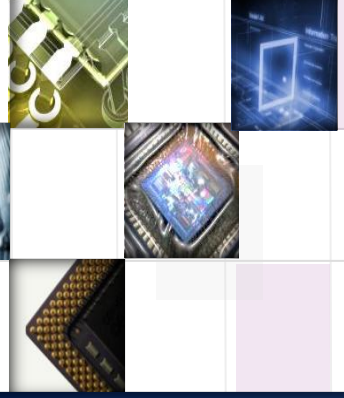


LEADCORE
联芯科技

OpenCL编程基础知识

张蕊

2016年3月

Part1：OpenCL简介

Part2：OpenCL编程框架

- 平台与设备
- 执行环境

Part3：应用示例：两向量相加

OpenCL简介

OpenCL（Open Computing Language，开放计算语言）是非盈利性技术联盟Khronos Group管理的异构编程框架。该框架用于开发可以在不同厂商所生产的各种设备上运行的应用程序。主要特点为：

➢1. 支持同构或**异构**体系平台编程，异构平台可由CPU、GPU以及其他类型的处理器组成，定义了主机端控制层和设备端语言。

➢2. 跨平台，程序**可移植性高**

➢3. 提供**任务并行**与**数据并行**计算方式

任务并行：配置不同的设备不同任务，每个任务处理不同数据

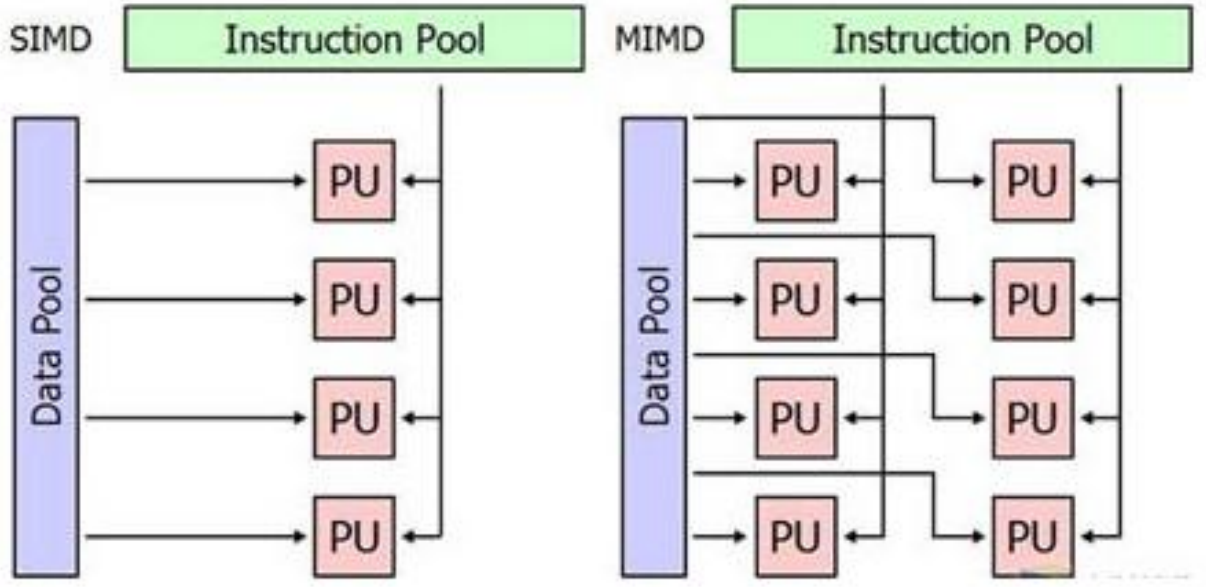
数据并行：各设备收到相同指令，处理数据集的不同部分

OpenCL简介

并行计算：

例如：for(i=0;…;i++)

CPU：多指令单数据流（MISD），擅长逻辑控制，**重复迭代多次**
GPU：单指令多数据流（SIMD），擅长并行运算，**只需执行一次**



OpenCL规范

- **平台模型**

异构系统的抽象硬件模型：宿主机(1)+可执行OpenCL的设备(>=1)

