Моделирование физических процессов и визуализация результатов на графических процессорах

Алгоритм вычислений и визуализации.

Программная реализация.

Пример атомной структуры и ее визуализации.

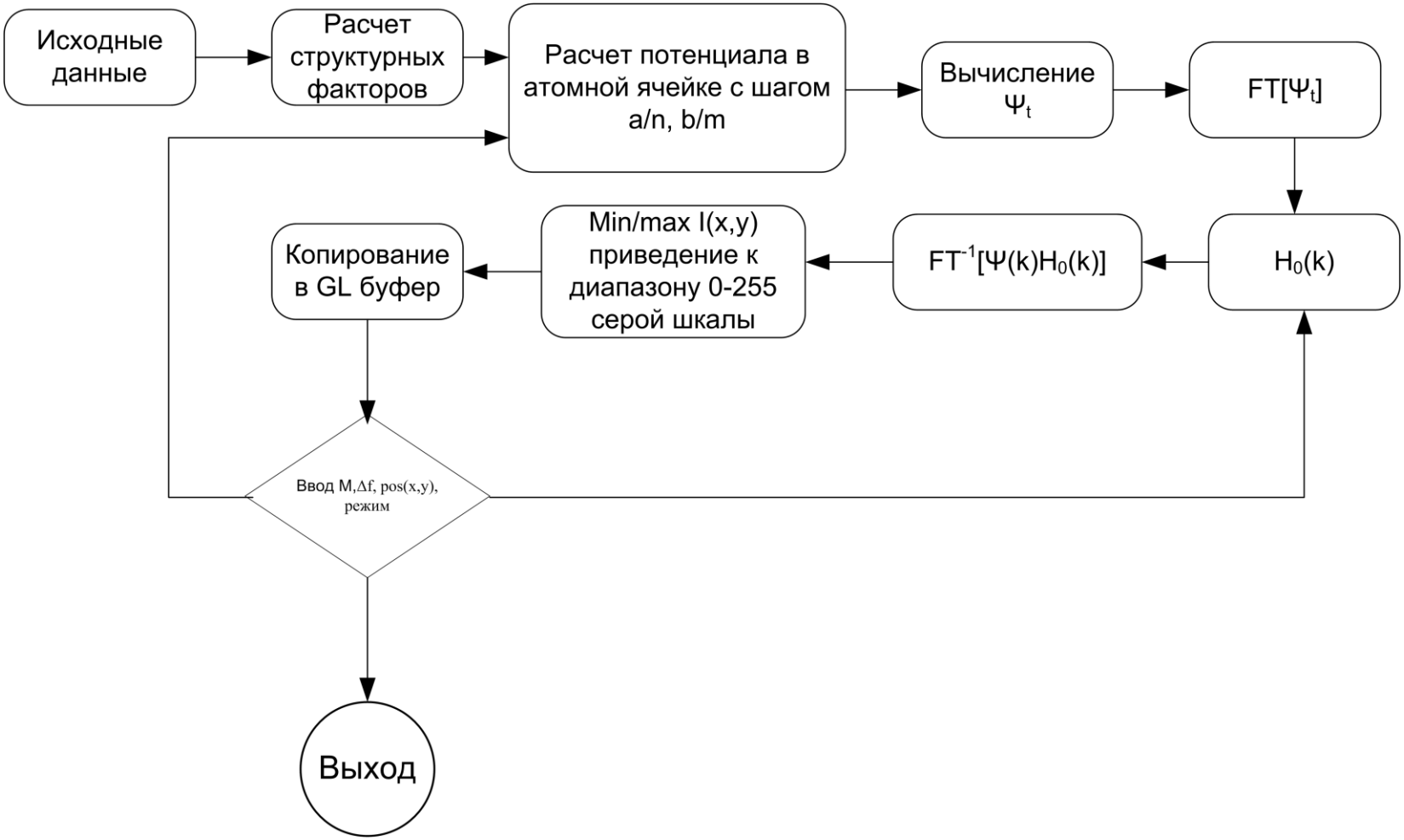
Структура лекции 4

- Алгоритм вычислений и визуализации.
- Программная реализация.
- Пример атомной структуры и ее визуализации.

