# [GPU 的硬件基本概念，Cuda和Opencl名词关系对应](https://www.cnblogs.com/biglucky/p/3754800.html)

## GPU 的硬件基本概念

## Nvidia的版本CUDA

实际上在 nVidia 的 GPU 里，最基本的处理单元是所谓的 SP(Streaming Processor)，而一颗 nVidia 的 GPU 里，会有非常多的 SP 可以同时做计算;而数个 SP 会在附加一些其他单元，一起组成一个 SM(Streaming Multiprocessor)。几个 SM 则会在组成所谓的 TPC(Texture Processing Clusters)。

在 G80/G92 的架构下，总共会有 128 个 SP，以 8 个 SP 为一组，组成 16 个 SM，再以两个 SM 为一个 TPC，共分成 8 个 TPC 来运作。而在新一代的 GT200 里，SP 则是增加到 240 个，还是以 8 个 SP 组成一个 SM，但是改成以 3 个 SM 组成一个 TPC，共 10 组 TPC。

而在 CUDA 中，应该是没有 TPC 的那一层架构，而是只要根据 GPU 的 SM、SP 的数量和资源来调整就可以了。

如果把 CUDA 的 Grid - Block - Thread 架构对应到实际的硬件上的话，会类似对应成 GPU - Streaming Multiprocessor - Streaming Processor;一整个 Grid 会直接丢给 GPU 来执行，而 Block 大致就是对应到 SM，thread 则大致对应到 SP。当然，这个讲法并不是很精确，只是一个简单的比喻而已。

## AMD 版本OPENCL架构

另外work-item对应硬件上的一个PE（processing element）,而一个work-group对应硬件上的一个CU（computing unit）。这种对应可以理解为，一个work-item不能被拆分到多个PE上处理；同样，一个work-group也不能拆分到多个CU上同时处理。当映射到OpenCL硬件模型上时，每一个work-item运行在一个被称为处理基元（processing element）的抽象硬件单元上，其中每个处理基元可以处理多个work-item(注：摘自《OpenCL异构计算》P87)。（如此而言，是不是说对于二维的globalx必须是localx的整数倍，globaly必须是localy的整数倍？那么如果我数据很大，work-item所能数量很多，如果一个group中work-item的数量不超过CU中PE的个数，那么group的数量就可能很多；如果我想让group数量小点，那work-item的数目就会很多，还能不能处理了呢？这里总是找不多一个权威的解释，还请高手指点！针对group和item的问题）

对应CUDA组织多个workgroup,每个workgroup划分为多个thread.

由于硬件的限制，比如cu中pe数量的限制，实际上workgroup中线程并不是同时执行的，而是有一个调度单位，同一个workgroup中的线程，按照调度单位分组，然后一组一组调度硬件上去执行。这个调度单位在nvidia的硬件上称作warp,在AMD的硬件上称作wavefront，或者简称为wave

## 总结

首先解释下Cuda中的名词：

Block: 相当于opencl 中的work-group

Thread：相当于opencl 中的work-item

SP: 相当于opencl 中的PE

SM: 相当于opencl 中的CU

warp:相当于opencl 中的wavefront(简称wave).

|  |  |  |
| --- | --- | --- |
| 比较 | CUDA | OpenCL |
|  | block | work-group |
|  | thread | work-item |
|  | sp | PE |
|  | sm | CU |
|  | warp | wavefront |

# OpenCL 和 CUDA 之间的区别

OpenCL马上就要发布了, 根据nvidia的官方文档,对OpenCL和CUDA的异同做比较:

## 指针遍历

OpenCL不支持CUDA那样的指针遍历方式, 你只能用下标方式间接实现指针遍历. 例子代码如下:

// CUDA

struct Node { Node\* next; }

n = n->next;

// OpenCL

struct Node { unsigned int next; }

n = bufBase + n;

## Kernel程序异同

CUDA的代码最终编译成显卡上的二进制格式，最后由cudart.dll装载到GPU并且执行。OpenCL中运行时库中包含编译器，使用伪代码，程序运行时即时编译和装载。这个类似JAVA, .net 程序，道理也一样，为了支持跨平台的兼容。kernel程序的语法也有略微不同，如下：

\_\_global\_\_ void vectorAdd(const float \* a, const float \* b, float \* c)

{ // CUDA

int nIndex = blockIdx.x \* blockDim.x + threadIdx.x;

c[nIndex] = a[nIndex] + b[nIndex];

}

\_\_kernel void vectorAdd(\_\_global const float \* a, \_\_global const float \* b, \_\_global float \* c)

{ // OpenCL

int nIndex = get\_global\_id(0);

c[nIndex] = a[nIndex] + b[nIndex];

