```
/*
 * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#ifndef _MARCHING_CUBES_KERNEL_CU_
#define _MARCHING_CUBES_KERNEL_CU_

#include <stdio.h>
#include <string.h>
#include <helper_cuda.h>    // includes for helper CUDA functions
#include <helper_math.h>
#include <cuda_runtime_api.h>
#include <thrust/device_vector.h>
#include <thrust/scan.h>

#include "defines.h"
#include "tables.h"

// textures containing look-up tables
texture<uint, 1, cudaReadModeElementType> edgeTex;
texture<uint, 1, cudaReadModeElementType> triTex;
texture<uint, 1, cudaReadModeElementType> numVertsTex;

// volume data
texture<uchar, 1, cudaReadModeNormalizedFloat> volumeTex;

extern "C"
void allocateTextures(uint **d_edgeTable, uint **d_triTable,  uint **d_numVertsTable)
{
    checkCudaErrors(cudaMalloc((void **) d_edgeTable, 256*sizeof(uint)));
    checkCudaErrors(cudaMemcpy((void *)*d_edgeTable, (void *)edgeTable, 256*sizeof(uint), cudaMemcpyHostToDevice));
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindUnsigned);
    checkCudaErrors(cudaBindTexture(0, edgeTex, *d_edgeTable, channelDesc));

    checkCudaErrors(cudaMalloc((void **) d_triTable, 256*16*sizeof(uint)));
    checkCudaErrors(cudaMemcpy((void *)*d_triTable, (void *)triTable, 256*16*sizeof(uint), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaBindTexture(0, triTex, *d_triTable, channelDesc));

    checkCudaErrors(cudaMalloc((void **) d_numVertsTable, 256*sizeof(uint)));
    checkCudaErrors(cudaMemcpy((void *)*d_numVertsTable, (void *)numVertsTable, 256*sizeof(uint), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaBindTexture(0, numVertsTex, *d_numVertsTable, channelDesc));
}

extern "C"
void bindVolumeTexture(uchar *d_volume)
{
    // bind to linear texture
    checkCudaErrors(cudaBindTexture(0, volumeTex, d_volume, cudaCreateChannelDesc(8, 0, 0, 0, cudaChannelFormatKindUnsigned)));
}

// an interesting field function
__device__
float tangle(float x, float y, float z)
{
    x *= 3.0f;
    y *= 3.0f;
    z *= 3.0f;
    return (x*x*x*x - 5.0f*x*x +y*y*y*y - 5.0f*y*y +z*z*z*z - 5.0f*z*z + 11.8f) * 0.2f + 0.5f;
}

// evaluate field function at point
__device__
float fieldFunc(float3 p)
{
    return tangle(p.x, p.y, p.z);
}

// evaluate field function at a point
// returns value and gradient in float4
__device__
float4 fieldFunc4(float3 p)
{
    float v = tangle(p.x, p.y, p.z);
    const float d = 0.001f;
    float dx = tangle(p.x + d, p.y, p.z) - v;
    float dy = tangle(p.x, p.y + d, p.z) - v;
    float dz = tangle(p.x, p.y, p.z + d) - v;
    return make_float4(dx, dy, dz, v);
}

// sample volume data set at a point
__device__
float sampleVolume(uchar *data, uint3 p, uint3 gridSize)
{
    p.x = min(p.x, gridSize.x - 1);
    p.y = min(p.y, gridSize.y - 1);
    p.z = min(p.z, gridSize.z - 1);
    uint i = (p.z*gridSize.x*gridSize.y) + (p.y*gridSize.x) + p.x;
    //    return (float) data[i] / 255.0f;
    return tex1Dfetch(volumeTex, i);
}

// compute position in 3d grid from 1d index
// only works for power of 2 sizes
__device__
uint3 calcGridPos(uint i, uint3 gridSizeShift, uint3 gridSizeMask)
{
    uint3 gridPos;