Алгоритм

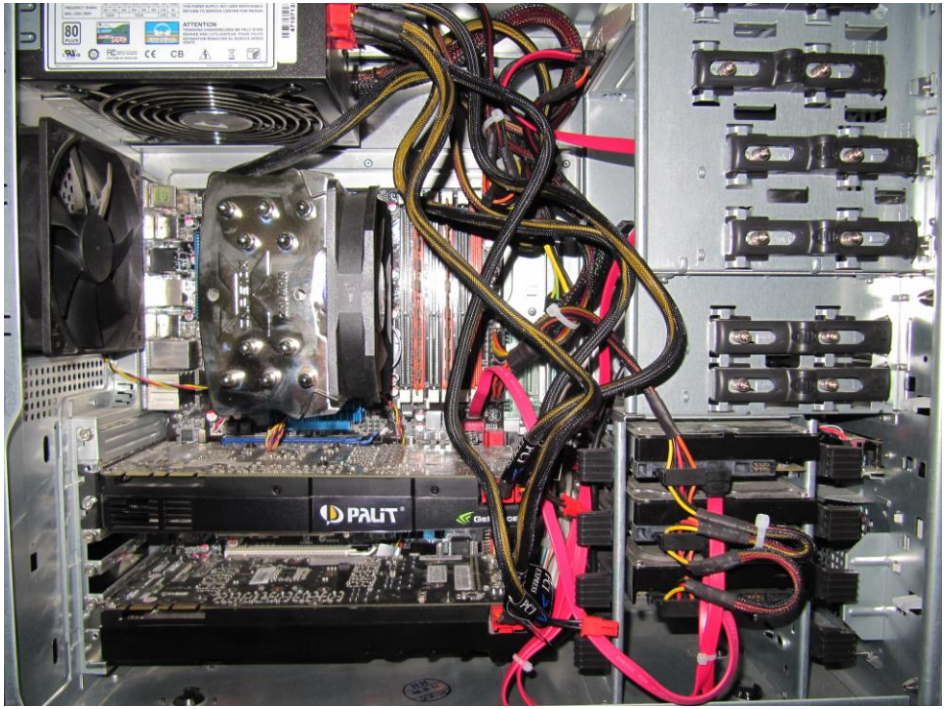
- Вычисление структурных факторов атомов
- Вычисление проекции потенциала и волновой функции
- Расчет передаточной функции
- Нормировка изображения
- Вывод средствами CUDA OpenGL

Блок-схема


Исходные данные
Расчет структурных факторов
Расчет потенциала в атомной ячейке с шагом a/n, b/m
Вычисление Ψ_t
$FT[\Psi_t]$
$H_0(k)$
$FT^{-1}[\Psi(k)H_0(k)]$
Min/max I(x,y) приведение к диапазону 0-255 серой шкалы
Копирование в GL буфер
Ввод M,Δf, pos(x,y), режим
Выход

Железо

- i7960 6 cores/12 thread, 8 Gb RAM
- 2 * Nvidia GTX 580 1.5 Gb, 512 core



Реализация

- Консольное C++ приложение с окном GL видеобуфера, двойная буферизация
- Управление: клавиатура, мышь, горячие клавиши, доп.клавиатура и меню.
- Порядка 3000 строк

Реализация

```
int main(int argc,  char** argv)
{

      if(!InitCUDA()) {
            return 0;
      }

…
//    подготовка консольного окна
…
//    открытие файла с данными
…
// вычисление констант – Ускоряющее напряжение, длинна волны, …
…
//инициализация GL

    initGL(argc, argv);
    cudaGLSetGLDevice( cutGetMaxGflopsDeviceId() );

    initCudaBuffers(image_c,dimx,dimy,nproj);

      initGLBuffers();

    atexit(cleanup);

    glutMainLoop();//основной цикл

    cudaThreadExit();

      return 0;
}
```

