# SiftGPU Manual

Changchang Wu

University of North Carolina at Chapel Hill

## Introduction

SiftGPU is a GPU implementation of David Lowe's Scale Invariant Feature Transform. The following steps can use GPU to process pixels/features in a parallel way:

1. Convert color into intensity, and up-sample or down-sample input images
2. Build Gaussian image pyramids (Intensity, Gradient, DOG)
3. Keypoint detection (sub-pixel and sub-scale localization)
4. Generate compact feature lists with GPU histogram reduction
5. Compute feature orientations
6. Compute feature descriptors

By taking advantages of the large number of graphic processing units in modern graphic cards, this GPU implementation of SIFT can achieve a large speedup over CPU.

Not all computation is faster on GPU, so the library is also trying to find the best option for each step. The latest version does intensity conversion, down-sampling, multi-orientation feature list rebuilding, and descriptor normalization on CPU. The latest keypoint list generation is also a GPU/CPU mixed implementation.

Running SiftGPU requires a high-end graphic card that 1) has a large graphic memory to keep the allocated intermediate textures for efficient processing of new images. 2) Supports dynamic branching. The loops in orientation computation and descriptor generation are decided by the scale of the features, and they cannot be unrolled.

SiftGPU now runs on GLSL by default. You can optionally use CG (requires fp40) or CUDA (experimental) for nVidia graphic cards.

## Interface: class SiftGPU

Class SiftGPU is provided as the interface of this library. The following examples will show how to use this class to run SIFT in a different ways.

**Initialization**, Normal initialization for an application

```
//create a SiftGPU instance
SiftGPU sift;

//processing parameters first
char * argv[] ={ "-fo", "-1", “-v”, “1”};
//-fo -1,  starting from -1 octave
//-v 1,    only print out # feature and overall time
sift.ParseParam(4, argv);

 //create an OpenGL context for computation
int support = sift.CreateContextGL();
//call VerfifyContexGL instead if using your own GL context
//int support = sift.VerifyContextGL();

if(support != SiftGPU::SIFTGPU_FULL_SUPPORTED) return;
```

**Example #1**, run sift on a set of images and get results:

```
//process an image, and save ASCII format SIFT files
if(sift.RunSIFT("1.jpg")) sift.SaveSIFT("1.sift");

//you can get the feature vector and store it yourself
sift.RunSIFT("2.jpg");
int num = sift.GetFeatureNum();//get feature count

//allocate memory for readback
vector<float> descriptors(128*num);
vector<SiftGPU::SiftKeypoint> keys(num);

//read back keypoints and normalized descritpros
//specify NULL if you don’t need keypionts or descriptors
sift.GetFeatureVector(&keys[0], &descriptors[0]);
```

**Example #2**, run SiftGPU with your own image data

```
// This is very convenient for camera application
int width = …, height =…;
unsigned char *data = … // your (intensity) image data
sift.RunSIFT (width, height, data, GL_RGBA, GL_UNSIGNED_BYTE);
//Better use GL_LUMINANCE  data to save transfer time
```

**Example #3**, specify  a set of image inputs using SetImageList

```
char * files[4] = { “1.jpg”, “2.jpg”, “3.jpg”, “4.jpg”};
sift.SetImageList(4, files);
//Now you can process an image with its index
sift.RunSIFT(1);
sift.RunSIFT(0);
```

**Example #4**, manual storage allocation

```
//Option1, use “-p”, “1024x1024” to initialize the texture
//storage for size 1024x1024, so that processing smaller
//images does not require texture re-allocation
//char * argv[] ={ "-m", "-s", “-p”, “1024x1024”};
//sift.ParseParam(4, argv);

//Option2, manually allocate the storage
sift.AllocatePyramid(1024, 1024);

// processing images with different sizes.
sift.RunSIFT(“1024x768.jpg”);
sift.RunSIFT(“768x1024.jpg”);
sift.RunSIFT(“800x600.jpg”);
```

