实验 CUDA编程

# 一、实验任务

熟悉GPU的计算原理，学会CUDA编程。

# 二、CUDA基本编程流程

为利用GPGPU的强大并行数据处理能力，需要把具体任务转化为GPU上执行的一个或若干个CUDA Kernel，要考虑Kernel之间，Kernel和其他API（比如cudaMemcpy），或者是和其他CPU任务之间怎么配合。为此，有必要先深入了解CUDA程序的各级组成单元。只有了解每层单元具体是怎么工作的，清楚它们相互同步、通信的机制，才能更好地让它们协作配合。请自行查找资料进行GPGPU CUDA编程学习。

基本的CUDA程序操作步骤是：

1）分配内存空间和显存空间

2）初始化内存空间

3）将要计算的数据从内存上复制到显存上

4）执行 kernel 计算

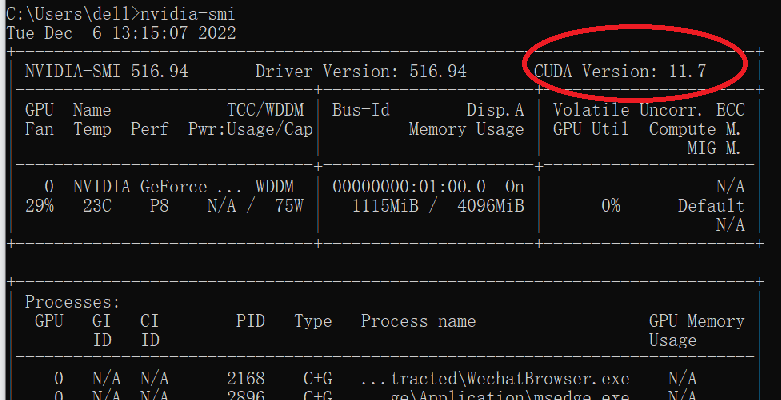
5）将计算后显存上的数据复制到内存上

6）处理复制到内存上的数据

# 三、Visual Studio 2022中CUDA程序开发环境配置

## 正确安装NVidia显卡驱动。

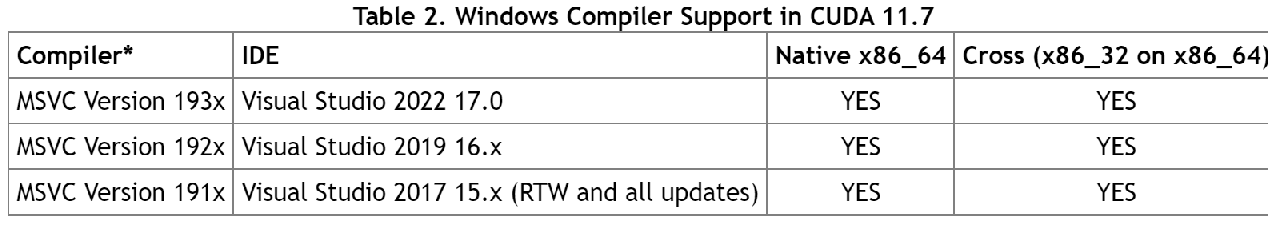
在命令行里输入nvidia-smi.exe，效果如下图所示。可以看到显示CUDA Version为11.7，说明该显卡最高支持到11.7。现在就选择11.7.0的版本，也可以选择更低的版本。



## 安装VS2022、CUDA、cuDNN（用于深度神经网络，本实验可以不装）

确定CUDA版本支持的VS版本。查询官方安装文档，可知支持的VS版本如下表：：

https://docs.nvidia.com/cuda/archive/11.7.0/cuda-installation-guide-microsoft-windows/index.html



这里选择Visual Studio 2022。

确定CUDA版本对应的cuDNN版本：在cudnn下载页面选择“Download cuDNN v8.6.0 (October 3rd, 2022) for CUDA 11.x”。