Реализация

```
extern "C" void DrawImageC(cufftComplex *d_img,unsigned int *d_result,int dimx,int dimy,int slice){
            cudaMemset(d_result,0,dimx*dimy*sizeof(uint));//обнуление изображения с результатом
            int BLOCK_DIM=16;

            dim3 grid(dimx / BLOCK_DIM, dimy / BLOCK_DIM, 1);// размер сетки в блоках
            dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);// размер блока в потоках
            //копирование Фурье образа исходной волновой функции во временную область
            cudaMemcpy(d_tmp_c,d_img+slice*dimx*dimy,dimx*dimy*sizeof(cufftComplex),cudaMemcpyDeviceToDevice);

            if (ImageOrDifr==1) {//показываем изображение
                  CTF<<<grid,threads>>>(d_tmp_c, dimx, dimy, Cs, Lambda,  Df,delta,
                  ux,uy,ImageOrCTF,H_Cc,tetta,Apperture,AppertureRadius);//Умножаем волновую функцию на передаточную функцию
                  микроскопа
                  if (ImageOrCTF==1) {//изображение
                        QuadrantTileComplex<<<grid,threads>>>(d_tmp_c,dimx,dimy);//переставляем квадранты Фурье-образа
                        cufftExecC2C( plan, d_tmp_c,d_tmp_c,  CUFFT_INVERSE);//вычисляем обратное Фурье преобразование
                        CalculateStatsGPU(d_tmp_c,dimx,dimy,img_min, img_max);//вычисляем мин/макс изображения на GPU
                        } else {//передаточную функцию микроскопа
                        CalculateStatsGPU(d_tmp_c,dimx,dimy,img_min, img_max);
                        img_max*=Contrast;
                        }
                  } else {//показываем дифракцию
                        CalculateStatsGPU(d_tmp_c,dimx,dimy,img_min, img_max);
                        img_max*=Contrast;
                  }
                  //преобразуем комплексные значения в реальные
                  CopyComplexToFloat<<<grid,threads>>>(d_result_tex,d_tmp_c, dimx, dimy,ImageOrCTF,RealCTFOrComplexCTF);
            //копируем в массив для текстуры
                  cutilSafeCall( cudaMemcpyToArray( cu_array, 0, 0, d_result_tex, dimx*dimy*sizeof(float),
                  cudaMemcpyDeviceToDevice));

            //создаем массив(изображение) RGB из текстуры с учетом увеличения и смещения.
      CreateRGBAfromTexture<<<grid,threads>>>(d_result,dimx,dimy,img_min,img_max,ImageOrCTF,RealCTFOrComplexCTF,
      Zoom,cx,cy,tx,ty);

      }
```

Реализация

```
__global__ void CTF(cufftComplex *d_img,int dimx, int dimy,float Cs, float Lambda, float Df,float Delta,float ux, float
uy,unsigned int ImageOrCTF,float H_Cc,float q0,unsigned int Apperture, float AppertureRadius)
      {
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;//координата X на сетке разбиения
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;//координата Y на сетке разбиения
   float CTFS,CTFC,Esct,Alpha,G0,G1,G2,G3,G4,Ar2;
      if((xIndex < dimx) && (yIndex < dimy)){// проверяем не вышли ли за пределы массива
            float k2=((int)xIndex-dimx/2)*ux*((int)xIndex-dimx/2)*ux+((int)yIndex-dimy/2)*uy*((int)yIndex-dimy/2)*uy;
                  G0=PI*Df*Lambda ;
                  G1=PI*0.5f*Cs*__powf(Lambda,3);
                  G2=-0.5f*__powf(PI*Lambda*Delta,2) ;
                  G3=q0*2.0*G1 ;
                  G4=q0*G0 ;
            int index  = xIndex + dimx*yIndex;//индекс элемента массива
            Alpha=(G0+G1*k2)*k2;
            CTFS=-__sinf(Alpha);
            CTFC=__cosf(Alpha);
            Esct=__expf(k2*(G2*k2-(G3*k2+G4)*(G3*k2+G4)));
            if (ImageOrCTF==0) {//показываем передаточную функцию
                  d_img[index].y=CTFS*Esct;
                  d_img[index].x=CTFC*Esct;
                  } else {//умножаем волновую функцию на передаточную функцию
                  d_img[index].y*=CTFS*Esct;
                  d_img[index].x*=CTFC*Esct;
                  }
            if ((Apperture==1) && (AppertureRadius*AppertureRadius<k2)){//если есть аппертура то обрезаем все что больше
                        d_img[index].x=0.0f;
                        d_img[index].y=0.0f;
                  }

            __syncthreads();//синхронизуем потоки
            }
      }
```