    gridPos.x = i & gridSizeMask.x;
    gridPos.y = (i >> gridSizeShift.y) & gridSizeMask.y;
    gridPos.z = (i >> gridSizeShift.z) & gridSizeMask.z;
    return gridPos;
}

// classify voxel based on number of vertices it will generate
// one thread per voxel
__global__ void
classifyVoxel(uint *voxelVerts, uint *voxelOccupied, uchar *volume,
              uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask, uint numVoxels,
              float3 voxelSize, float isoValue)
{
    uint blockId = __mul24(blockIdx.y, gridDim.x) + blockIdx.x;
    uint i = __mul24(blockId, blockDim.x) + threadIdx.x;

    uint3 gridPos = calcGridPos(i, gridSizeShift, gridSizeMask);

    // read field values at neighbouring grid vertices
#if SAMPLE_VOLUME
    float field[8];
    field[0] = sampleVolume(volume, gridPos, gridSize);
    field[1] = sampleVolume(volume, gridPos + make_uint3(1, 0, 0), gridSize);
    field[2] = sampleVolume(volume, gridPos + make_uint3(1, 1, 0), gridSize);
    field[3] = sampleVolume(volume, gridPos + make_uint3(0, 1, 0), gridSize);
    field[4] = sampleVolume(volume, gridPos + make_uint3(0, 0, 1), gridSize);
    field[5] = sampleVolume(volume, gridPos + make_uint3(1, 0, 1), gridSize);
    field[6] = sampleVolume(volume, gridPos + make_uint3(1, 1, 1), gridSize);
    field[7] = sampleVolume(volume, gridPos + make_uint3(0, 1, 1), gridSize);
#else
    float3 p;
    p.x = -1.0f + (gridPos.x * voxelSize.x);
    p.y = -1.0f + (gridPos.y * voxelSize.y);
    p.z = -1.0f + (gridPos.z * voxelSize.z);

    float field[8];
    field[0] = fieldFunc(p);
    field[1] = fieldFunc(p + make_float3(voxelSize.x, 0, 0));
    field[2] = fieldFunc(p + make_float3(voxelSize.x, voxelSize.y, 0));
    field[3] = fieldFunc(p + make_float3(0, voxelSize.y, 0));
    field[4] = fieldFunc(p + make_float3(0, 0, voxelSize.z));
    field[5] = fieldFunc(p + make_float3(voxelSize.x, 0, voxelSize.z));
    field[6] = fieldFunc(p + make_float3(voxelSize.x, voxelSize.y, voxelSize.z));
    field[7] = fieldFunc(p + make_float3(0, voxelSize.y, voxelSize.z));
#endif

    // calculate flag indicating if each vertex is inside or outside isosurface
    uint cubeindex;
    cubeindex =  uint(field[0] < isoValue);
    cubeindex += uint(field[1] < isoValue)*2;
    cubeindex += uint(field[2] < isoValue)*4;
    cubeindex += uint(field[3] < isoValue)*8;
    cubeindex += uint(field[4] < isoValue)*16;
    cubeindex += uint(field[5] < isoValue)*32;
    cubeindex += uint(field[6] < isoValue)*64;
    cubeindex += uint(field[7] < isoValue)*128;

    // read number of vertices from texture
    uint numVerts = tex1Dfetch(numVertsTex, cubeindex);

    if (i < numVoxels)
    {
        voxelVerts[i] = numVerts;
        voxelOccupied[i] = (numVerts > 0);
    }
}

extern "C" void
launch_classifyVoxel(dim3 grid, dim3 threads, uint *voxelVerts, uint *voxelOccupied, uchar *volume,
                     uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask, uint numVoxels,
                     float3 voxelSize, float isoValue)
{
    // calculate number of vertices need per voxel
    classifyVoxel<<<grid, threads>>>(voxelVerts, voxelOccupied, volume,