<https://developer.nvidia.com/rdp/cudnn-archive>

安装的顺序是先安装Visual Studio、CUDA，然后是cuDNN。

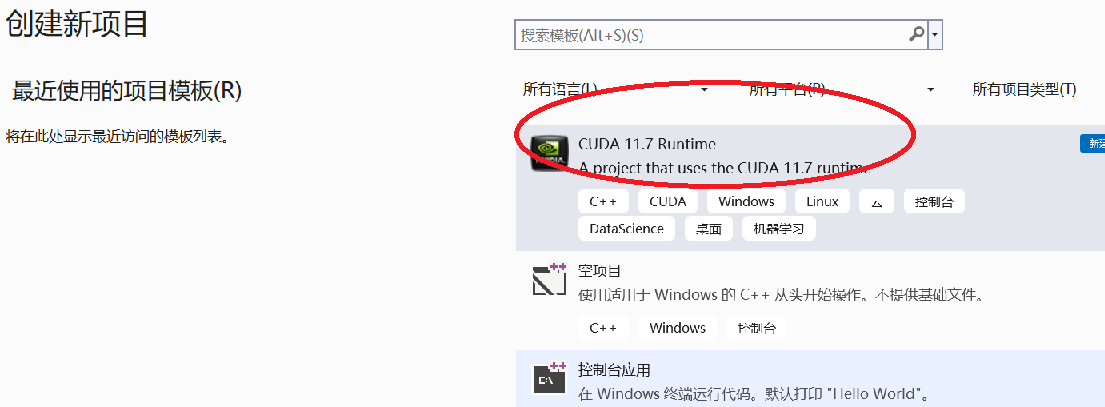
CUDA安装时会提示与Visual Studio整合的情况。

通过运行如下命令检查CUDA是否安装成功：打开cmd，输入 nvcc -V，会显示版本。

CUDA里面有个demo\_suite文件夹，有些工具可以运行一下，例如：deviceQuery、bandwidthTest。

至于cuDNN，下载下来是个压缩包，直接解压缩，把三个文件夹（bin、include、lib）文件分别拷贝到CUDA安装目录对应的文件夹中即可。

现在VS2022中新建项目，在模板中出现了CUDA模板，使用它新建项目。



模板会自动产生一个可编译的完整代码，编译这个模板并尝试运行，如果成功，说明VS2022中CUDA编程环境已经搭建完毕。

# 四、实验内容

## 运行矩阵相乘CUDA程序，分析哪些因素影响程序执行时间

程序可以自己编，也可以使用下面这个在环境Win10 VS2022+CUDA Toolkit 11.7上的样例源代码。

//本程序用GPGPU计算矩阵乘：A[M,K] \* B[K,N] = C[M,N]

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define M 1024

#define N 512

#define K 1024

//kernel函数，这个函数是运行在CUDA核心上的，负责计算 结果矩阵的一个元素

\_\_global\_\_ void multiplyKernel(const int \*dev\_A, const int \* dev\_B, int \* dev\_C) {

int Row = blockIdx.y \* blockDim.y + threadIdx.y; //行

int Col = blockIdx.x \* blockDim.x + threadIdx.x; //列

int sum = 0;

for (int k = 0; k < K; k++) sum += dev\_A [Row \* K + k] \* dev\_B [k \* N + Col];

dev\_C [Row \* N + Col] = sum; //C[M,N]

}

//检查C[x,y]计算是否正确

void check\_matrix(int x, int y, int \*A, int \*B, int \*C) {

int sum = 0;

for (int i = 0; i < K; i++) sum += A[x\*K + i] \* B[i\*N + y];

if (sum == C[x \*N + y] )

printf("C[%d,%d] is right\n",x,y);

else

printf("C[%d,%d]=%d, sum=%d\n",x,y, C[x \* N + y], sum);

}

// Helper function for using CUDA to multiply matrices in parallel.

cudaError\_t multiplyWithCuda(int \*C, const int \*A, const int \*B) {

int \*dev\_A = 0;

int \*dev\_B = 0;

int \*dev\_C = 0;

cudaError\_t cudaStatus;

// Choose which GPU to run on, change this on a multi-GPU system.

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Any CUDA-capable GPU installed?");

goto Error;

}

// Allocate GPU buffers(global memory) for three matrices A B C.

cudaStatus = cudaMalloc((void\*\*)&dev\_A, M\*K \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_B, K\*N \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc ( (void\*\*)& dev\_C, M\*N \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

// Copy input matrices from host memory to GPU buffers.

cudaStatus = cudaMemcpy(dev\_A, A, M\*K \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_B, B, K\*N \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

dim3 dimGrid(16, 32); //grid的维度

dim3 dimBlock(32, 32); //block维度

// Launch a kernel on the GPU with one thread for each element of C matrix.

multiplyKernel<<<dimGrid, dimBlock >>>(dev\_A, dev\_B, dev\_C );

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr,"multiplyKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudaDeviceSynchronize waits for the kernel to finish, and returns any errors.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d!\n", cudaStatus);

goto Error;

}

// Copy output matrix from GPU buffer to host memory.