- **执行模型**

指令流在异构平台上执行的抽象表示：主机如何配置OpenCL环境以及在设备上如何执行kernel函数

- **内存模型**

内存类型的集合及不同内存类型交互的规范：定义被kernel所用的抽象内存层次，无需考虑实际底层内存框架

- **编程模型**

程序员设计算法来实现一个应用时的一种高层抽象：如何将并发模型映射到物理硬件上

OpenCL编程框架

1、平台模型

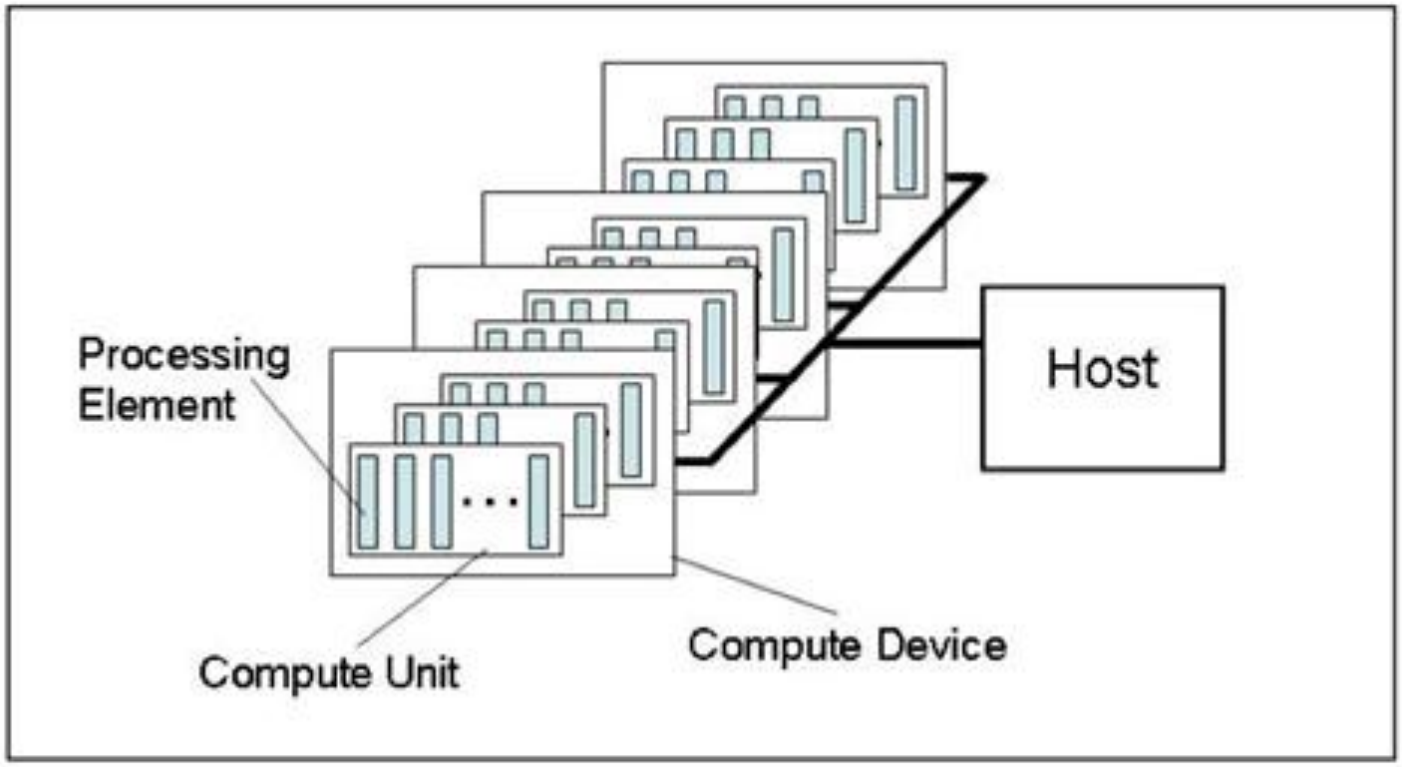
指定有一个处理器（主机Host）来协调程序的执行， 一个或多个处理器（设备Devices）来执行OpenCL C代码。 程序员编写OpenCL C函数（称之为kernel）并在不同的设备上执行。