                                     gridSize, gridSizeShift, gridSizeMask,
                                     numVoxels, voxelSize, isoValue);
    getLastCudaError("classifyVoxel failed");
}


// compact voxel array
__global__ void
compactVoxels(uint *compactedVoxelArray, uint *voxelOccupied, uint *voxelOccupiedScan, uint numVoxels)
{
    uint blockId = __mul24(blockIdx.y, gridDim.x) + blockIdx.x;
    uint i = __mul24(blockId, blockDim.x) + threadIdx.x;

    if (voxelOccupied[i] && (i < numVoxels))
    {
        compactedVoxelArray[ voxelOccupiedScan[i] ] = i;
    }
}

extern "C" void
launch_compactVoxels(dim3 grid, dim3 threads, uint *compactedVoxelArray, uint *voxelOccupied, uint *voxelOccupiedScan, uint numVoxels)
{
    compactVoxels<<<grid, threads>>>(compactedVoxelArray, voxelOccupied,
                                     voxelOccupiedScan, numVoxels);
    getLastCudaError("compactVoxels failed");
}

// compute interpolated vertex along an edge
__device__
float3 vertexInterp(float isolevel, float3 p0, float3 p1, float f0, float f1)
{
    float t = (isolevel - f0) / (f1 - f0);
    return lerp(p0, p1, t);
}

// compute interpolated vertex position and normal along an edge
__device__
void vertexInterp2(float isolevel, float3 p0, float3 p1, float4 f0, float4 f1, float3 &p, float3 &n)
{
    float t = (isolevel - f0.w) / (f1.w - f0.w);
    p = lerp(p0, p1, t);
    n.x = lerp(f0.x, f1.x, t);
    n.y = lerp(f0.y, f1.y, t);
    n.z = lerp(f0.z, f1.z, t);
    //    n = normalize(n);
}

// generate triangles for each voxel using marching cubes
// interpolates normals from field function
__global__ void
generateTriangles(float4 *pos, float4 *norm, uint *compactedVoxelArray, uint *numVertsScanned,
                  uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask,
                  float3 voxelSize, float isoValue, uint activeVoxels, uint maxVerts)
{
    uint blockId = __mul24(blockIdx.y, gridDim.x) + blockIdx.x;
    uint i = __mul24(blockId, blockDim.x) + threadIdx.x;

    if (i > activeVoxels - 1)
    {
        // can't return here because of syncthreads()
        i = activeVoxels - 1;
    }

#if SKIP_EMPTY_VOXELS
    uint voxel = compactedVoxelArray[i];
#else
    uint voxel = i;
#endif

    // compute position in 3d grid
    uint3 gridPos = calcGridPos(voxel, gridSizeShift, gridSizeMask);

    float3 p;
    p.x = -1.0f + (gridPos.x * voxelSize.x);
    p.y = -1.0f + (gridPos.y * voxelSize.y);
    p.z = -1.0f + (gridPos.z * voxelSize.z);

    // calculate cell vertex positions
    float3 v[8];
    v[0] = p;
    v[1] = p + make_float3(voxelSize.x, 0, 0);
    v[2] = p + make_float3(voxelSize.x, voxelSize.y, 0);
    v[3] = p + make_float3(0, voxelSize.y, 0);
    v[4] = p + make_float3(0, 0, voxelSize.z);
    v[5] = p + make_float3(voxelSize.x, 0, voxelSize.z);
    v[6] = p + make_float3(voxelSize.x, voxelSize.y, voxelSize.z);
    v[7] = p + make_float3(0, voxelSize.y, voxelSize.z);

    // evaluate field values
    float4 field[8];
    field[0] = fieldFunc4(v[0]);
    field[1] = fieldFunc4(v[1]);
    field[2] = fieldFunc4(v[2]);
    field[3] = fieldFunc4(v[3]);
    field[4] = fieldFunc4(v[4]);