**Example #5**, runtime  library loading

```
//new exported function CreateNewSiftGPU
SiftGPU* (*pCreateNewSiftGPU)(int) = NULL;
//Load siftgpu dll… use dlopen in linux/mac
HMODULE  hsiftgpu = LoadLibrary("siftgpu.dll");
 //get function address
pCreateNewSiftGPU = (SiftGPU* (*) (int))
          GetProcAddress(hsiftgpu, "CreateNewSiftGPU");
//create a new siftgpu instance
//exported functions are all virtual
SiftGPU * psift = pCreateNewSiftGPU(1);
```

**Example #6**, Compute descriptor for user-specified keypoints(NEW!)

```
vector<SiftGPU::SiftKeypoint>  keys;
//load your sift keypoints using your own function
LoadMyKeyPoints(…);

//Specify the keypoints for next image to siftgpu
sift.SetKeypointList(keys.size(), &keys[0]);
sift.RunSIFT(new_image_path);// RunSIFT on your image data
//****If it is to re-run SIFT with different keypoints***
//Use sift.RunSIFT(keys.size(), &keys[0]) to skip filtering

//Get the feature descriptor
float descriptor* = new [128 * keys.size()];
//We only need to read back the descriptors
sift.GetFeatureVector(NULL, descriptor);
```

**Example #7**, Compute (guided) putative sift matches. (NEW)

```
//specify the naximum number of features to match
SiftMatchGPU matcher(4096);
//You can call SetMaxSift anytime change this limit
//You can call SetLanguage to select shader language
//between CG/GLSL/CUDA before initialization

//Verify current OpenGL Context and do initialization
if(matcher.VerifyContextGL() == 0) return;

//Set two sets of descriptor data to the matcher
matcher.SetDescriptors(0, num1, des1);
matcher.SetDescriptors(1, num2, des2);

//Match and read back result to input buffer
int match_buf[4096][2];
int nmatch = matcher.GetSiftMatch(4096, match_buf);

// You can also use homography and/or fundamental matrix to
// guide the putative matching
// Check function SiftMatchGPU::GetGuidedMatch

// For more details of the above functions, check
// SimpleSIFT.cpp  and SiftGPU.h in the code package.
```

## OpenGL Context

SiftGPU uses OpenGL, and there has to be an OpenGL context to run the program. There are several ways to initialize the OpenGL context:

1. Use function SiftGPU::CreateContextGL. It simply uses GLUT to create an invisible window and use that GL context to run the shaders. (The example SimpleSIFT is doing this way).
2. Use GLUT yourself (see example project TestWinGlut).
3. Use raw OpenGL functions (see example project TestWin). You have to make sure you have an active context before calling SiftGPU functions (for example: call **WglMakeCurrent** to set the context in windows).

## Multiple Implementations (CG/GLSL/CUDA)

The code package includes 5 different implementations of SiftGPU. CG Unpacked/Packed, GLSL Unpacked/Packed and CUDA. They can be selected by using combination of "-glsl","-cg", "-unpack", "-pack" and "-cuda". "-glsl -pack" is now default.

The processing speed decreases when the image size increases. On NVIDIA 8800 GTX, the CG/GLSL packed versions are faster than CUDA for large images, while CUDA is faster for small images. This order could be different on different GPUs, and you can just try them on your computer to select the best one for different image sizes.

The packed versions take the smallest amount of GPU memory. The CUDA version takes more memory than others because part of the intermediate results has two copies (Both linear memory and 2D texture).

SiftMatchGPU also has implementations for CG, GLSL and CUDA, and they can be selected by calling function SiftMatchGPU::SetLanguage. CG/GLSL matching is slightly slower than CUDA for exhaustive putative matching. CG/GLSL matching is faster for guided putative matching.