宿主机(Host)：

设备(Device)：

计算单元(CU)：

处理单元(PE)：



OpenCL编程框架

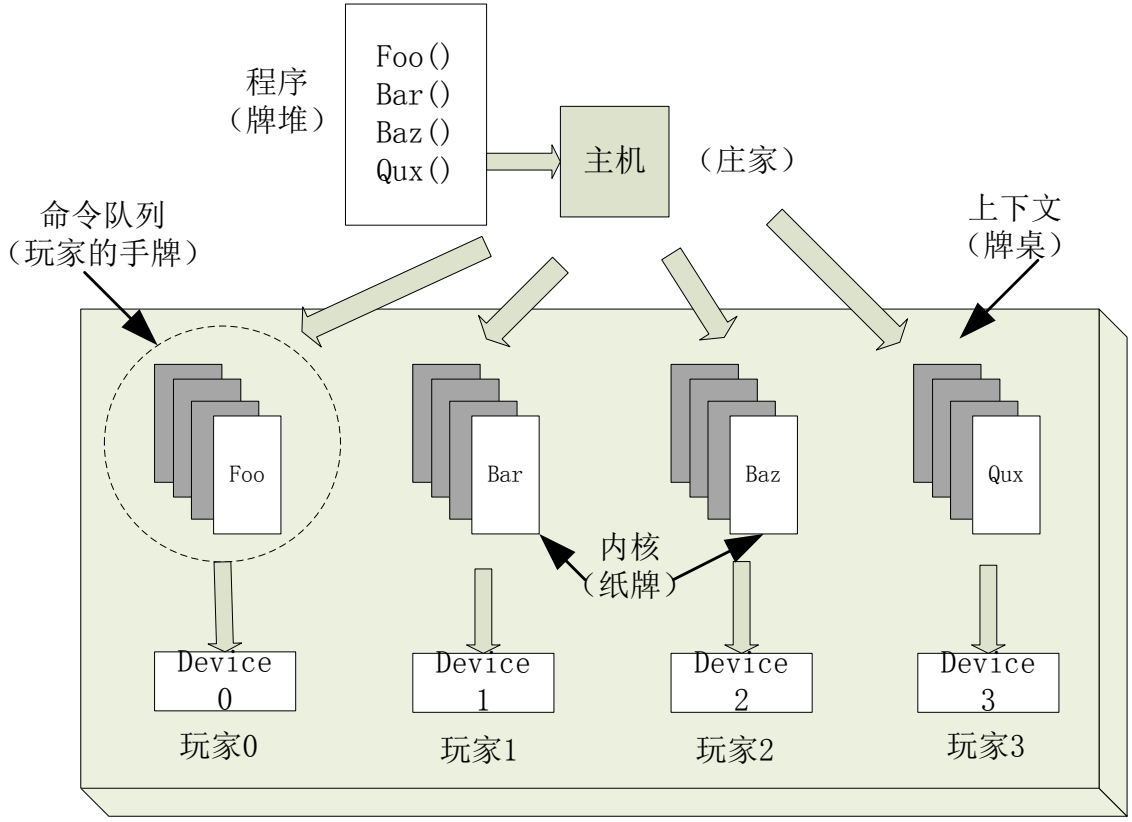
2、执行模型

上下文管理多个设备， 每个设备有一个命令队列，主机程序将内核程序提交到不同的命令队列上执行。

上下文：环境上下文，一个Context包含几个device（单个CPU或GPU）

程序：由一堆代码组成

命令队列：给每个Device提交的指令序列，主机通过命令队列与设备通信，包含许多kernel函数与数据处理命令