    field[5] = fieldFunc4(v[5]);
    field[6] = fieldFunc4(v[6]);
    field[7] = fieldFunc4(v[7]);

    // recalculate flag
    // (this is faster than storing it in global memory)
    uint cubeindex;
    cubeindex =  uint(field[0].w < isoValue);
    cubeindex += uint(field[1].w < isoValue)*2;
    cubeindex += uint(field[2].w < isoValue)*4;
    cubeindex += uint(field[3].w < isoValue)*8;
    cubeindex += uint(field[4].w < isoValue)*16;
    cubeindex += uint(field[5].w < isoValue)*32;
    cubeindex += uint(field[6].w < isoValue)*64;
    cubeindex += uint(field[7].w < isoValue)*128;

    // find the vertices where the surface intersects the cube

#if USE_SHARED
    // use partioned shared memory to avoid using local memory
    __shared__ float3 vertlist[12*NTHREADS];
    __shared__ float3 normlist[12*NTHREADS];

    vertexInterp2(isoValue, v[0], v[1], field[0], field[1], vertlist[threadIdx.x], normlist[threadIdx.x]);
    vertexInterp2(isoValue, v[1], v[2], field[1], field[2], vertlist[threadIdx.x+NTHREADS], normlist[threadIdx.x+NTHREADS]);
    vertexInterp2(isoValue, v[2], v[3], field[2], field[3], vertlist[threadIdx.x+(NTHREADS*2)], normlist[threadIdx.x+(NTHREADS*2)]);
    vertexInterp2(isoValue, v[3], v[0], field[3], field[0], vertlist[threadIdx.x+(NTHREADS*3)], normlist[threadIdx.x+(NTHREADS*3)]);
    vertexInterp2(isoValue, v[4], v[5], field[4], field[5], vertlist[threadIdx.x+(NTHREADS*4)], normlist[threadIdx.x+(NTHREADS*4)]);
    vertexInterp2(isoValue, v[5], v[6], field[5], field[6], vertlist[threadIdx.x+(NTHREADS*5)], normlist[threadIdx.x+(NTHREADS*5)]);
    vertexInterp2(isoValue, v[6], v[7], field[6], field[7], vertlist[threadIdx.x+(NTHREADS*6)], normlist[threadIdx.x+(NTHREADS*6)]);
    vertexInterp2(isoValue, v[7], v[4], field[7], field[4], vertlist[threadIdx.x+(NTHREADS*7)], normlist[threadIdx.x+(NTHREADS*7)]);
    vertexInterp2(isoValue, v[0], v[4], field[0], field[4], vertlist[threadIdx.x+(NTHREADS*8)], normlist[threadIdx.x+(NTHREADS*8)]);
    vertexInterp2(isoValue, v[1], v[5], field[1], field[5], vertlist[threadIdx.x+(NTHREADS*9)], normlist[threadIdx.x+(NTHREADS*9)]);
    vertexInterp2(isoValue, v[2], v[6], field[2], field[6], vertlist[threadIdx.x+(NTHREADS*10)], normlist[threadIdx.x+(NTHREADS*10)]);
    vertexInterp2(isoValue, v[3], v[7], field[3], field[7], vertlist[threadIdx.x+(NTHREADS*11)], normlist[threadIdx.x+(NTHREADS*11)]);
    __syncthreads();

#else
    float3 vertlist[12];
    float3 normlist[12];

    vertexInterp2(isoValue, v[0], v[1], field[0], field[1], vertlist[0], normlist[0]);
    vertexInterp2(isoValue, v[1], v[2], field[1], field[2], vertlist[1], normlist[1]);
    vertexInterp2(isoValue, v[2], v[3], field[2], field[3], vertlist[2], normlist[2]);
    vertexInterp2(isoValue, v[3], v[0], field[3], field[0], vertlist[3], normlist[3]);

    vertexInterp2(isoValue, v[4], v[5], field[4], field[5], vertlist[4], normlist[4]);
    vertexInterp2(isoValue, v[5], v[6], field[5], field[6], vertlist[5], normlist[5]);
    vertexInterp2(isoValue, v[6], v[7], field[6], field[7], vertlist[6], normlist[6]);
    vertexInterp2(isoValue, v[7], v[4], field[7], field[4], vertlist[7], normlist[7]);

