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 to 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 and 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 this library also tries 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.

NEW: V360 supports Multi-process Mode to use multiple GPUS (or GPUs on different computers) without **changing your programming interface**.

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);
//Using GL_LUMINANCE data saves 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, control 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

```
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.

```
//specify the naximum number of features to match
SiftMatchGPU matcher(4096);
//You can call SetMaxSift anytime to 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.
```

Example #8, Choosing the GPU for computation (under Multi-GPU system)

```
//Suppose (1024,0) is in the screen of the second GPU
//For X-server, use “-display”, “display_name”
//For CUDA, use “-cuda”, “device_index”
char * argv[] ={"-fo", "-1", “-winpos”, “1024x0”};
siftgpu->ParseParam.(4, argv);
if(!siftgpu->VerifyContextGL) return;

//GPU selection can’t be changed after VerifyContextGL
```

Example #9, Using a local multi-process mode **NEW**

```
//1st parameter, 7777 for the socket port used by server
//2nd parameter, NULL for local process mode
SiftGPU* siftgpu = new ServerSiftGPU(7777, NULL);
//You can create multiple ServerSiftGPU instances, and
//let them use different GPUs.

//Everything else is the same.
siftgpu->ParseParam(…);  //choose GPU if more than one
if(!siftgpu->VerifyContextGL()) return;

//Call RunSIFT functions. Most functions are supported
siftgpu->RunSIFT("1.jpg");

//when you call delete, the server will be shut down.
delete siftgpu;
```

Example #10, Using a remote SiftGPU on different computer NEW

```
//Suppose, you have a computer at mygpu.com
//you first start a siftgpu server by run
//      bin/server_siftgpu –server 7777 [siftgpu param]
//From a different computer you can do as follows

SiftGPU* siftgpu = new ServerSiftGPU(7777, “mygpu.com”);
siftgpu->ParseParam(…)

//if GPU selection is already done on the server.
//New GPU selection won’t work
if(!siftgpu->VerifyContextGL()) return;

delete siftgpu;
//after delete, server keeps running and accept new clients
```

OpenGL Context

SiftGPU uses OpenGL (*Not for the new multi-process mode and remote mode*), 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 uses Win32/XLib (or GLUT depending on your compilation setting) 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.

SiftGPU for Multiple-GPU (NEW)

Device Selection: You can select a particular GPU for SiftGPU computation. Different method need to be used in different systems and different implementations.

When using CUDA-based SiftGPU, you need to set parameter "-cuda device_index" when you want to use a particular device. For SiftMatchGPU, you need call SiftMatchGPU:: SetLanguage(SIFTMATCH_CUDA + device_index)

When using OpenGL-based implementation under Win32, you need to use a point coordinate (in the monitor that corresponds to the GPU you want) in virtual screen to select the GPU. The parameter format is "-winpos XxY" (for example, "-winpos 1024x0".)

When using OpenGL-based implementation under X-Window, you need use parameter "-display hostname:number.screen_number" to select a display to let SiftGPU use the corresponding GPU for computation.

Multiple SiftGPU instances (NEW)

Note that CUDA version can be multi-threaded if each thread is setup to use different device. See MultiThreadSIFT for an demo of this.

It is hard for the OpenGL-based one to use the multiple GPUs in the same process. However, you can run multiple GPU programs in different process to utilize different GPUs. The new version of SiftGPU is able to simply work as a client and controls multiple worker processes. It is able to automatically create worker processes on local computer and connect to some existing server on local/remote computer.

SiftGPU includes an implementation of SiftGPU server [server_siftgpu in server.cpp] and SiftGPU client [class ServerSiftGPU], with which you can easily run multiple SiftGPU instances on different GPUs on your local computer, or use a remote computer to run all the computation. What's most important is that it

doesn't change any of the programming interfaces, and all the wrapping is done internally.

You can look at server.cpp for examples. It is not only the implementation of the several but also includes some client-server example. The two command line options "-test" and "-test2" gives you the two examples.

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 storage by default 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. SiftGPU reuses existing storage to process any smaller images that can fit in (See example 4).

Optionally, you can select a tight mode by calling function *SiftGPU::SetTightPyrmid(int tight = 1).* The storage will then resize to any new image size. It does save memory for smaller images, but there will be a re-allocation each time when the image size changes.

NOTE: 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

-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) -m2p and –m1 may be slower than the default (-m 2)
-s <int=1>	Enable sub-pixel Localization. Use 0 to disable sub-pixel.
-lc <int =-1>	CPU/GPU mixed Feature List Generation (default : 6) Use GPU first, and use CPU when reduction size <= 2 ^num 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 <index=0>	Use CUDA based implementation, and select device
-cg	Use CG instead of GLSL, (GLSL is default)

-tight	Automatically resize storage to fit tightly to new image size (in the default mode, the storage dimension is only increased)
-p WxH	Set the dimension for initializing pyramids. For example: -p 1024x768 will let all pyramid initialized to 1024x768
-v <level> ♠	Same effect as calling SetVerbose(level) 0, no output at all, except errors 1, print out over all timing and features numbers 2, print out timing for each steps 3/4, print out timing for each octaves/ 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 For CUDA, use dynamic array indexing in descriptor generation.
-pack(default) -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.
-winpos XxY	Use by win32 to select GPU according to screen coordinate
-display name	Used through xLib to select GPU according to display
-tc, -tc1 <int> ♠ -tc2 <int> ♠ -tc3 <int> ♠	Set a soft limit to number of detected features, provide -1 to disable. -tc, -tc1, keep the highest levels. -tc2, keep the highest level, (should be faster than -tc) -tc3, keep the lowest levels
-nogl	Use –nogl for CUDA to skip all OpenGL calls. Previously OpenGL is still used for data transfer. Will use CPU if –nogl is used.

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

SiftGPU Viewers

There are 2 GUI viewers for SiftGPU

TestWinGlut is a GLUT-based viewer

TestWin.exe directly uses Win32 API to control OpenGL Contexts

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

4. **ServerSiftGPU** shows how to use SiftGPU as a computation server. The file /src/ServerSiftGPU/server.cpp gives the implementation of the server. With argument “-test” and “-test2” it can run as a client, which demos the usage of ServerSiftGPU.

5. **MultiThreadSIFT** shows how to multi-thread SiftGPU and use multiple GPU devices.