx0, double *dx1, int N )  
{double aa, sum = 0.;  
  int t = blockIdx.x * blockDim.x + threadIdx.x;  
  
  for ( int j = 0; j < N; j++ )  
  {sum += dA [ t + j * N ] * dx0[j]; // транспонированная матрица  
    if ( j == t ) aa = dA [ t + j * N ];  
  }  
  dx1[t] = dx0[t] + ( dF[t] - sum ) / aa;  
}
```

НОВЫЙ ВАРИАНТ

Функция-ядро «SolveTex»

```
texture <float, 1, cudaReadModeElementType> X0Ref;
```

```
__global__ void SolveTex (float *dA, float *dF, float *dX0, float *dX1, int N)
{
    float aa, sum = 0.0f;
    int t = blockIdx.x * blockDim.x + threadIdx.x;

    for (int j = 0; j < N; j++)
    {
        AA = dA[t+j*N];
        sum += AA*tex1Dfetch( X0Ref, j ); // обращение через linear memory
        if (j == t) aa = AA;
    }
    dX1[t] = x0+(dF[t]-sum)/aa;
}
```

НОВЫЙ ВАРИАНТ

Фрагмент функции «main»

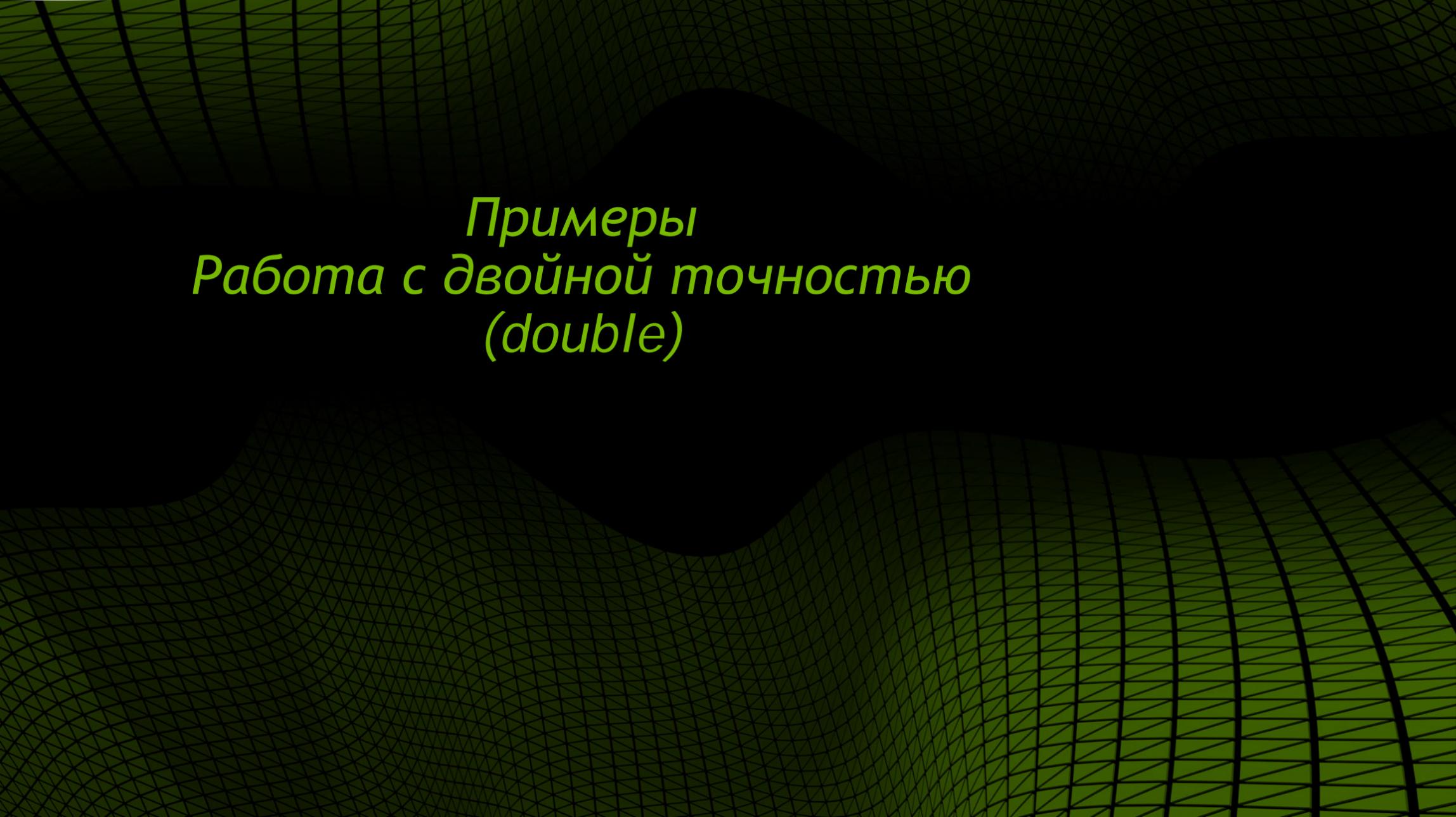
```
int main()
{...
  cudaBindTexture (0, X0Ref, dX0, mem_sizeX);
  cudaMemcpy (dX0,hX0,mem_sizeX,cudaMemcpyHostToDevice);
  ...
  while ( eps > EPS ) // Итерационный процесс
  {SolveTex <<< N_blocks, N_thread >>> (dA,dF,dX0,dX1,N);
    ...
  }
  ...
  return 0;
}
```

РЕЗУЛЬТАТЫ

CPU Core2 Duo P8600 2.4 ГГц MS VS 1-ядро

GPU «1» Geforce GT 9650 / GPU «2» Geforce GTX 260

Variants	: Global	Texture
GPU «1» calculation time:	4400 ms	569 ms
GPU «2» calculation time:	200 ms	200 ms
CPU calculation time	: 3900 ms	3900 ms
Rate «1»	: 0.9 x	6.9 x
Rate «2»	: 19.5 x	19.5x



*Примеры
Работа с двойной точностью
(double)*

ТЕКСТУРНАЯ ПАМЯТЬ

Двойная точность (double)

- ▶ Использовать переменные типа «float2», суммарно дающие размер 64-бита, что совпадает с размером переменной типа «double»
- ▶ Необходимо использовать дополнительную разделяемую память (динамическую), выделяемую на блок нитей, в которую скопировать два подряд идущих слова по 32-бита и считать их потом как одно слово в 64-бит

ТЕКСТУРНАЯ ПАМЯТЬ

Двойная точность (double)), tex1Dfetch