Реализация

```
__global__ void CreateRGBAfromTexture(uint *d_result, int dimx,int dimy,float imin, float imax, unsigned int
ImageorCTF,unsigned int RealCTFOrComplexCTF,float Zoom,float cx,float cy,float tx,float ty){

   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;//координата X
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;//координата Y
      if((xIndex < dimx) && (yIndex < dimy)){
                  int index  = xIndex + dimx*yIndex;
                  float x,y;
                  x=(float)xIndex;
                  y=(float)yIndex;
                  x=(x-cx)/Zoom+cx+tx;//учет увеличения и смещения
                  y=(y-cy)/Zoom+cy+ty; //учет увеличения и смещения
                  …
                  if (ImageorCTF==1) {
                  d_result[index]=rgbaFloatToInt(__fdividef((tex2D(tex_image_2d,x+0.5f,y+0.5f)-imin),(imax-imin)));
                  } else {
                  d_result[index]= rgbaFloatToIntcolor(__fdividef(tex2D(tex_image_2d,x+0.5f,y+0.5f)-imin,(imax-imin)));
                        }
            }
      }


__device__ uint rgbaFloatToIntcolor(float rgba)
{
      uint color=0;
      if (rgba >0.0f) {
            color=(uint(0*255)<<24) | (uint(rgba*255)<<16) | (uint(0*255)<<8) | uint(0*255);
            } else {
                  color=(uint(0*255)<<24) | (uint(0*255)<<16) | (uint(0*255)<<8) | uint(-rgba*255);
            }
    return color;
}
```

Реализация

```
float GPU_max(int  n,int  numThreads,int  numBlocks,int  maxThreads,int  maxBlocks,cufftComplex* d_idata,float* d_odata){


      float gpu_result = 0;

    gpu_result = 0;

    cudaThreadSynchronize();
        // execute the kernel
        kernel_max1(n, numThreads, numBlocks,  d_idata, d_odata)
            // Первый вызов вычисляет модуль комплексных величин и ищет максимум
            // в d_odata частичные максимумы в пределах блоков, всего значений numBlocks
        // check if kernel execution generated an error
        cutilCheckMsg("Kernel execution failed");

            // sum partial block sums on GPU
            int s=numBlocks;
            while(s > 1) // до тех пор пока блоков больше 1, вычисляем частичные паксимумы в пределах блоков
            {
                int threads = 0, blocks = 0;
                  // вычисляем количество блоков в сетке и потоков в блоке при размере массива s
                getNumBlocksAndThreads( s, maxBlocks, maxThreads, blocks, threads);

                kernel_max(s, threads, blocks, d_odata, d_odata);
                  // последующие вызовы вичисления максимально значения «по месту»
                    s = (s + (threads*2-1)) / (threads*2);
                  // изменяем количество оставшихся значений в массиве
            }


        cudaThreadSynchronize();

        // copy final sum from device to host  копируем результат из устройства в память компьютера
        cutilSafeCallNoSync( cudaMemcpy( &gpu_result, d_odata, sizeof(float), cudaMemcpyDeviceToHost) );

    return gpu_result;
      }
```