## Memory Management

SiftGPU needs to allocate OpenGL textures (or CUDA linear memory/texture) for storing intermediate results. This allocation is a time-consuming step, and it would be efficient if memory re-allocation is infrequent and the storage can be re-used to process lots of images. The best performance can be obtained when you pre-resize all images to a same size, and process them with one SiftGPU instance.

When starting up, you can pre-allocate the memories to fit some specified size or SiftGPU will automatically fit the first image. You can also manually re-allocate the active pyramid at anytime by calling *SiftGPU::AllocatePyramid(int width, int height)*.

While processing an image that has a different size, the memory behavior is defined by the memory management mode as follows:

1) **Auto** (default). The storage will automatically resize to fit the largest width and the largest height so far. But you can pre-allocate it to the largest size you know so that there won't be any re-allocation after. SiftGPU will reuse existing storage that is allocated for a large image to process small images (See example 4)
2) **Tight**. The storage will automatically resize to any new image size. It saves memory for smaller images, but there will be a re-allocation each time when the image size changes. If you start to process a set of images of a same size, you can change to "tight" mode for better performance. (not available in CUDA version )

The memory management mode can be changed at runtime by calling function *SiftGPU::SetTightPyrmid(int tight = 1).*

When you run TestWinGlut with the first input image, it will print out the total number of megabytes of textures it takes (not including the copy of the original 4-channel image). Please do compare that number with your total number of GPU memory.

# Parameter System (used by SiftGPU::ParseParam)

♠ the parameter can be changed after initialization in all implementations
♦ the parameter can be changed after initialization in CUDA implementation

| | |
|---|---|
| -h -help | Display usage help |
| -i <strings> | Filenames of the input images (for example: -i 1.jpg 2.jpg 3.jpg) |
| -il <string> | Filename of an image list file |
| -o <string> | Where to save SIFT features |
| -f <float> ♦ | Factor for filter width [2*factor*sigma+1 ] (default : 4.0) |
| -w <float> ♦ | Factor for orientation sample window [2*factor*sigma] (default : 2.0) |
| -dw <float> ♠ | Factor for descriptor grid size [4*factor*sigma] (default : 3.0) |
| -fo <int> ♠ | First Octave to start detection(default: 0) |
| -no <int> | Maximum number of octaves (default: not limit) |
| -d <int> | DOG levels in an octave (default: 3) |
| -t <float> ♦ | DOG threshold (default: 0.02/3) |
| -e <float> ♦ | Edge Threshold (default : 10.0) |
| -m –mo <int=2> | Number of possible Feature Orientations (default : 2) |
| -m2p | Use packed orientations (one float to store 2 orientations)<br>-m2p and –m1 may be slower than the default (-m 2) |
| -s <int=1> | Enable sub-pixel, sub-scale Localization, If the number is 0, sub-pixel is disabled. If the number is larger than 1, multiple refinement will be done, but only available in cg-unpacked |
| -lc <int =-1> | CPU/GPU mixed Feature List Generation (default : 6)<br>Use GPU first, and use CPU when reduction size <= 2 ^num<br>When <num> equals -1, no GPU reduction will be used |
| -noprep | Upload raw data to GPU if specified (Converting RGB to LUM and down-sampling is running on CPU by default) |
| -sd | Skip descriptor computation if specified |
| -unn ♠ | Write un-normalized descriptor if specified |
| -b ♠ | Write binary format descriptors |
| -fs <int> | Block size for feature storage <default : 4> (4 or 8 might be better than 1 in GPU parallelism) |