cudaStatus = cudaMemcpy(C, dev\_C, M\*N \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_C);

cudaFree(dev\_A);

cudaFree(dev\_B);

return cudaStatus;

}

//初始化矩阵

void init(int rowNum, int colNum, int\* matrix) {

srand((unsigned int)time(0));

for (int i = 0; i < rowNum \* colNum; i++) matrix[i] = rand() % 200;

}

int main() {

int\* A = (int\*) malloc(M \* K \* sizeof(int)); //A[M,K]

int\* B = (int\*) malloc(K \* N \* sizeof(int)); //B[K,N]

int\* C = (int\*) malloc(M \* N \* sizeof(int)); //C[M,N] 结果矩阵

//initialize matrix A and B

init(M, K, A);

init(K, N, B);

clock\_t start\_clock, finish\_clock; //for timing

double duration\_seconds;

start\_clock = clock();

cudaError\_t cudaStatus = multiplyWithCuda(C, A, B);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "multiplyWithCuda failed!");

return 1;

}

finish\_clock = clock();

duration\_seconds = (double)(finish\_clock - start\_clock) / CLOCKS\_PER\_SEC;

printf("CUDA use time (s): %f\n", duration\_seconds);

printf("================================================\n");

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceReset failed!");

return 1;

}

check\_matrix(1023, 511, A, B, C);//检查C[1023,511]是否计算正确

free(A);

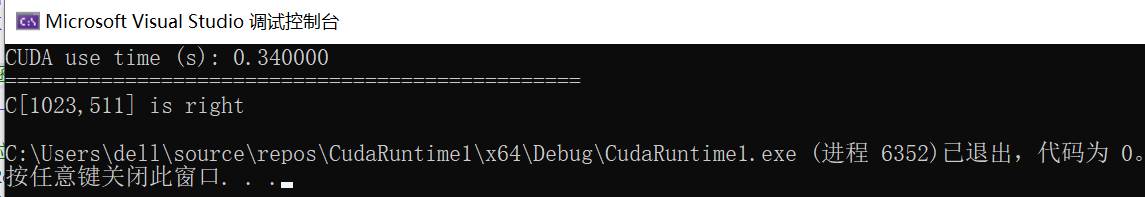
free(B);

free(C);

return 0;

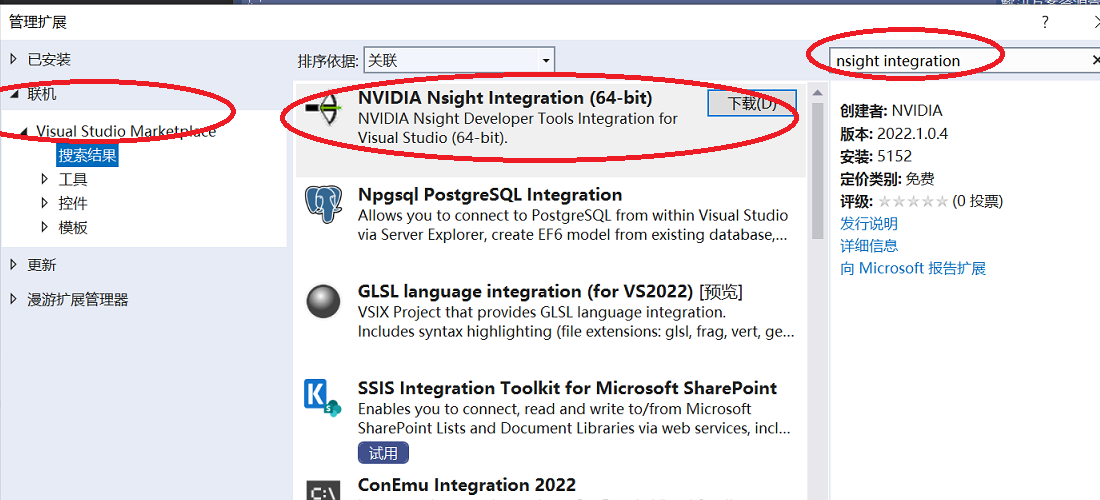
}

编译运行结果：



## 性能分析

如果在VS2022的“扩展”menu中没有看到Nsight Compute和Nsight System，请通过“管理扩展” sub-menu来安装“NVIDIA Nsight Integration”，如下图：