}

可以看出大部分都相同。只是细节有差异：

1）CUDA的kernel函数使用“\_\_global\_\_”申明而OpenCL的kernel函数使用“\_\_kernel”作为申明。

2）OpenCL的所有参数都有“\_\_global”修饰符，代表这个参数所指地址是在全局内存。

3）众所周知，CUDA采用threadIdx.{x|y|z}, blockIdx.{x|y|z}来获得当前线程的索引号，而OpenCL通过一个特定的get\_global\_id()函数来获得在kernel中的全局索引号。OpenCL中如果要获得在当前work-group（对等于CUDA中的block）中的局部索引号，可以使用get\_local\_id()

## Host代码的异同

把上面的kernel代码编译成“vectorAdd.cubin”，CUDA调用方法如下：

const unsigned int cnBlockSize = 512;

const unsigned int cnBlocks = 3;

const unsigned int cnDimension = cnBlocks \* cnBlockSize;

CUdevice hDevice;

CUcontext hContext;

CUmodule hModule;

CUfunction hFunction;

// create CUDA device & context

cuInit(0);

cuDeviceGet(&hContext, 0); // pick first device

cuCtxCreate(&hContext, 0, hDevice));

cuModuleLoad(&hModule, “vectorAdd.cubin”);

cuModuleGetFunction(&hFunction, hModule, "vectorAdd");

// allocate host vectors

float \* pA = new float[cnDimension];

float \* pB = new float[cnDimension];

float \* pC = new float[cnDimension];

// initialize host memory

randomInit(pA, cnDimension);

randomInit(pB, cnDimension);

// allocate memory on the device

CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;

cuMemAlloc(&pDeviceMemA, cnDimension \* sizeof(float));

cuMemAlloc(&pDeviceMemB, cnDimension \* sizeof(float));

cuMemAlloc(&pDeviceMemC, cnDimension \* sizeof(float));

// copy host vectors to device

cuMemcpyHtoD(pDeviceMemA, pA, cnDimension \* sizeof(float));

cuMemcpyHtoD(pDeviceMemB, pB, cnDimension \* sizeof(float));

// setup parameter values

cuFuncSetBlockShape(cuFunction, cnBlockSize, 1, 1);

cuParamSeti(cuFunction, 0, pDeviceMemA);

cuParamSeti(cuFunction, 4, pDeviceMemB);

cuParamSeti(cuFunction, 8, pDeviceMemC);

cuParamSetSize(cuFunction, 12);

// execute kernel

cuLaunchGrid(cuFunction, cnBlocks, 1);

// copy the result from device back to host

cuMemcpyDtoH((void \*) pC, pDeviceMemC, cnDimension \* sizeof(float));

delete[] pA;

delete[] pB;

delete[] pC;

cuMemFree(pDeviceMemA);

cuMemFree(pDeviceMemB);

cuMemFree(pDeviceMemC);

OpenCL的代码以文本方式存放在“sProgramSource.cl” 调用方式如下：

const unsigned int cnBlockSize = 512;

const unsigned int cnBlocks = 3;

const unsigned int cnDimension = cnBlocks \* cnBlockSize;

// create OpenCL device & context

cl\_context hContext;

hContext = clCreateContextFromType(0, CL\_DEVICE\_TYPE\_GPU, 0, 0, 0);

// query all devices available to the context

size\_t nContextDescriptorSize;

clGetContextInfo(hContext, CL\_CONTEXT\_DEVICES, 0, 0, &nContextDescriptorSize);

cl\_device\_id \* aDevices = malloc(nContextDescriptorSize);

clGetContextInfo(hContext, CL\_CONTEXT\_DEVICES, nContextDescriptorSize, aDevices, 0);

// create a command queue for first device the context reported

cl\_command\_queue hCmdQueue;

hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

// create & compile program

cl\_program hProgram;

hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource.cl, 0, 0);

clBuildProgram(hProgram, 0, 0, 0, 0, 0); // create kernel

cl\_kernel hKernel;

hKernel = clCreateKernel(hProgram, “vectorAdd”, 0);

// allocate host vectors

float \* pA = new float[cnDimension];

float \* pB = new float[cnDimension];

float \* pC = new float[cnDimension];

// initialize host memory

randomInit(pA, cnDimension);

randomInit(pB, cnDimension);

// allocate device memory

cl\_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;

hDeviceMemA = clCreateBuffer(hContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, cnDimension \* sizeof(cl\_float), pA, 0);

hDeviceMemB = clCreateBuffer(hContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, cnDimension \* sizeof(cl\_float), pA, 0);

hDeviceMemC = clCreateBuffer(hContext,

CL\_MEM\_WRITE\_ONLY,

cnDimension \* sizeof(cl\_float), 0, 0);

