数字媒体技术 实验二 CUDA的使用

姓名：朱本金

班级：2014级软件工程卓越班

学号：201430615165

一： bilateral filtering using CUDA and CPU

\* 双边滤波是一种保边缘、非线性平滑滤波器，这种滤波器有 3 个参数：gaussian delta, euclidean delta and iterations

\* 随着 euclidean delta 值增大，大多数纹理信息会被过滤掉，然后轮廓信息则会被保留。当 euclidean delta 接近无穷大时，filter 就变成了一个

\* 普通的高斯滤波器。

\* 随着 gaussian delta 增大，图像细致的纹理会变得越来越模糊。

\* 多次迭代有将一幅图片扁平化的效果，这并不会模糊边缘，而是产生一种卡通画的效果。

其中 欧氏距离计算方法如下：

//Euclidean Distance (x, y, d) = exp((|x - y| / d)^2 / 2)

\_\_device\_\_ float euclideanLen(float4 a, float4 b, float d)

{

float mod = (b.x - a.x) \* (b.x - a.x) +

(b.y - a.y) \* (b.y - a.y) +

(b.z - a.z) \* (b.z - a.z);

return \_\_expf(-mod / (2.f \* d \* d));

}

核函数如下所示：

\_\_global\_\_ void d\_bilateral\_filter(uint\* od, int w, int h, float e\_d, int r) {

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

if (x >= w || y >> h)

return;

float sum = 0.0f;

float factor;

float4 t = {0.f, 0.f, 0.f, 0.f};

//tex2D - performs a texture lookup in a given 2D sampler and,

//in some cases, a shadow comparison. May also use pre computed derivatives if those are provided.

float4 center = tex2D(rgbaTex, x, y);

for (int i = -r; i <= r; i++) {

for (int j = -r, j <= r; j++) {

float4 curPix = tex2D(rgbaTex, x+j, y+i);

factor = cGaussian[i+r] \* cGaussian[j+r] \* //domian factor

euclideanLen[curPix, center, e\_d]; //range factor

t += factor \* curPix;

sum += factor;

}

}

od[y\*w + x] = rgbaFloatToInt(t/sum);

}

完整的彩色图像双边滤波如下所示：

\* 双边滤波器是一个非线性滤波器，它是一个 range filter 和一个 domain filter 的混合体。

\* range filter 保留清晰的边缘，而 domain filter 用于过滤噪声。

\* 一个像素点的 intensity value 被它周围的点的加权值替代。

\*

\* 其中，权值银子通过 domain filter 的 component（使用高斯分布作为空间分布）） 和 range filter 的 component（中心像素和当前邻域内像素的欧氏距离）的乘积计算得到

\* 由于这个过程是非线性的，所以采用的是逐点运算的方式。

\*

\* 纹理 Texture 自动钳点到图像边缘。1D 高斯数组映射为 1D 的纹理，而不是使用 shared memory，因为那将会导致严重的 bank conflict 问题。

\*

\* 线程是纵向执行的，因为输出是合并的（coalesced）

\*

\* 参数：

\* od - pointer to output data in global memory

\* d\_f - pointer to the 1D gaussian array

\* e\_d - eculidean delta

\* w - image width

\* h - image height

\* r - filter radius

// RGBA version

extern "C"

double bilateralFilterRGBA(uint \*dDest,

int width, int height,

float e\_d, int radius, int iterations,

StopWatchInterface \*timer)

{

// var for kernel computation timing

double dKernelTime;

// Bind the array to the texture

cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar4>();

checkCudaErrors(cudaBindTexture2D(0, rgbaTex, dImage, desc, width, height, pitch));

for (int i=0; i<iterations; i++)

{

// sync host and start kernel computation timer

dKernelTime = 0.0;

checkCudaErrors(cudaDeviceSynchronize());

sdkResetTimer(&timer);

dim3 gridSize((width + 16 - 1) / 16, (height + 16 - 1) / 16);

dim3 blockSize(16, 16);

d\_bilateral\_filter<<< gridSize, blockSize>>>(

dDest, width, height, e\_d, radius);

// sync host and stop computation timer

checkCudaErrors(cudaDeviceSynchronize());

dKernelTime += sdkGetTimerValue(&timer);

if (iterations > 1)

{

// copy result back from global memory to array

checkCudaErrors(cudaMemcpy2D(dTemp, pitch, dDest, sizeof(int)\*width,

sizeof(int)\*width, height, cudaMemcpyDeviceToDevice));

checkCudaErrors(cudaBindTexture2D(0, rgbaTex, dTemp, desc, width, height, pitch));

}

}

return ((dKernelTime/1000.)/(double)iterations);

}

作为对比 CPU版本的核心部分代码如下所示：

void bilateralFilterGold(unsigned int \*pSrc,

unsigned int \*pDest,

float e\_d,

int w, int h, int r)

{

float4 \*hImage = new float4[w \* h];

float domainDist, colorDist, factor;

for (int y = 0; y < h; y++)

{

for (int x = 0; x < w; x++)

{

hImage[y \* w + x] = hrgbaIntToFloat(pSrc[y \* w + x]);

}

}

for (int y = 0; y < h; y++)

{

for (int x = 0; x < w; x++)

{

float4 t(0.0f);

float sum = 0.0f;

for (int i = -r; i <= r; i++)

{

int neighborY = y + i;

//clamp the neighbor pixel, prevent overflow

if (neighborY < 0)

{

neighborY = 0;

}

else if (neighborY >= h)

{

neighborY = h - 1;

}

for (int j = -r; j <= r; j++)

{

domainDist = gaussian[r + i] \* gaussian[r + j];

//clamp the neighbor pixel, prevent overflow

int neighborX = x + j;

if (neighborX < 0)

{

neighborX = 0;

}

else if (neighborX >= w)

{

neighborX = w - 1;

}

colorDist = heuclideanLen(hImage[neighborY \* w + neighborX], hImage[y \* w + x], e\_d);

factor = domainDist \* colorDist;

sum += factor;

t = add4(t, mul(factor, hImage[neighborY \* w + neighborX]));

}

}

pDest[y \* w + x] = hrgbaFloatToInt(mul(1 / sum, t));

}

}

delete[] hImage;

}

下面展示在 kernel 是 radius = 1 和 7 时 不同的滤波效果：



二： 图像去噪 using GPU

本次使用 NLM2 算法进行图片去噪，代码如下：

////////////////////////////////////////////////////////////////////////////////

// 0 1 2 3 4 5 6 7

// 0 + . . . . . . .