| | |
|---|---|
| -cuda | Use CUDA based implementation |
| -cg | Use CG instead of GLSL |
| -glsl | Use GLSL instead of CG (This is default) |
| -tight | Automatically resize storage to fit tightly to new image size<br>(in the default mode, the storage is only enlarged) |
| -p WxH | Set the dimension for initializing pyramids. For example: -p 1024x768 will let all pyramid initialized to 1024x768 |
| -lm <num><br>-lmp <percent> | Maximum feature count for a level (default: 4096), and that as a ratio of the pixel count (default: 0.05). The two parameters are only used in initial allocation, and textures will be automatically re-allocated when the storage is insufficient. |
| -v <level> ♠ | Same effect as calling SetVerbose(level)<br>0, no output at all, except errors<br>1, print out over all timing and features numbers<br>2, print out timing for each steps<br>3, print out timing for each octaves<br>4, print out timing for each levels |
| -ofix ♠ | Fix the orientation of all features to 0 |
| -ofix-not ♠ | Disable -ofix |
| -loweo ♠ | Let (0, 0) be center of top-left pixel instead of corner with this parameter. The corner is (0, 0) by default, but Lowe's SIFT and sift++ are using the pixel center. |
| -maxd ♠ | Maximum working dimension. When some level images are larger than this, the input image will be automatically down-sampled. (default: 2560(unpacked) / 3200(packed)) |
| -exit | Exit the TestWinGlut application after processing the image. (otherwise the viewer will show up) |
| -di ♦ | For OpenGL-based, use dynamic array indexing in histogram computation in the orientation computation<br>For CUDA, use dynamic array indexing in descriptor generation. |
| -pack(default)<br>-unpack | Use packed/unpacked implementation. The packed version should be faster than the unpacked version. |
| -sign ♦ | When specified, output scale of local DOG minimum keypoints will be multiplied by -1. |

You can also change the default parameters in GlobalUtil.cpp and compile it yourself.

## SiftGPU Viewers

There are 2 GUI viewers for SiftGPU in the folder TestWin\bin.

TestWinGlut.exe is a Glut-based viewer that has console output.

TestWin.exe is a Win32-based viewer that does not have console output

There are 7 view modes in the viewers:

0, original image and feature (drawn as blue points or rectangles):

1, Gaussian pyramid

2, octaves (View different octaves one by one)

3, levels (View different levels one by one)

4, the pyramid of difference of Gaussian

5, the pyramid of image gradient

6, detected keypoints in levels. Red points are the local maxima, and green points are local minima. You can **zoom** to see the details in levels

# Viewer keys

You can loop through these view modes by pressing the following keys:

| | |
|---|---|
| **Enter**, | next view |
| **Backspace**, | previous view |
| **Space . (>)** | next sub-view/level/octave (in view mode 0, 2, 3) |
| **,** (<) | previous sub-view/ level/octave (in view mode 0, 2, 3) |
| **x, Escape** | exit |

Some other controls are as follows:

| | |
|---|---|
| Mouse | click, hold and move to pan the view |
| **r** | Go to the next image if there is, and re-compute SIFT |
| **o** | reset coordinate |
| **+, =** | zoom in |
| **-,** | zoom out |
| **l** | start/stop **l**oopy processing of a set of images |
| **c** | Randomize the **c**olors for sift feature box display in view mode 0-2 |
| **q** | Change verbose level 2-1-0-2-1-0… |

# Demos

1. There are three demo batch files in the ‘demos’ folder.

   **Demo1.bat** is a basic example of SiftGPU. Try step through all the views to see the intermediate results of SIFT using the controls explained in the last section.

   **Demo2.bat** shows the processing of a file that contains a list of image filenames. The images in this demo are all of size 640x480. After the viewer shows up, press ‘l’ to start/stop process the input images one by one repeatedly. Other keys also work to change the view modes during the loop.

   **Demo3.bat** shows the processing of a list of images of **varying sizes**.

   **Evaluation-box.bat** computes the sift features for comparing with Lowe’s result.

2. **SimpleSIFT** project in the workspace/solution shows how to use SiftGPU without GUI. It also shows how to read back SIFT results from SiftGPU. There is also an optional macro which enables runtime loading of SiftGPU library.

3. **Speed** project shows how to evaluate the speed of SiftGPU