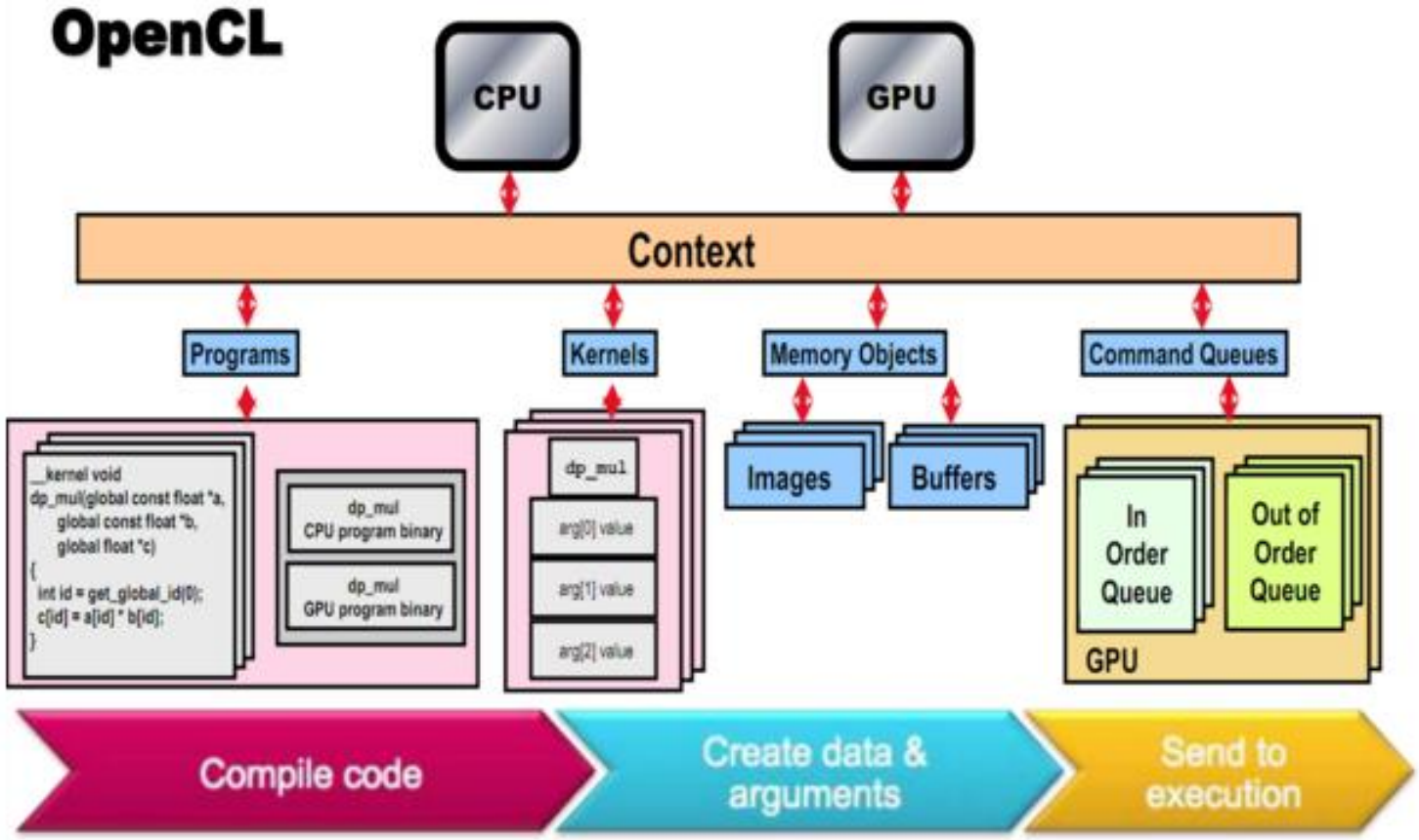


OpenCL处理与纸牌游戏

OpenCL编程框架

主机程序：运行在主机上的应用程序，一般由C/C++编写

内核程序：运行在设备上的程序，由OpenCL标准编写，是唯一可以从主机上调用执行的函数



OpenCL编程框架

3、内存模型

设备若想完成复杂的计算任务，还需要有待处理数据的缓存以及保存处理结果的缓存。OpenCL定义了抽象的内存模型，程序实现的时候只需关注抽象的内存模型，具体向硬件上的映射由驱动来完成。