// setup parameter values

clSetKernelArg(hKernel, 0, sizeof(cl\_mem), (void \*)&hDeviceMemA);

clSetKernelArg(hKernel, 1, sizeof(cl\_mem), (void \*)&hDeviceMemB);

clSetKernelArg(hKernel, 2, sizeof(cl\_mem), (void \*)&hDeviceMemC);

// execute kernel

clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// copy results from device back to host

clEnqueueReadBuffer(hContext, hDeviceMemC, CL\_TRUE, 0, cnDimension \* sizeof(cl\_float), pC, 0, 0, 0);

delete[] pA;

delete[] pB;

delete[] pC;

clReleaseMemObj(hDeviceMemA);

clReleaseMemObj(hDeviceMemB);

clReleaseMemObj(hDeviceMemC);

## 初始化设备的异同

CUDA 在使用任何API之前必须调用cuInit(0)，然后是获得当前系统的可用设备并获得Context。  
cuInit(0);  
cuDeviceGet(&hContext, 0);  
cuCtxCreate(&hContext, 0, hDevice));

OpenCL不用全局的初始化，直接指定设备获得句柄就可以了  
cl\_context hContext;  
hContext = clCreateContextFromType(0, CL\_DEVICE\_TYPE\_GPU, 0, 0, 0);

设备创建完毕后，可以通过下面的方法获得设备信息和上下文：  
size\_t nContextDescriptorSize;  
clGetContextInfo(hContext, CL\_CONTEXT\_DEVICES, 0, 0, &nContextDescriptorSize);  
cl\_device\_id \* aDevices = malloc(nContextDescriptorSize);

clGetContextInfo(hContext, CL\_CONTEXT\_DEVICES, nContextDescriptorSize, aDevices, 0);

OpenCL introduces an additional concept: Command Queues. Commands launching kernels and reading or writing memory are always issued for a specific command queue. A command queue is created on a specific device in a context. The following code creates a command queue for the device and context created so far:  
cl\_command\_queue hCmdQueue;  
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

With this the program has progressed to the point where data can be uploaded to the device’s memory and processed by launching compute kernels on the device.

## 核函数创建的异同

CUDA kernel 以二进制格式存放与CUBIN文件中间，其调用格式和DLL的用法比较类似，先装载二进制库，然后通过函数名查找函数地址，最后用将函数装载到GPU运行。示例代码如下：  
CUmodule hModule;  
cuModuleLoad(&hModule, “vectorAdd.cubin”);  
cuModuleGetFunction(&hFunction, hModule, "vectorAdd");

OpenCL 为了支持多平台，所以不使用编译后的代码，采用类似JAVA的方式，装载文本格式的代码文件，然后即时编译并运行。需要注意的是，OpenCL也提供API访问kernel的二进制程序，前提是这个kernel已经被编译并且放在某个特定的缓存中了。

// 装载代码，即时编译  
cl\_program hProgram;  
hProgram = clCreateProgramWithSource(hContext, 1, “vectorAdd.c", 0, 0);  
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// 获得kernel函数句柄  
cl\_kernel hKernel;  
hKernel = clCreateKernel(hProgram, “vectorAdd”, 0);

## 设备内存分配

内存分配没有什么大区别，OpenCL提供两组特殊的标志，CL\_MEM\_READ\_ONLY 和 CL\_MEM\_WRITE\_ONLY 用来控制内存的读写权限。另外一个标志比较有用：CL\_MEM\_COPY\_HOST\_PTR 表示这个内存在主机分配，但是GPU可以使用，运行时会自动将主机内存内容拷贝到GPU，主机内存分配，设备内存分配，主机拷贝数据到设备，3个步骤一气呵成。  
// CUDA

CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;  
cuMemAlloc(&pDeviceMemA, cnDimension \* sizeof(float));  
cuMemAlloc(&pDeviceMemB, cnDimension \* sizeof(float));  
cuMemAlloc(&pDeviceMemC, cnDimension \* sizeof(float));  
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension \* sizeof(float));  
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension \* sizeof(float));

// OpenCL  
hDeviceMemA = clCreateBuffer(hContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, cnDimension \* sizeof(cl\_float), pA, 0);  
hDeviceMemB = clCreateBuffer(hContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, cnDimension \* sizeof(cl\_float), pA, 0);  
hDeviceMemC = clCreateBuffer(hContext, CL\_MEM\_WRITE\_ONLY, cnDimension \* sizeof(cl\_float), 0, 0);