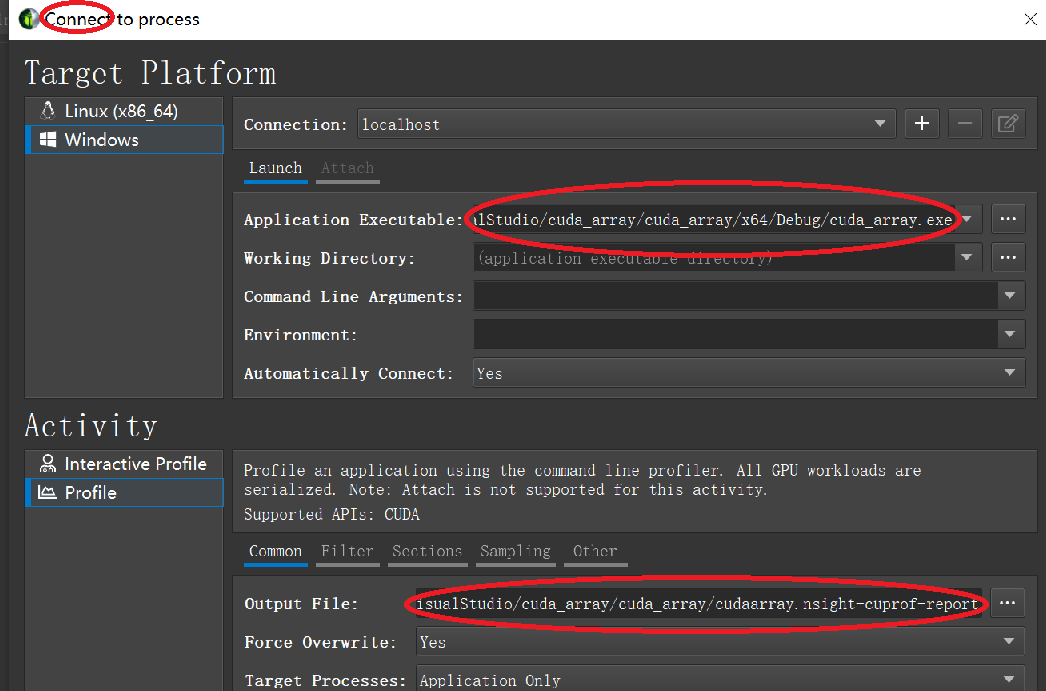


### 2.1、用Nsight Compute(ncu)进行性能分析

Nsight Compute 是NVIDIA用于监测 kernel 内部信息的工具，是一个交互式的CUDA程序的kernel profiler，可以输出每个kernel的SASS汇编、运行时间等非常详细的内容。ncu通过intra kernel的hardware counters获取细粒度性能参数。

ncu在profile的时候，通过–section来指定想使用的section。这里的section是指一组hardware counter的集合，可以通过nsightcompute/profile/sections/来看ncu自带的section有哪些。如果未指定section，则按照与默认的set关联的section来收集hw counters的信息。

使用命令 ncu --list-sets可以列出列表，看到不同的set关联的不同的sections。



设置好Application Executable和Output File，然后“Launch”就可以得到一份分析报告。

如果出现“permission”错误，那么试试用“管理员”身份运行Nsight Compute；如果出现“Profiling is not supported on device 0. To find out supported GPUs refer --list-chips option”，那么查看一下GPU设备是否在支持列表中。例如，Geforce 1050 Ti就不被Nsight Compute 2020之后的版本所支持，这个卡是Pascal-microarchitecture GPUs (Compute Capability 6.x) ，只被2019.5.1之前版本的Nsight Compute所支持。可以考虑安装老版本的Nsight Compute，它可以集成在Visual Studio中，也可以独立安装，但要注意与CUDA版本兼容。