```
int main()
{int size = 2048; // задание размера массива
  dim3 nThreads (256,1,1);
  dim3 nBlocks (size/nThreads.x,1,1);

  int mem_size = sizeof(double) * size;
  double *hA = (double*) malloc (mem_size);
  double *hB = (double*) malloc (mem_size);
  for (int i=0; i < size; i++) hA[i]= sin((double)i);
  double *dA, *dB;
  cudaMalloc ((void**) &dA, mem_size);
  cudaMalloc ((void**) &dB, mem_size);

  cudaMemcpy (dA, hA, mem_size, cudaMemcpyHostToDevice);
  cudaBindTexture (0, TexRef_A, dA, mem_size);
```

ТЕКСТУРНАЯ ПАМЯТЬ

Двойная точность (double)), tex1Dfetch

```
// задание дополнительной разделяемой памяти на блок
int nBytes = sizeof(double)*(nThreads.x);

// запуск функции – ядра с параметром nBytes
TexDouble <<< nBlocks, nThreads, nBytes >>> (dB);
cudaMemcpy (hB, dB, mem_size, cudaMemcpyDeviceToHost);

// вывод результатов на экран
for (int i=0; i < size; i++)
printf ("\n i=%i, hA=%1.16e, hB=%1.16e",i,hA[i],hB[i]);

free (hA); free (hB);
cudaFree (dA);

return 0;
}
```

ВЫЧИСЛЕНИЕ СВЁРТКИ

Двойная точность (double), cudaArray

```
// объявление массивов для 2-х 32 битовых слов/одно 64 битовое
extern __shared__ float2 smem_real32[];
extern __shared__ double smem_real64[];
// объявление текстурной ссылки с типом float2
texture < float2, 2, cudaReadModeElementType> TexRef_A;
...
__global__ void Conv_Tex (double *dConv, int W, int H, int delta)
{
    ...
    // считывание переменной типа float2
    int ind = threadIdx.x + blockIdx.x * blockDim.x;
    smem_real32[ind] = tex2D ( S_TexRef, idx+ix+0.5f, idy+iy+0.5f);
    cov += K*smem_real64[ind]; // запись в глобальную память той же переменной,
                               // но уже как переменную типа double
    ...
}
```

ВЫЧИСЛЕНИЕ СВЁРТКИ

Двойная точность (double), cudaArray

```
int main ()
{ ...
  // создание контейнера в текстурной памяти
  cudaChannelFormatDesc channelDesc =
      cudaCreateChannelDesc(32,32,0,0,cudaChannelFormatKindFloat);
  ...
  // задание дополнительной разделяемой памяти на блок
  int nBytes = sizeof(double)*(nThreads.x * nThreads.y);

  // запуск функции-ядра с параметром nBytes
  Conv_Tex <<< nBlocks, nThreads, nBytes >>> (dConv, W, H, delta);
  ...
  return 0;
}
```