CUDA Documentation

Mehrdad Yousefi (CCIT Visualization Group)

August 17, 2017
Abstract

In this documentation, we will provide some intermediate and advanced information about CUDA/OpenGL visualization programming. It needs some advanced C/C++ and CUDA programming knowledges. The all pictures, codes and datasets which are used in this documentation are taken from work of Marek Fiser. First we will review how to read the simulation data structures into NVIDIA GPU device and then we will discuss some fluid dynamics visualization methods for this big parallel visualization scheme.
Introduction

NVIDIA GPU device can be used as a little parallel supercomputer which is built based on CUDA programming language. This parallel scheme can be used for several scientific programming purposes such as parallel programming, simulation and visualization. In this documentation we will discuss a tutorial for CUDA scientific visualization from work of Marek Fiser which is about loading a huge simulation output file in NVIDIA GPU device and process them to create some wonderful fluid dynamics visualization from those raw data. So first we will provide some programming idea for how to read the raw binary data into NVIDIA GPU device and then briefly we will discuss some mathematical background of this visualization.

Loading raw data structure into NVIDIA GPU device

In this documentation we want to load a simulation output from fluid dynamics modeling around a delta wing. So our data structure includes three velocity direction value for velocity vector and the position of each node which is corresponding to that velocity vector. In Fig. 1 you may find the geometry of simulated air flow around that delta wing. In order to load data in NVIDIA

![Figure 1: Delta wing geometry and its mesh representation which was an input for fluid dynamics simulation.](image)

GPU device first of all we should load it just one time in CPU by using C++ standard library for binary data reading function and then transfer it to the GPU for processing. In Listing 1 you could find a peace of C++ code for reading data from binary files.

```
Listing 1: Load binary data into CPU
1 std::ifstream inputStream(filePath, std::ios::binary);
2 // Reading of header information skipped.
3 / ... float3 size initialized with size of VF
4 size_t totalSize = size.x * size.y * size.z;
5 float4* data = new float4[totalSize];
6
7 for (size_t i = 0; i < totalSize; ++i) {
8    float4* f4Ptr = &data[i];
9    // Read x, y, z.
10    inputStream.read((char*)f4Ptr, sizeof(float3));
```

1
// Compute magnitude as w.
curr->w = std::sqrtf
(f4Ptr->x * f4Ptr->x +
f4Ptr->y * f4Ptr->y +
f4Ptr->z * f4Ptr->z);
}

Listing 1: First can calculate the total size of three-dimensional data structure
and allocate a four-dimensional vector (Note: we will use four-dimensional vec-
tors instead of three-dimensional ones because it is more efficient and can store
the three-dimensional vector plus its magnitude) according to that total size.
Then we will read each three direction of velocity vector into four-dimensional
vector and also compute the magnitude of this vector and transfer it to the
four-dimensional vector as the fourth value. The next step is the transferring
the loaded data structure into NVIDIA GPU device. Also in order to create
more powerful visualization we need to do some interpolation on velocity vec-
torial field. Fortunately, CUDA has some powerful tools to do interpolation
on four-dimensional vectors and we could get another benefit from using four-
dimensional vectors instead of three-dimensional ones. You may find loading
and interpolation of data structure CUDA peace of code in Listing 2.

Listing 2: Transfer loaded data from CPU to CUDA GPU and interpolate the
four-dimensional vectorial field

texture<float4, cudaTextureType3D,
cudaReadModeElementType> vectorFieldTex;
cudaArray* d_volumeArray = nullptr;

// Allocate 3D array.
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<float4>();
cudaMalloc3DArray(&d_volumeArray, &channelDesc, volumeSize);

// Copy data to 3D array using pitched ptr.
cudaMemcpy3DParms copyParams = {0};
copyParams.srcPtr = make_cudaPitchedPtr((void*)h_volume,
volumeSize.width * sizeof(float4),
volumeSize.width, volumeSize.height);
copyParams.dstArray = d_volumeArray;
copyParams.extent = volumeSize;
copyParams.kind = cudaMemcpyHostToDevice;

// Set texture parameters.
vectorFieldTex.normalized = false;
vectorFieldTex.filterMode = cudaFilterModeLinear;
vectorFieldTex.addressMode[0] = cudaAddressModeClamp;
vectorFieldTex.addressMode[1] = cudaAddressModeClamp;
vectorFieldTex.addressMode[2] = cudaAddressModeClamp;
// Bind 3D array to 3D texture.
cudaBindTextureToArray(vectorFieldTex,
   d_volumeArray, channelDesc);
}

In Listing 2 first we need to allocate a three-dimensional texture for transferring the data from CPU to GPU and then by using cudaMemcpyHostToDevice we will copy the data from CPU to GPU device and initializing the vectorial texture and do linear interpolation on that four-dimensional vectorial texture. By connecting CUDA and OpenGL, we could some wonderful advantages for visualization. For example it can minimizing the need for data transferring between CPU and GPU and just we do our computation in the GPU device and visualize it immediately from device. So in Listing 3 we showed that how we could connect OpenGL and CUDA buffer and transfer the address of allocated memory for OpenGL to CUDA buffer.

Listing 3: How to connect OpenGL and CUDA and transfer the memory buffer pointer for visualization GPU device

cudaError_t createCudaSharedVbo(GLuint* vbo,
   GLenum target, uint size,
   cudaGraphicsResource** cudaResource) {

   glGenBuffers(1, vbo);
   glBindBuffer(target, *vbo);
   glBufferData(target, size, 0, GL_DYNAMIC_DRAW);
   glBindBuffer(target, 0);
   return cudaGraphicsGLRegisterBuffer(cudaResource,
      vbo, cudaGraphicsMapFlagsNone);
}

In Listing 3 first we initialized an OpenGL buffer and then transfer it to CUDA graphical buffer for rendering the visualization without CPU/GPU data transfer.

Visualizing Streamlines with CUDA/OpenGL

Now by using numerical integration methods and speed up of CUDA and rendering by CUDA/OpenGL buffers, we could visualize the vectorial field for example in CFD simulations which are shown in Fig. 2, Fig. 3, Fig. 4, Fig. 5, Fig. 6, Fig. 7, Fig. 8, Fig. 9 and Fig. 10.
Figure 2: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.

Figure 3: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.
Figure 4: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.

Figure 5: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.
Figure 6: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.

Figure 7: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.
Figure 8: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.

Figure 9: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.
Figure 10: Result of CUDA/OpenGL rendering of CFD simulations from Marek Fiser works.