- **buffer对象：** 一块地址连续的区域，类似于数组，可以作为任何与图像处理无关的数据载体

  Subbuffer对象： buffer对象的一部分

- **image对象：** 表示二维或三维的图像数据，仅限于图像数据相关数据类型，不能通过指针访问

OpenCL编程框架

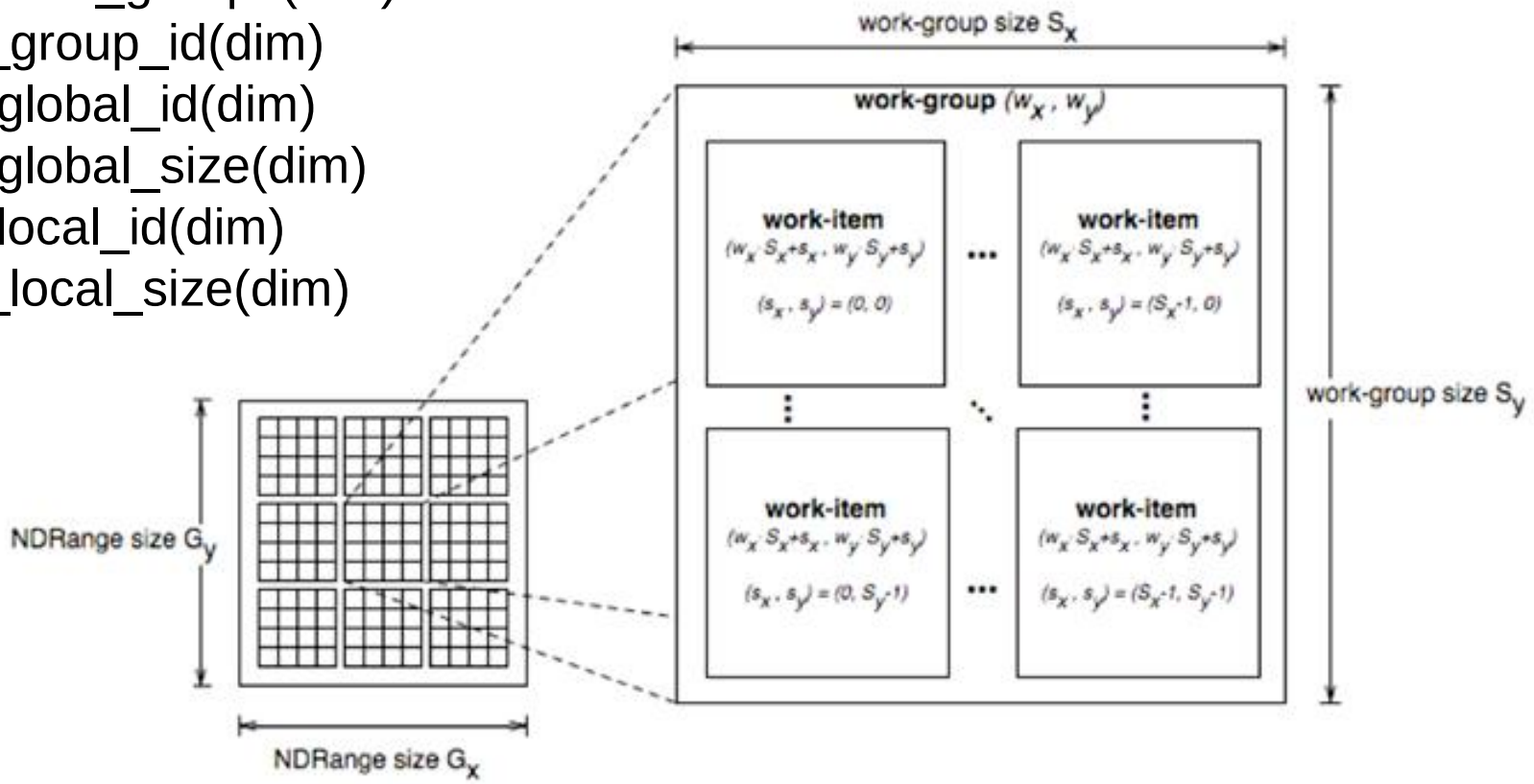
数据划分（与计算性能有关）

Workitem(也就是线程）：Kernel 的每个运行实例

Workgroup：workitem 组织在一起

OpenCL 中，每个workgroup 之间都是相互独立的。通过一个global id(唯一）或者一个workgroup id 和一个workgroup 内的local id，就能标定一个workitem。

- get_num_groups(dim)
- get_group_id(dim)
- get_global_id(dim)
- get_global_size(dim)
- get_local_id(dim)
- get_local_size(dim)