### 2.2、NVIDIA Nsight Systems, CUDA 跟踪工具

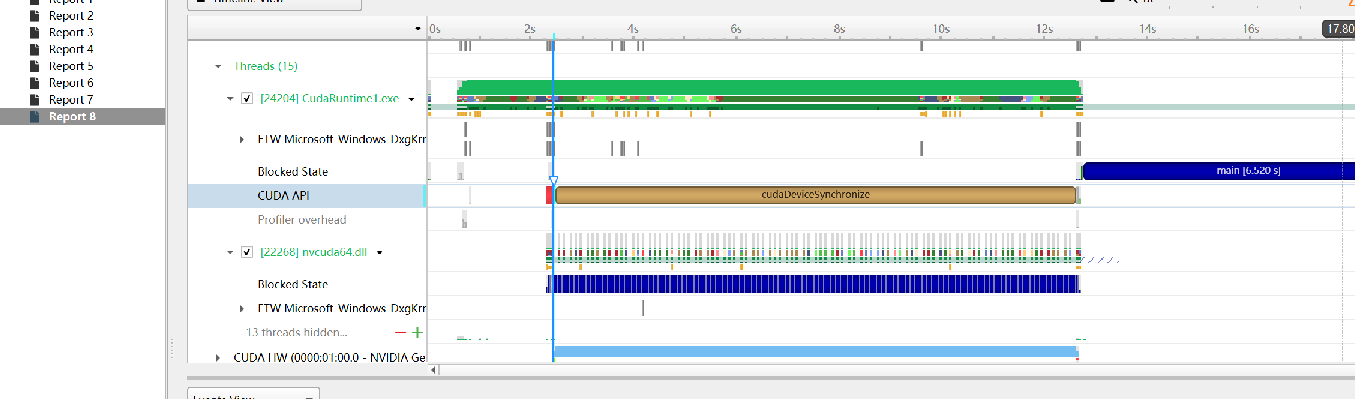
（https://docs.nvidia.com/nsight-systems/UserGuide/index.html）

Nsight systems是系统层面的分析工具，除了分析GPU的使用，还可以分析CPU的使用以及CPU和GPU的交互情况，可以捕捉CPU和GPU的各种事件，发现CPU和GPU上的等待以及不必要的同步，可以通过Nsight systems将任务均匀的分配到CPU和GPU上，支持linux和Windows。Nsight Systems 是nvprof的继任者，可以用于监测 kernel timeline。Kernel Timeline 输出的是以gpu kernel 为单位的一段时间的运行时间线，通过它可以观察GPU在什么时候有闲置或者利用不够充分的行为，也可以计算 kernel 运行的总时间，以准确地定位需要优化的问题。

能够跟踪（trace）程序的CUDA信息,可以在报告的时间轴上收集和呈现以下数据：

* CUDAAPI跟踪，跟踪应用程序的CUDA运行时和CUDA驱动程序调用。CUDA运行时调用通常以CUDA前缀开始（例如cudaLaunch）；CUDA驱动程序调用通常以cu前缀开始（例如，cuDeviceGetCount）。
* 跟踪主机上发生的操作， 例如拷贝内存操作。在使用CUDAAPI的线程中，时间轴树中将显示其他子行。

获取CUDA信息最简单的方法是从NVIDIA Nsight Systems启动进程，只需设置正常启动并选中Collect CUDA trace复选框。



## 练习

编写如下程序：随机生成两个大小为10000的数组，设为A数组和B数组，设结果数组C，利用CUDA计算：

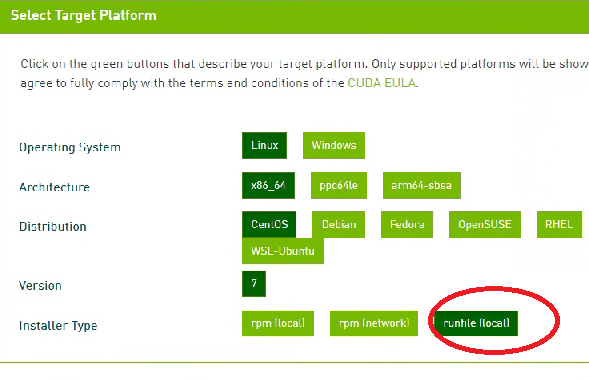


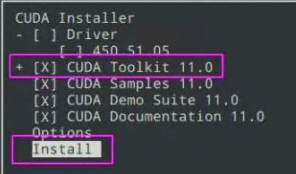
要求在CUDA的Kernel函数不使用if 判断语句，结果数组C输出到文件中。分析比较“使用GPU”和“不使用GPU”这两种版本的程序的性能。

# 附录：Linxu操作系统上的CUDA程序开发环境配置

1. 下载兼容OS的GPU设备驱动，https://www.nvidia.cn/Download/index.aspx
2. 下载合适的CUDA，参见：[CUDA Installation Guide for Linux (nvidia.com)](https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html)。“runfile(local)”类型比较简单，可以下载这个版本。注意安装前要求已经有合适版本的gcc kernel-hearders kernel-develop。

执行“sh XXXXX.run"，则开始自动安装。安装CUDA时，不要选驱动，驱动先事前安装好。





1. 配置环境（~/.bashrc添加[环境变量](https://so.csdn.net/so/search?q=%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Mr_yangsc/article/details/_blank)）