    vertexInterp2(isoValue, v[0], v[4], field[0], field[4], vertlist[8], normlist[8]);
    vertexInterp2(isoValue, v[1], v[5], field[1], field[5], vertlist[9], normlist[9]);
    vertexInterp2(isoValue, v[2], v[6], field[2], field[6], vertlist[10], normlist[10]);
    vertexInterp2(isoValue, v[3], v[7], field[3], field[7], vertlist[11], normlist[11]);
#endif

    // output triangle vertices
    uint numVerts = tex1Dfetch(numVertsTex, cubeindex);

    for (int i=0; i<numVerts; i++)
    {
        uint edge = tex1Dfetch(triTex, cubeindex*16 + i);

        uint index = numVertsScanned[voxel] + i;

        if (index < maxVerts)
        {
#if USE_SHARED
            pos[index] = make_float4(vertlist[(edge*NTHREADS)+threadIdx.x], 1.0f);
            norm[index] = make_float4(normlist[(edge*NTHREADS)+threadIdx.x], 0.0f);
#else
            pos[index] = make_float4(vertlist[edge], 1.0f);
            norm[index] = make_float4(normlist[edge], 0.0f);
#endif
        }
    }
}

extern "C" void
launch_generateTriangles(dim3 grid, dim3 threads,
                         float4 *pos, float4 *norm, uint *compactedVoxelArray, uint *numVertsScanned,
                         uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask,
                         float3 voxelSize, float isoValue, uint activeVoxels, uint maxVerts)
{
    generateTriangles<<<grid, NTHREADS>>>(pos, norm,
                                          compactedVoxelArray,
                                          numVertsScanned,
                                          gridSize, gridSizeShift, gridSizeMask,
                                          voxelSize, isoValue, activeVoxels,
                                          maxVerts);
    getLastCudaError("generateTriangles failed");
}


// calculate triangle normal
__device__
float3 calcNormal(float3 *v0, float3 *v1, float3 *v2)
{
    float3 edge0 = *v1 - *v0;
    float3 edge1 = *v2 - *v0;
    // note - it's faster to perform normalization in vertex shader rather than here
    return cross(edge0, edge1);
}

// version that calculates flat surface normal for each triangle
__global__ void
generateTriangles2(float4 *pos, float4 *norm, uint *compactedVoxelArray, uint *numVertsScanned, uchar *volume,
                   uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask,
                   float3 voxelSize, float isoValue, uint activeVoxels, uint maxVerts)
{
    uint blockId = __mul24(blockIdx.y, gridDim.x) + blockIdx.x;
    uint i = __mul24(blockId, blockDim.x) + threadIdx.x;

    if (i > activeVoxels - 1)
    {
        i = activeVoxels - 1;
    }

#if SKIP_EMPTY_VOXELS
    uint voxel = compactedVoxelArray[i];
#else
    uint voxel = i;
#endif

    // compute position in 3d grid
    uint3 gridPos = calcGridPos(voxel, gridSizeShift, gridSizeMask);

    float3 p;
    p.x = -1.0f + (gridPos.x * voxelSize.x);
    p.y = -1.0f + (gridPos.y * voxelSize.y);
    p.z = -1.0f + (gridPos.z * voxelSize.z);

    // calculate cell vertex positions
    float3 v[8];
    v[0] = p;
    v[1] = p + make_float3(voxelSize.x, 0, 0);
    v[2] = p + make_float3(voxelSize.x, voxelSize.y, 0);