OpenCL编程框架

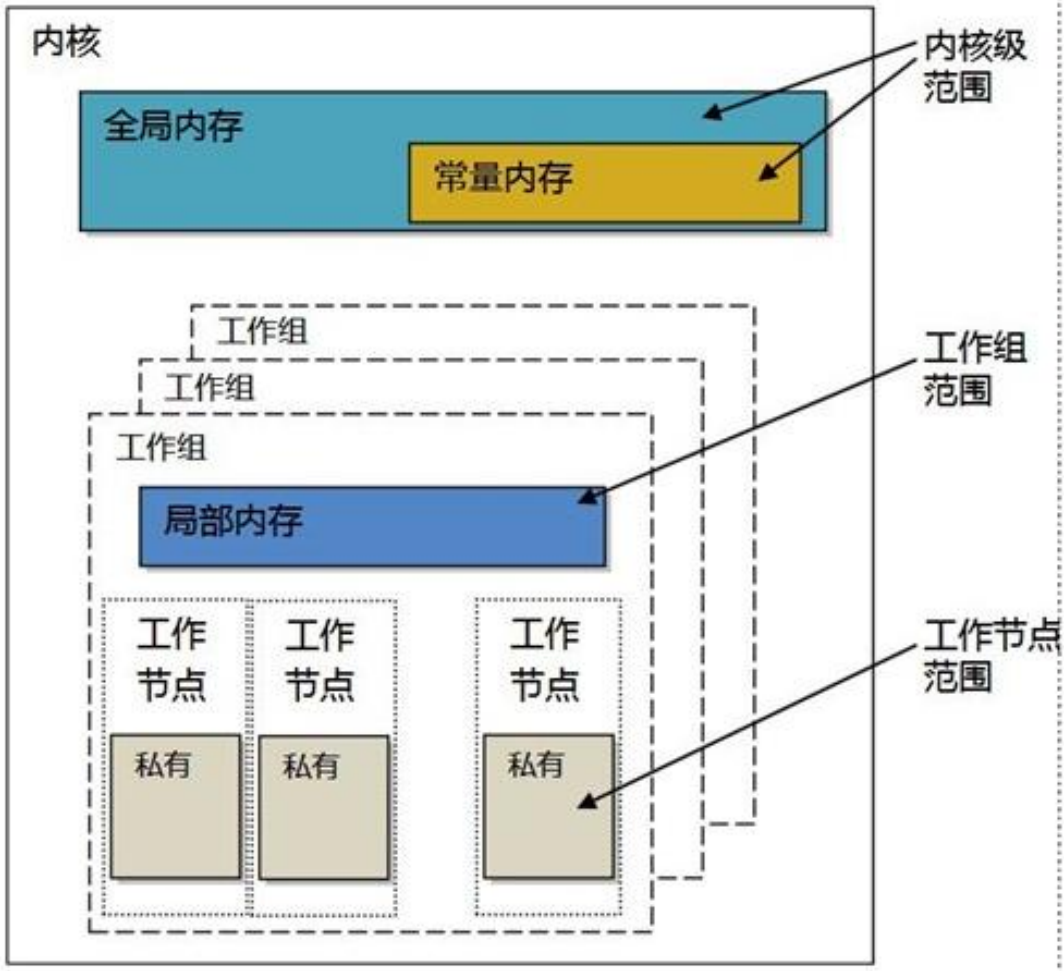
内存区域：

全局内存：所有工作组(workgroup)中的所有工作项(workitem)都可以对其进行读写。

常量内存：全局内存中的一块区域，在内核的执行过程中保持不变。 宿主机负责对此中内存对象的分配和初始化。

局部内存：隶属于一个工作组(workgroup)的内存区域，且对该工作组内的所有工作项(workitem)可访问。

私有内存：隶属于一个工作项的内存区域。



OpenCL编程框架

OpenCL同步

OpenCL采用宽松的同步模型和内存一致性模型

barrier：实现同一个workgroup内的workitem同步

event：实现在同一个上下文中命令间的同步

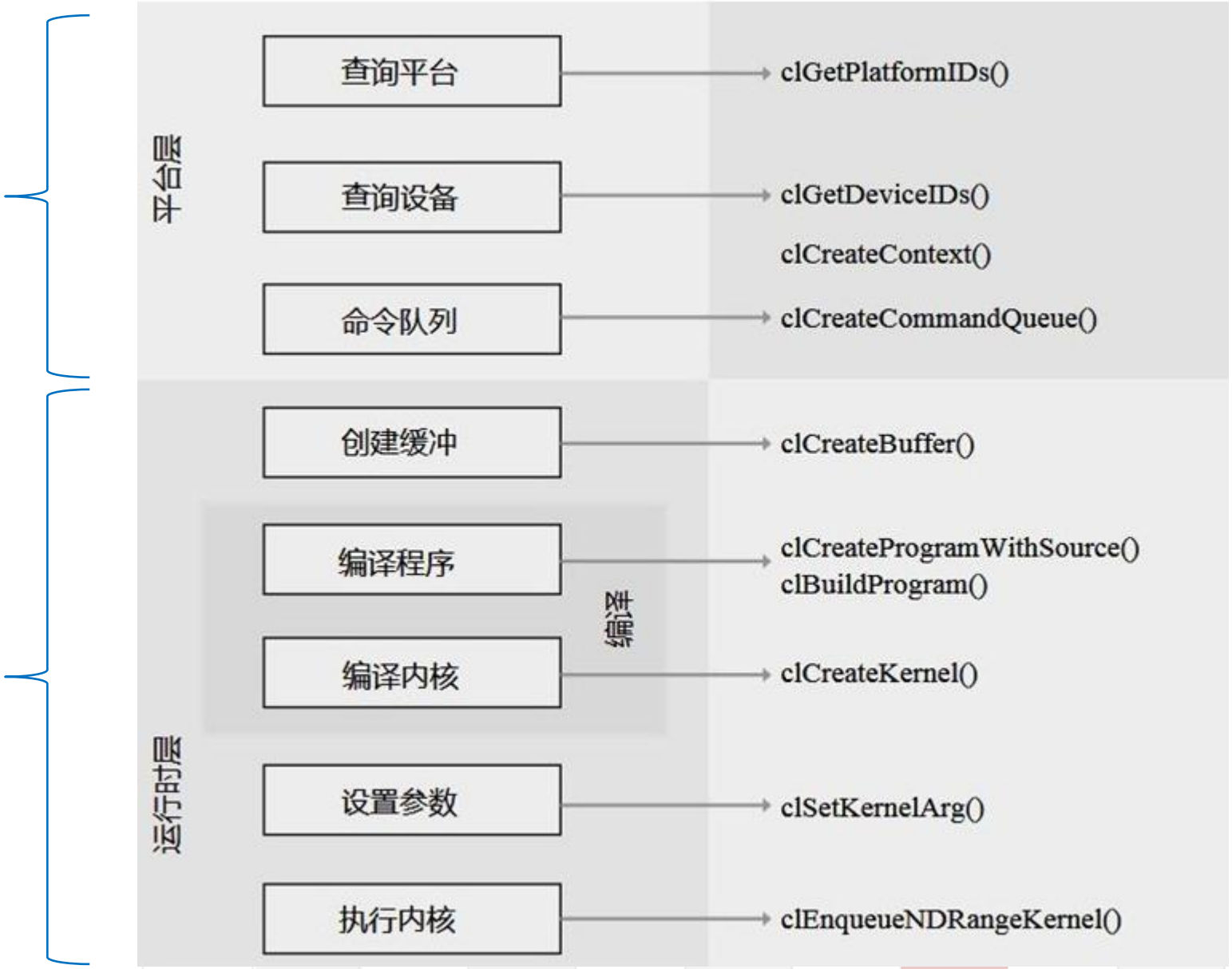
事件回调：在事件到达指定状态时触发

举例，主机在设备运行时为提高系统效率，将一个主机程序设定为回调函数。

clFinishi：实现设备间数据共享

OpenCL编程框架

OpenCL编程步骤



OpenCL编程框架

基本数据类型

标量数据类型	位宽	目的
cl_char	8	有符号补码表示的整数
cl_uchar	8	无符号补码表示的整数
cl_short	16	有符号补码表示的整数
cl_ushort	16	无符号补码表示的整数
cl_int	32	有符号补码表示的整数
cl_uint	32	无符号补码表示的整数
cl_long	64	有符号补码表示的整数
cl_ulong	64	无符号补码表示的整数
cl_half	16	半精度浮点数
cl_float	32	单精度浮点数
cl_double	64	双精度浮点数

六种新数据结构

数据结构	名称
cl_platform_id	平台
cl_device_id	设备
cl_context	上下文
cl_program	程序
cl_kernel	内核
cl_command_queue	命令队列

OpenCL编程框架

第一步：初始化OpenCL平台

cl_int clGetPlatformIDs(cl_unit num_entries，cl_platform_id* platforms，cl_uint* num_platforms);

Platforms：用来保存可用平台id
num_platforms：可用平台数
一般调用两次该函数：第一次获取可用平台数，第二次获取一个可用平台

第二步：选择设备

clGetDeviceIDs(cl_platform_id platform，cl_device_type device_type，cl_unit num_entries，cl_device_id* devices，cl_unit* num_devices);

device_type：设备类型，取值有CL_DEVICE_TYPE_ALL，CL_DEVICE_TYPE_DEFAULT， CL_DEVICE_TYPE_GPU等

一般也是调用两次

OpenCL编程框架

第三步：创建上下文

```
cl_context
clCreateContest(const cl_context_properties* properties,  cl_uint
num_devices,  const cl_device_id* devices,
void (CL_CALLBACK *pfn_notify)(…),
void* user_data,  cl_int* errcode_ret);
```

直接确定设备的方式来创建

或者

```
cl_context
clCreateContestFromType(constcl_context_properties*
properties,  cl_uint num_devices,  const cl_device_id* devices,
void (CL_CALLBACK *pfn_notify)(…),
void* user_data,  cl_int* errcode_ret);