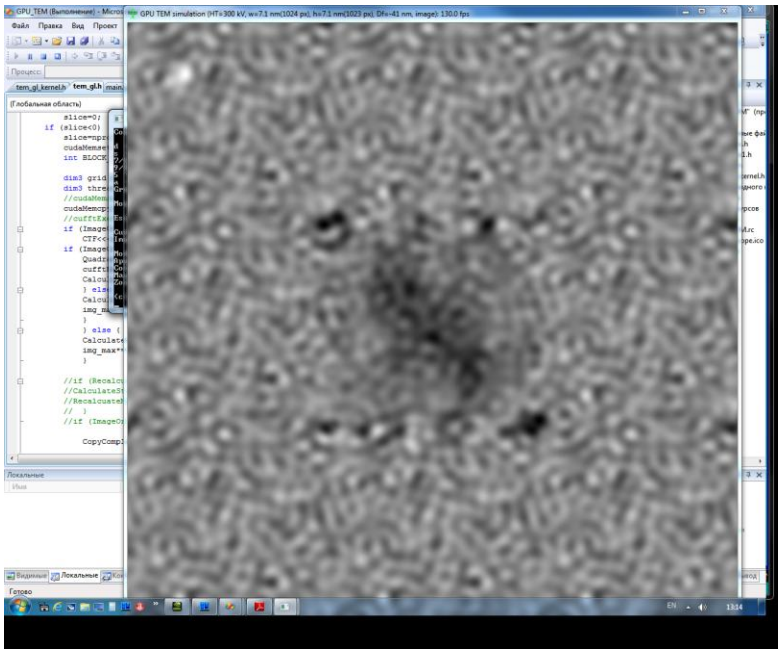
Реализация

```
float GPU_max(int  n,int  numThreads,int  numBlocks,int  maxThreads,int  maxBlocks,cufftComplex* d_idata,float* d_odata){
      float gpu_result = 0;
      gpu_result = 0;

    cudaThreadSynchronize();
        // execute the kernel
        kernel_max1(n, numThreads, numBlocks,  d_idata, d_odata)
            // Первый вызов вычисляет модуль комплексных величин и ищет максимум
            // в d_odata частичные максимумы в пределах блоков, всего значений numBlocks
        cutilCheckMsg("Kernel execution failed");
            int s=numBlocks;
            while(s > 1) // до тех пор пока блоков больше 1, вычисляем частичные паксимумы в пределах блоков
            {
                int threads = 0, blocks = 0;
                  // вычисляем количество блоков в сетке и потоков в блоке при размере массива s
                getNumBlocksAndThreads( s, maxBlocks, maxThreads, blocks, threads);
                kernel_max(s, threads, blocks, d_odata, d_odata);
                  // последующие вызовы вичисления максимально значения «по месту»
                    s = (s + (threads*2-1)) / (threads*2);
                  // изменяем количество оставшихся значений в массиве
            }
        cudaThreadSynchronize();
        // copy final sum from device to host  копируем результат из устройства в память компьютера
        cutilSafeCallNoSync( cudaMemcpy( &gpu_result, d_odata, sizeof(float), cudaMemcpyDeviceToHost) );
    return gpu_result;
      }

extern "C" void kernel_max(int size, int threads, int blocks, float *d_idata, float *d_odata){
   dim3 dimBlock(threads, 1, 1);
   dim3 dimGrid(blocks, 1, 1);
   int smemSize = (threads <= 32) ? 2 * threads * sizeof(float) : threads * sizeof(float);
   max6<<< dimGrid, dimBlock, smemSize >>>(threads, d_idata, d_odata, size);
      }
```