    v[3] = p + make_float3(0, voxelSize.y, 0);
    v[4] = p + make_float3(0, 0, voxelSize.z);
    v[5] = p + make_float3(voxelSize.x, 0, voxelSize.z);
    v[6] = p + make_float3(voxelSize.x, voxelSize.y, voxelSize.z);
    v[7] = p + make_float3(0, voxelSize.y, voxelSize.z);

#if SAMPLE_VOLUME
    float field[8];
    field[0] = sampleVolume(volume, gridPos, gridSize);
    field[1] = sampleVolume(volume, gridPos + make_uint3(1, 0, 0), gridSize);
    field[2] = sampleVolume(volume, gridPos + make_uint3(1, 1, 0), gridSize);
    field[3] = sampleVolume(volume, gridPos + make_uint3(0, 1, 0), gridSize);
    field[4] = sampleVolume(volume, gridPos + make_uint3(0, 0, 1), gridSize);
    field[5] = sampleVolume(volume, gridPos + make_uint3(1, 0, 1), gridSize);
    field[6] = sampleVolume(volume, gridPos + make_uint3(1, 1, 1), gridSize);
    field[7] = sampleVolume(volume, gridPos + make_uint3(0, 1, 1), gridSize);
#else
    // evaluate field values
    float field[8];
    field[0] = fieldFunc(v[0]);
    field[1] = fieldFunc(v[1]);
    field[2] = fieldFunc(v[2]);
    field[3] = fieldFunc(v[3]);
    field[4] = fieldFunc(v[4]);
    field[5] = fieldFunc(v[5]);
    field[6] = fieldFunc(v[6]);
    field[7] = fieldFunc(v[7]);
#endif

    // recalculate flag
    uint cubeindex;
    cubeindex =  uint(field[0] < isoValue);
    cubeindex += uint(field[1] < isoValue)*2;
    cubeindex += uint(field[2] < isoValue)*4;
    cubeindex += uint(field[3] < isoValue)*8;
    cubeindex += uint(field[4] < isoValue)*16;
    cubeindex += uint(field[5] < isoValue)*32;
    cubeindex += uint(field[6] < isoValue)*64;
    cubeindex += uint(field[7] < isoValue)*128;

    // find the vertices where the surface intersects the cube

#if USE_SHARED
    // use shared memory to avoid using local
    __shared__ float3 vertlist[12*NTHREADS];

    vertlist[threadIdx.x] = vertexInterp(isoValue, v[0], v[1], field[0], field[1]);
    vertlist[NTHREADS+threadIdx.x] = vertexInterp(isoValue, v[1], v[2], field[1], field[2]);
    vertlist[(NTHREADS*2)+threadIdx.x] = vertexInterp(isoValue, v[2], v[3], field[2], field[3]);
    vertlist[(NTHREADS*3)+threadIdx.x] = vertexInterp(isoValue, v[3], v[0], field[3], field[0]);
    vertlist[(NTHREADS*4)+threadIdx.x] = vertexInterp(isoValue, v[4], v[5], field[4], field[5]);
    vertlist[(NTHREADS*5)+threadIdx.x] = vertexInterp(isoValue, v[5], v[6], field[5], field[6]);
    vertlist[(NTHREADS*6)+threadIdx.x] = vertexInterp(isoValue, v[6], v[7], field[6], field[7]);
    vertlist[(NTHREADS*7)+threadIdx.x] = vertexInterp(isoValue, v[7], v[4], field[7], field[4]);
    vertlist[(NTHREADS*8)+threadIdx.x] = vertexInterp(isoValue, v[0], v[4], field[0], field[4]);
    vertlist[(NTHREADS*9)+threadIdx.x] = vertexInterp(isoValue, v[1], v[5], field[1], field[5]);
    vertlist[(NTHREADS*10)+threadIdx.x] = vertexInterp(isoValue, v[2], v[6], field[2], field[6]);
    vertlist[(NTHREADS*11)+threadIdx.x] = vertexInterp(isoValue, v[3], v[7], field[3], field[7]);
    __syncthreads();
#else

    float3 vertlist[12];

    vertlist[0] = vertexInterp(isoValue, v[0], v[1], field[0], field[1]);
    vertlist[1] = vertexInterp(isoValue, v[1], v[2], field[1], field[2]);