```

通过给定设备的方式来创建

OpenCL编程框架

第四步：创建命令队列

cl_command_queue
clCreateCommandQueue(cl_context context, cl_device_id device,
cl_command_queue_properties properties, cl_int* errcode_ret);

Properties取值：
CL_QUEUE_PROFILING_ENABLE——使能性能分析事件
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE ——使能命令队列的乱序执行

OpenCL编程框架

第五步：创建数据缓冲区

```
cl_mem
clCreateBuffer(cl_context context,  cl_mem_flags flags,
size_t size,  void* host_ptr,  cl_int* errcode_ret);
```

flags：确定缓存对象的可访问性以及分配方式
size：　缓存大小
host_ptr：一般是一个有效的host buffer 对象，也可以是NULL
只有被clCreateBuffer分配的内存，才能被同时映射到CPU和GPU上的虚拟空间

OpenCL编程框架

第六步：将host数据写进设备缓冲区

```
cl_int
clEnqueueWriteBuffer(cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_write,
size_t offset,
size_t data_size,
const void* ptr,
cl_uint num_events_in_wait_list,
const cl_event* event_wait_list,
cl_event* event);
```

buffer：创建的缓存对象
data_size 、 **offset** ：传输数据的大小与起始位置

包括初始化内存对象以及把**host** 数据传到**device** 内存这两种操作

OpenCL编程框架

第七步：创建程序对象

```
cl_program
clCreateProgramWithSource(cl_context context,
cl_uint count,
const char** strings,
const size_t* src_sizes,
cl_int* errcode_ret);
```

strings：由多个文本文件组成的字符串指针数组
count：函数所需文本个数
src_sizes：每个文本字符串的大小
要求缓存中的代码是文本形式

还可以
cl_program **clCreateProgramWithBinary**(…) 创建

OpenCL编程框架

并且编译该程序

```
cl_int
clBuildProgram(cl_program program,
cl_uint num_devices,
const cl_device_id* device_list,
const char* options,
void (CL_CALLBACK *pfn_notify)
void* user_data)；
```

对context 中的每个设备，这个函数编译、连接源代码对象，产生device 可以执行的文件。

OpenCL编程框架

第八步：创建kernel对象

```
cl_kernel
clCreateKernel(cl_program program,
const char* kernel_name,
cl_int* errcode_ret);
```

kernel_name：指定函数名

Kernel对象包含Kernel函数及其参数列表。
写法示例：

```
__kernel void vecadd(__global
const float* A, __global const
float* B, __global
float* C)
{}
```

