**GPGPU-Sim Learn**

**1.**GPGPU-Sim源码执行流程

GPGPU-Sim is initialized by the function GPGPUSim\_Init() which is called when the CUDA or OpenCL application performs its first CUDA/OpenCL API call. Our implementation of the CUDA/OpenCL API function implementations either directly call GPGPUSim\_Init() or they call GPGPUSim\_Context() which in turn calls GPGPUSim\_Init(). **An example API call that calls GPGPUSim\_Context() is cudaMalloc().** Note that by utilizing static variables GPGPUSim\_Init() is not called every time a cudaMalloc() is called.

./libcuda/cuda\_runtime\_api.cc cudaMalloc(void \*\*devPtr, size\_t size)

./libcuda/cuda\_runtime\_api.cc GPGPUSim\_Context()

./libcuda/cuda\_runtime\_api.cc GPGPUSim\_Init()

./src/gpgpusim\_entrypoint.cc gpgpu\_ptx\_sim\_init\_perf()

…

**./libopencl/opencl\_runtime\_api.cc class \_cl\_device\_id \*GPGPUSim\_Init()**

**./src/gpgpusim\_entrypoint.cc void \*gpgpu\_sim\_thread\_sequential(void\*)**

./libcuda/cuda\_runtime\_api.cc load\_constants( …)

./src/cuda-sim/ptx.y function\_defn: function\_decl

./src/cuda-sim/ptx\_parser.cc void end\_function()

./src/cuda-sim/ptx\_ir.cc void gpgpu\_ptx\_assemble( std::string kname, void \*kinfo )

./src/cuda-sim/cuda-sim.cc void function\_info::ptx\_assemble()

./src/cuda-sim/cuda-sim.cc void ptx\_instruction::pre\_decode()

./src/cuda-sim/cuda-sim.cc address\_type get\_converge\_point( address\_type pc )

./src/cuda-sim/cuda-sim.cc find\_reconvergence\_points( function\_info \*finfo )

**2. Ubuntu下GitHub的使用**

<http://www.pythoner.com/263.html#1482744-tsina-1-31141-a60185a16c01612f8d678e4b2d855b15>

**5.clone来自GitHub的项目**  
可以用如下方式将GitHub远程版本库中的代码clone到本地：  
git clone git@github.com:alioth310/test.git  
git clone git://github.com/alioth310/test.git

**6.其他常用的Git命令**

git init # 初始化本地Git版本库  
git add # 暂存文件，如果使用.表示当前目录及其子目录  
git commit -m “first commit” # 提交，-m选项后跟内容为提交所用的注释  
git remote -v # 查看当前项目远程连接的是哪个版本库地址  
git push origin master # 将本地项目提交到远程版本库

git fetch origin # 取得远程更新（到origin/master），但还没有合并  
git merge origin/master # 把更新的内容（origin/master）合并到本地分支（master）  
git pull origin master # 相当于fetch和merge的合并，但分步操作更保险

<http://www.cnblogs.com/yourihua/archive/2012/07/07/2580147.html>

具体指令：

　　a. 初始化git配置：

　　git config --global user.name "Your Name"

　　git config --global user.email yourihua@sina.com

　　b. 在你要提交的工程的根目录，执行

　　git init

　　git add .

　　git status #查看当前的git状态

　　git commit -m "上传第一个版本"

　　git log #查看git日志

　　c. 上传工程到github

　　git remote rm origin

　　git remote add origin git@github.com:<username>/first\_app.git

　　git push -u origin master

**3.** **github常见操作和常见错误！错误提示：fatal: remote origin already exists.**

[**http://blog.163.com/023\_dns/blog/static/1187273662013111301046930/**](http://blog.163.com/023_dns/blog/static/1187273662013111301046930/)

使用git在本地创建一个项目的过程

    $ makdir ~/hello-world    //创建一个项目hello-world  
    $ cd ~/hello-world       //打开这个项目  
    $ git init             //初始化   
    $ touch README  
    $ git add README        //更新README文件  
    $ git commit -m 'first commit'     //提交更新，并注释信息“first commit”  
    $ git remote add origin [git@github.com:defnngj/hello-world.git](mailto:git@github.com:defnngj/hello-world.git)     //连接远程github项目    
    $ git push -u origin master     //将本地项目更新到github项目上去

**4.** **linux sem信号量使用**

**semaphore 是一种通用的同步机制** --- 既可以在进程中使用也可以在线程中使用

**Linux下关于信号量结构体表示为:sem\_t**

**操作结构体的函数:**

**初始化函数: sem\_init(sem\_t \* \_\_sem,int \_\_pshared,unsigned int \_\_value);**

**触发信号量值:sem\_post(sem\_t \* \_\_sem);**

**等待信号量触发:**

**通常有:**

**一直等待:sem\_wait(sem\_t \* \_\_sem);**

**测试\_\_sem是否触发:sem\_trywait(sem\_t \* \_\_sem);**

**等待超时:sem\_timedwait(sem\_t \* \_\_restrict \_\_sem, \_\_ const struct timespec \* \_\_restrict \_\_abstime);**

**释放销毁信号量:**

**sem \_destroy(sem\_t \* \_\_sem);**

**/\* Initialize semaphore object SEM to VALUE. If PSHARED then share it with other processes. \*/**

**extern int sem\_init (sem\_t \*\_\_sem, int \_\_pshared, unsigned int \_\_value) \_\_THROW;**

**/\* Free resources associated with semaphore object SEM. \*/**

**extern int sem\_destroy (sem\_t \*\_\_sem) \_\_THROW;**

**/\* Open a named semaphore NAME with open flags OFLAG. \*/**

**extern sem\_t \*sem\_open (const char \*\_\_name, int \_\_oflag, ...) \_\_THROW;**

**/\* Close descriptor for named semaphore SEM. \*/**

**extern int sem\_close (sem\_t \*\_\_sem) \_\_THROW;**

**/\* Remove named semaphore NAME. \*/**

**extern int sem\_unlink (const char \*\_\_name) \_\_THROW;**

**/\* Wait for SEM being posted. This function is a cancellation point and therefore not marked with \_\_THROW. \*/**

**extern int sem\_wait (sem\_t \*\_\_sem);**

**#ifdef \_\_USE\_XOPEN2K**

**/\* Similar to `sem\_wait' but wait only until ABSTIME.This function is a cancellation point and therefore not marked with \_\_THROW. \*/**

**extern int sem\_timedwait (sem\_t \*\_\_restrict \_\_sem,const struct timespec \*\_\_restrict \_\_abstime);**

**#endif**

**/\* Test whether SEM is posted. \*/**

**extern int sem\_trywait (sem\_t \*\_\_sem) \_\_THROWNL;**

**/\* Post SEM. \*/**

**extern int sem\_post (sem\_t \*\_\_sem) \_\_THROWNL;**

**/\* Get current value of SEM and store it in \*SVAL. \*/**

**extern int sem\_getvalue (sem\_t \*\_\_restrict \_\_sem, int \*\_\_restrict \_\_sval) \_\_THROW;**