    vertlist[2] = vertexInterp(isoValue, v[2], v[3], field[2], field[3]);
    vertlist[3] = vertexInterp(isoValue, v[3], v[0], field[3], field[0]);

    vertlist[4] = vertexInterp(isoValue, v[4], v[5], field[4], field[5]);
    vertlist[5] = vertexInterp(isoValue, v[5], v[6], field[5], field[6]);
    vertlist[6] = vertexInterp(isoValue, v[6], v[7], field[6], field[7]);
    vertlist[7] = vertexInterp(isoValue, v[7], v[4], field[7], field[4]);

    vertlist[8] = vertexInterp(isoValue, v[0], v[4], field[0], field[4]);
    vertlist[9] = vertexInterp(isoValue, v[1], v[5], field[1], field[5]);
    vertlist[10] = vertexInterp(isoValue, v[2], v[6], field[2], field[6]);
    vertlist[11] = vertexInterp(isoValue, v[3], v[7], field[3], field[7]);
#endif

    // output triangle vertices
    uint numVerts = tex1Dfetch(numVertsTex, cubeindex);

    for (int i=0; i<numVerts; i+=3)
    {
        uint index = numVertsScanned[voxel] + i;

        float3 *v[3];
        uint edge;
        edge = tex1Dfetch(triTex, (cubeindex*16) + i);
#if USE_SHARED
        v[0] = &vertlist[(edge*NTHREADS)+threadIdx.x];
#else
        v[0] = &vertlist[edge];
#endif

        edge = tex1Dfetch(triTex, (cubeindex*16) + i + 1);
#if USE_SHARED
        v[1] = &vertlist[(edge*NTHREADS)+threadIdx.x];
#else
        v[1] = &vertlist[edge];
#endif

        edge = tex1Dfetch(triTex, (cubeindex*16) + i + 2);
#if USE_SHARED
        v[2] = &vertlist[(edge*NTHREADS)+threadIdx.x];
#else
        v[2] = &vertlist[edge];
#endif

        // calculate triangle surface normal
        float3 n = calcNormal(v[0], v[1], v[2]);

        if (index < (maxVerts - 3))
        {
            pos[index] = make_float4(*v[0], 1.0f);
            norm[index] = make_float4(n, 0.0f);

            pos[index+1] = make_float4(*v[1], 1.0f);
            norm[index+1] = make_float4(n, 0.0f);

            pos[index+2] = make_float4(*v[2], 1.0f);
            norm[index+2] = make_float4(n, 0.0f);
        }
    }
}

extern "C" void
launch_generateTriangles2(dim3 grid, dim3 threads,
                          float4 *pos, float4 *norm, uint *compactedVoxelArray, uint *numVertsScanned, uchar *volume,
                          uint3 gridSize, uint3 gridSizeShift, uint3 gridSizeMask,
                          float3 voxelSize, float isoValue, uint activeVoxels, uint maxVerts)
{
    generateTriangles2<<<grid, NTHREADS>>>(pos, norm,
                                           compactedVoxelArray,
                                           numVertsScanned, volume,
                                           gridSize, gridSizeShift, gridSizeMask,
                                           voxelSize, isoValue, activeVoxels,
                                           maxVerts);
    getLastCudaError("generateTriangles2 failed");
}

extern "C" void ThrustScanWrapper(unsigned int *output, unsigned int *input, unsigned int numElements)
{
    thrust::exclusive_scan(thrust::device_ptr<unsigned int>(input),
                           thrust::device_ptr<unsigned int>(input + numElements),
                           thrust::device_ptr<unsigned int>(output));
}

#endif
```