检查cuda是否安装成功：nvcc -V

如果没有这个命令的话，需要配置。（同时也要去看安装的路径下是否有nvcc）

在.bashrc 中添加跟CUDA有关的搜索路径

CUDA\_HOME

LD\_LIBRARY\_PATH

PATH

执行source ./bashrc令环境设置生效，现在可以成功执行 nvcc -V。

# **附录：没有GPGPU卡的，可以通过Docker容器方式使用GPGPU-Sim模拟器**

**（下述内容在CentOS7虚拟机上验证过，2023.12.11）**

1）检查是否安装了Docker，如果没有，请参考下述步骤完成安装和启动Docker。

主要步骤包括：

* 设置一个yum源（阿里仓库）

#yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

* 选择docker版本并安装

#yum list docker-ce --showduplicates | sort -r

#yum -y install docker-ce-18.03.1.ce （不限于这个版本）

* 检查是否成功

#docker version

* 启动Docker

#systemctl start docker

* 可以设置开机启动Docker

#systemctl enable docker

1. 参考下述步骤启动一个GPGPU-sim容器，在容器内进行cuda程序测试

（假设当前用户身份是root，下述内容摘自  
https://blog.csdn.net/eloudy/article/details/132826486）

* 拉取 docker image

#docker pull socalucr/gpgpu-sim

* 启动容器

# export NEW\_DIR=gpgpu-sim\_001 \

&& export WORK\_SPACE=/home/hipper/workspace/bitbucket \

&& mkdir -p ${WORK\_SPACE}/${NEW\_DIR} \

&& cd ${WORK\_SPACE}/${NEW\_DIR} \

&& sudo docker run --name ${NEW\_DIR}\_Jim -it \

-v ${WORK\_SPACE}/${NEW\_DIR}:${WORK\_SPACE}/${NEW\_DIR} \

-v /dev:/dev -v /usr/src/:/usr/src -v /lib/modules/:/lib/modules --privileged --cap-add=ALL \

socalucr/gpgpu-sim:latest /bin/bash

* 编译gpgpu-sim

在容器中，编译gpgpu-sim：

# cd /root/gpgpu-sim\_distribution

# make clean

# source ./setup\_environment

# make

* 运行 cuda程序

先按照上述步骤编译gpgpu-sim，才能编译运行示例程序，运行时需要动态调用 libcudart.so.4

libcudart.so.4 => /root/gpgpu-sim\_distribution/lib/gcc-4.4.7/cuda-4000/release/libcudart.so.4

a） 测试一个光线跟踪程序

# cd /root/ispass2009-benchmarks

# make -j -f Makefile.ispass-2009

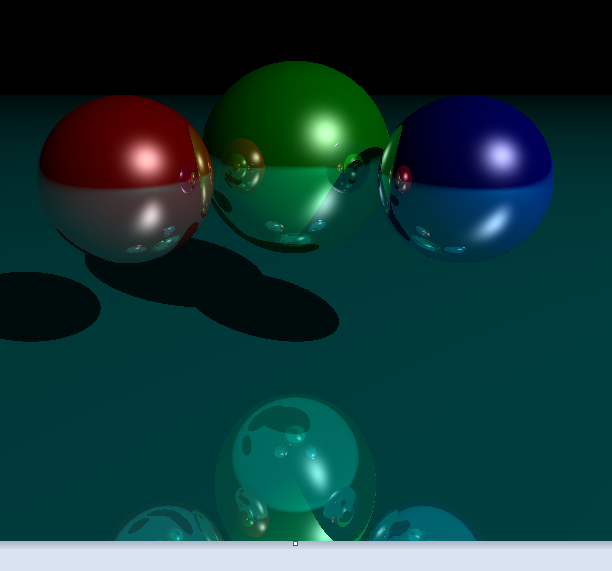
# mkdir /root/test03

# cd /root/test03/

# cp /root/gpgpu-sim\_distribution/configs/GTX480/\* ./ （这里假设要模拟GTX480显卡）

# /root/ispass2009-benchmarks/bin/release/RAY 640 640

会生成一张640\*640的图片output.img：



b) 测试向量加程序

# cd /root/NVIDIA\_GPU\_Computing\_SDK/C/

# make -j

# mkdir ~/test\_vectorAdd\_01

# cd ~/test\_vectorAdd\_01/

# cp ~/gpgpu-sim\_distribution/configs/GTX480/\* ./

# /root/NVIDIA\_GPU\_Computing\_SDK/C/bin/linux/release/vectorAdd