// 1 . . . . . . . .

// 2 . . . . . . . .

// 3 . . . \* . . . .

// 4 . . . . . . . .

// 5 . . . . . . . .

// 6 . . . . . . . .

// 7 . . . . . . . .

//

// \* - Base point for every thread, + - pixel around which ColorDistance is computed

// The idea behind this method:

// - Every thread in a 8x8 block computes just one ColorDistance

// - It is saved in the weights array that is shared across the threads

// - Threads are synced

// - For every pixel inside the block weights are considered to be constants

////////////////////////////////////////////////////////////////////////////////

\_\_global\_\_ void NLM2(

TColor \*dst,

int imageW,

int imageH,

float Noise,

float lerpC

)

{

//Weights cache

\_\_shared\_\_ float fWeights[BLOCKDIM\_X \* BLOCKDIM\_Y];

const int ix = blockDim.x \* blockIdx.x + threadIdx.x;

const int iy = blockDim.y \* blockIdx.y + threadIdx.y;

//Add half of a texel to always address exact texel centers

const float x = (float)ix + 0.5f;

const float y = (float)iy + 0.5f;

const float cx = blockDim.x \* blockIdx.x + NLM\_WINDOW\_RADIUS + 0.5f;

const float cy = blockDim.x \* blockIdx.y + NLM\_WINDOW\_RADIUS + 0.5f;

if (ix < imageW && iy < imageH)

{

//Find color distance from current texel to the center of NLM window

float weight = 0;

for (float n = -NLM\_BLOCK\_RADIUS; n <= NLM\_BLOCK\_RADIUS; n++)

for (float m = -NLM\_BLOCK\_RADIUS; m <= NLM\_BLOCK\_RADIUS; m++)

weight += vecLen(

tex2D(texImage, cx + m, cy + n),

tex2D(texImage, x + m, y + n)

);

//Geometric distance from current texel to the center of NLM window

float dist =

(threadIdx.x - NLM\_WINDOW\_RADIUS) \* (threadIdx.x - NLM\_WINDOW\_RADIUS) +

(threadIdx.y - NLM\_WINDOW\_RADIUS) \* (threadIdx.y - NLM\_WINDOW\_RADIUS);

//Derive final weight from color and geometric distance

weight = \_\_expf(-(weight \* Noise + dist \* INV\_NLM\_WINDOW\_AREA));

//Write the result to shared memory

fWeights[threadIdx.y \* BLOCKDIM\_X + threadIdx.x] = weight;

//Wait until all the weights are ready

\_\_syncthreads();

//Normalized counter for the NLM weight threshold

float fCount = 0;

//Total sum of pixel weights

float sumWeights = 0;

//Result accumulator

float3 clr = {0, 0, 0};

int idx = 0;

//Cycle through NLM window, surrounding (x, y) texel

for (float i = -NLM\_WINDOW\_RADIUS; i <= NLM\_WINDOW\_RADIUS + 1; i++)

for (float j = -NLM\_WINDOW\_RADIUS; j <= NLM\_WINDOW\_RADIUS + 1; j++)

{

//Load precomputed weight

float weightIJ = fWeights[idx++];

//Accumulate (x + j, y + i) texel color with computed weight

float4 clrIJ = tex2D(texImage, x + j, y + i);

clr.x += clrIJ.x \* weightIJ;

clr.y += clrIJ.y \* weightIJ;

clr.z += clrIJ.z \* weightIJ;

//Sum of weights for color normalization to [0..1] range

sumWeights += weightIJ;

//Update weight counter, if NLM weight for current window texel

//exceeds the weight threshold

fCount += (weightIJ > NLM\_WEIGHT\_THRESHOLD) ? INV\_NLM\_WINDOW\_AREA : 0;

}

//Normalize result color by sum of weights

sumWeights = 1.0f / sumWeights;

clr.x \*= sumWeights;

clr.y \*= sumWeights;

clr.z \*= sumWeights;

//Choose LERP quotient basing on how many texels

//within the NLM window exceeded the weight threshold

float lerpQ = (fCount > NLM\_LERP\_THRESHOLD) ? lerpC : 1.0f - lerpC;

//Write final result to global memory

float4 clr00 = tex2D(texImage, x, y);

clr.x = lerpf(clr.x, clr00.x, lerpQ);

clr.y = lerpf(clr.y, clr00.y, lerpQ);

clr.z = lerpf(clr.z, clr00.z, lerpQ);

dst[imageW \* iy + ix] = make\_color(clr.x, clr.y, clr.z, 0);

}

}

extern "C"

void cuda\_NLM2(

TColor \*d\_dst,

int imageW,

int imageH,

float Noise,

float LerpC

)

{

dim3 threads(BLOCKDIM\_X, BLOCKDIM\_Y);

dim3 grid(iDivUp(imageW, BLOCKDIM\_X), iDivUp(imageH, BLOCKDIM\_Y));

NLM2<<<grid, threads>>>(d\_dst, imageW, imageH, Noise, LerpC);

}

效果如下图所示：