## 核函数参数定义的异同

The next step in preparing the kernels for launch is to establish a mapping between the kernels’parameters, essentially pointers to the three vectors A, B and C, to the three device memory regions,which were allocated in the previous section.  
Parameter setting in both APIs is a pretty low-level affair. It requires knowledge of the total number, order, and types of a given kernel’s parameters. The order and types of the parameters are used todetermine a specific parameters offset inside the data block made up of all parameters. The offset in bytes for the n-th parameter is essentially the sum of the sizes of all (n-1) preceding parameters.  
Using the CUDA Driver API:  
In CUDA device pointers are represented as unsigned int and the CUDA Driver API has a dedicated method for setting that type. Here’s the code for setting the three parameters. Note how the offset is incrementally computed as the sum of the previous parameters’ sizes.  
cuParamSeti(cuFunction, 0, pDeviceMemA);  
cuParamSeti(cuFunction, 4, pDeviceMemB);  
cuParamSeti(cuFunction, 8, pDeviceMemC);  
cuParamSetSize(cuFunction, 12);

Using OpenCL:  
In OpenCL parameter setting is done via a single function that takes a pointer to the location of the parameter to be set.  
clSetKernelArg(hKernel, 0, sizeof(cl\_mem), (void \*)&hDeviceMemA);  
clSetKernelArg(hKernel, 1, sizeof(cl\_mem), (void \*)&hDeviceMemB);  
clSetKernelArg(hKernel, 2, sizeof(cl\_mem), (void \*)&hDeviceMemC);

## 核函数启用的异同

Launching a kernel requires the specification of the dimension and size of the “thread-grid”. The CUDA Programming Guide and the OpenCL specification contain details about the structure of those grids. For NVIDIA GPUs the permissible structures are the same for CUDA and OpenCL.

For the vectorAdd sample we need to start one thread per vector-element (of the output vector). The number of elements in the vector is given in the cnDimension variable. It is defined to be cnDimension = cnBlockSize \* cnBlocks. This means that cnDimension threads need to be executed. The threads are structured into cnBlocks one-dimensional thread blocks of size cnBlockSize.  
Using the CUDA Driver API:

A kernel’s block size is specified in a call separate from the actual kernel launch using cuFunctSetBlockShape. The kernel launching function cuLaunchGrid then only  
specifies the number of blocks to be launched.

cuFuncSetBlockShape(cuFunction, cnBlockSize, 1, 1);  
cuLaunchGrid (cuFunction, cnBlocks, 1);

Using OpenCL:  
The OpenCL equivalent of kernel launching is to “enqueue” a kernel for execution into a command queue. The enqueue function takes parameters for both the work group size (work group is the OpenCL equivalent of a CUDA thread-block), and the global work size, which is the size of the global array of threads.  
Note: Where in CUDA the global work size is specified in terms of number of thread  
blocks, it is given in number of threads in OpenCL.  
Both work group size and global work size are potentially one, two, or three dimensional arrays. The function expects pointers of unsigned ints to be passed in the fourth and fifth parameters.  
For the vectorAdd example, work groups and total work size is a one-dimensional grid of threads.  
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, &cnBlockSize, 0, 0, 0);  
The parameters of cnDimension and cnBlockSize must be pointers to unsigned int. Work group sizes that are dimensions greater than 1, the parameters will be a pointer to arrays of sizes.

## 结果数据传回的异同

Both kernel launch functions (CUDA and OpenCL) are asynchronous, i.e. they return immediately after scheduling the kernel to be executed on the GPU. In order for a copy operation that retrieves the result vector C (copy from device to host) to produce correct results in synchronization with the kernel completion needs to happen.  
CUDA memcpy functions automatically synchronize and complete any outstanding kernel launches proceeding. Both API’s also provide a set of asynchronous memory transfer functions which allows a user to overlap memory transfers with computation to increase throughput.  
Using the CUDA Driver API:  
Use cuMemcpyDtoH() to copy results back to the host.  
cuMemcpyDtoH((void \*)pC, pDeviceMemC, cnDimension \* sizeof(float));

Using OpenCL:  
OpenCL’s clEnqueueReadBuffer() function allows the user to specify whether a read is to be synchronous or asynchronous (third argument). For the simple vectorAdd sample a synchronizing read is used, which results in the same behavior as the simple synchronous CUDA memory copy above:  
clEnqueueReadBuffer(hContext, hDeviceC, CL\_TRUE, 0, cnDimension \* sizeof(cl\_float), pC, 0, 0, 0);

When used for asynchronous reads, OpenCL has an event mechanism that allows the host application to query the status or wait for the completion of a given call.

The current version of OpenCL does not support stream offsets at the API/kernel invocation level. Offsets must be passed in as a parameter to the kernel and the address of the memory computed inside it. CUDA kernels may be started at offsets within buffers at the API/kernel invocation level.