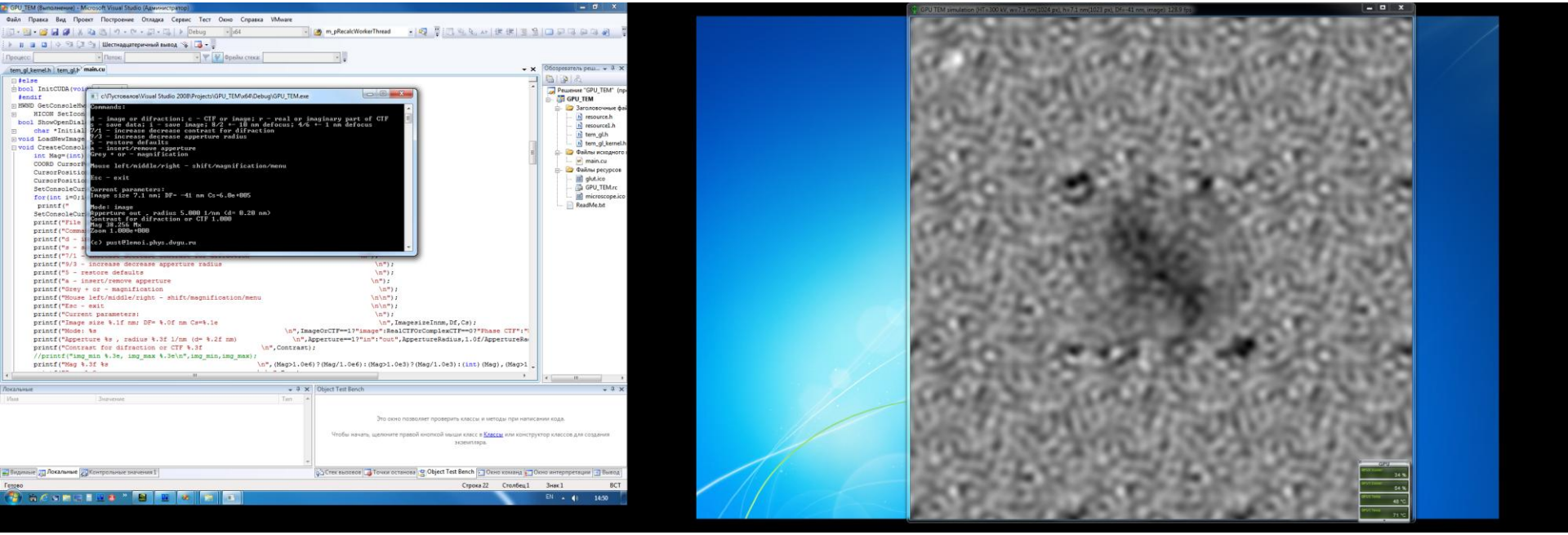
Реализация

```
__global__ void max6(unsigned int blockSize,float *g_idata, float *g_odata, unsigned int n)
{
    volatile float *sdata = SharedMemory<float>(); //объявляем разделяемую память
      // первый шаг поиска максимума читаем из глобальнйо памяти и пишем в разделяемую память
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;
    float mySum = -1e+30f;
      // ищем максимальный элемент в потоке, число элементов определяется количеством активных блоков (gridDim).
      // большее количество блоков приводит к меньшему числу элементов на поток
    while (i < n) {
       mySum = fmaxf(g_idata[i],mySum);
        // ищем максимум среди элементов с индексом blockIdx.x*blockSize*2 + threadIdx.x и    blockIdx.x*blockSize*2 +
threadIdx.x+  blockSize с шагом  gridSize
        if ( i + blockSize < n)
            mySum = fmaxf(g_idata[i+blockSize],mySum);
        i += gridSize; }
    // каждый максимум из потока пишем в разделяемую память
    sdata[tid] = mySum;
    __syncthreads();// ждем пока все потоки не закончат вычисления
    // ищем максимум в размедяемой памяти
    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = fmaxf(mySum,sdata[tid + 256]); } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = fmaxf(mySum,sdata[tid + 128]); } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = fmaxf(mySum,sdata[tid +  64]); } __syncthreads(); }

#ifndef __DEVICE_EMULATION__ // проверяется только в случае реального устройства, а не его эмуляции
    if (tid < 32)
#endif
    {
        if (blockSize >=  64) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid + 32]); EMUSYNC; }
        if (blockSize >=  32) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid + 16]); EMUSYNC; }
        if (blockSize >=  16) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid +  8]); EMUSYNC; }
        if (blockSize >=   8) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid +  4]); EMUSYNC; }
        if (blockSize >=   4) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid +  2]); EMUSYNC; }
        if (blockSize >=   2) { sdata[tid] = mySum = fmaxf(mySum , sdata[tid +  1]); EMUSYNC; }
    }
    // пишем результат в глобальную память с индексом = номеру блока
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```

Внешний вид



Внешний вид


GPU_TEM (Выполнение) - Microsoft Visual Studio (Администратор)
c:\Пустовалов\Visual Studio 2008\Projects\GPU_TEM\x64\Debug\GPU_TEM.exe
Commands:
d - image or difraction; c - CTF or image; r - real or imaginary part of CTF
s - save data; i - save image; 0/2 +- 10 nm defocus; 4/6 +- 1 nm defocus
7/1 - increase decrease contrast for difraction
9/3 - increase decrease apperture radius
5 - restore defaults
a - insert/remove apperture
Grey + or - magnification
Mouse left/middle/right - shift/magnification/menu
Esc - exit
Current parameters:
Image size 7.1 nm; DF= -41 nm Cs=6.8e+005
Mode: image
Apperture out , radius 5.000 1/nm (d= 0.20 nm)
Contrast for difraction or CTF 1.000
Mag 38.256 MX
Zoom 1.000e+000
(c) pust@lenoi.phys.dvgu.ru
GPU_TEM simulation

Видео