返回类型必须是void
参数必须声明内存类型

第九步：设置内核参数

```
cl_int
clSetKernelArg(cl_kernel kernel,
cl_uint arg_index,
size_t arg_size,
const void* arg_value);
```

arg_index：访问kernel的第几个参数
arg_value：缓存对象

OpenCL编程框架

第十步：kernel入队执行

```
cl_int
clEnqueueNDRangeKernel(cl_command_queue command_queue,
cl_kernel kernel,
cl_uint work_dim,
const size_t* global_work_offset,
const size_t* global_work_size,
const size_t* local_work_size,
cl_uint num_events_in_wait_list,
const cl_event* event_wait_list,
cl_event* event);
```

global_work_offset、 global_work_size、 local_work_size：完成数据的分组

OpenCL编程框架

第十一步：取回计算结果

cl_int clEnqueueReadBuffer(…)

第十二步：释放内存资源

clRelease…(…)

Note：所有使用clCreate申请的（缓冲区、kernel、队列等）必须使用clRelease释放

应用示例

两个向量相加：

```
#include "stdafx.h"
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <fstream>
using namespace std;
#define NWITEMS 262144
#pragma comment (lib,"OpenCL.lib")
//把文本文件读入一个string 中
int convertToString(const char *filename,
std::string& s)
{
size_t size;
char* str;
std::fstream f(filename, (std::fstream::in |
std::fstream::binary));
if(f.is_open())
{
size_t fileSize;
f.seekg(0, std::fstream::end);
size = fileSize = (size_t)f.tellg();
f.seekg(0, std::fstream::beg);
str = new char[size+1];
if(!str)
{
f.close();
return NULL; }
f.read(str, fileSize);
f.close();
str[size] = '\0';
s = str;
delete[] str;
return 0;
}
printf("Error: Failed to open file %s\n",
filename);
return 1;
}
```

应用示例

```
int main(int argc, char* argv[])
{
//在host 内存中创建三个缓冲区
float *buf1 = 0;
float *buf2 = 0;
float *buf = 0;
buf1 =(float *)malloc(NWITEMS *
sizeof(float));
buf2 =(float *)malloc(NWITEMS *
sizeof(float));
buf =(float *)malloc(NWITEMS *
sizeof(float));
//初始化buf1 和buf2 的内容
int i;
srand( (unsigned)time( NULL ) );
for(i = 0; i < NWITEMS; i++)
buf1[i] = rand()%65535;
srand( (unsigned)time( NULL )
+1000);
for(i = 0; i < NWITEMS; i++)
buf2[i] = rand()%65535;
cl_uint status;
cl_platform_id platform;
//创建平台对象
status = clGetPlatformIDs( 1, &platform,
NULL );
cl_device_id device;
//创建GPU 设备
clGetDeviceIDs( platform,
CL_DEVICE_TYPE_GPU,
1,
&device,
NULL);
//创建context
cl_context context =
clCreateContext( NULL,
1,
&device,
NULL, NULL, NULL);
);
```

应用示例

```
//创建命令队列
cl_command_queue queue =
clCreateCommandQueue( context,
device,
CL_QUEUE_PROFILING_ENABLE,
NULL );
//创建三个OpenCL 内存对象，并把
buf1 的内容通过隐式拷贝的方式
//拷贝到clbuf1 , buf2 的内容通过显
示拷贝的方式拷贝到clbuf2
cl_mem clbuf1 =
clCreateBuffer(context,
CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
NWITEMS*sizeof(cl_float),buf1,
NULL );
cl_mem clbuf2 =
clCreateBuffer(context,
CL_MEM_READ_ONLY ,
NWITEMS*sizeof(cl_float),NULL,
NULL );
status = clEnqueueWriteBuffer(queue,
clbuf2, 1,
0, NWITEMS*sizeof(cl_float), buf2, 0, 0, 0);
cl_mem buffer = clCreateBuffer( context,
CL_MEM_WRITE_ONLY,
NWITEMS * sizeof(cl_float),
NULL, NULL )
const char * filename = "add.cl";
std::string sourceStr;
status = convertToString(filename,
sourceStr);
const char * source = sourceStr.c_str();
size_t sourceSize[] = { strlen(source) };
//创建程序对象
cl_program program =
clCreateProgramWithSource(
context,1,&source,sourceSize,NULL);
```

内核函数名

```
//编译程序对象
status = clBuildProgram( program, 1,
&device, NULL, NULL, NULL );
if(status != 0)
{
printf("clBuild failed:%d\n", status);
char tbuf[0x10000];
clGetProgramBuildInfo(program, device,
CL_PROGRAM_BUILD_LOG, 0x10000, tbuf,
NULL);
printf("\n%s\n", tbuf);
return  -1;
}
//创建Kernel 对象
cl_kernel kernel = clCreateKernel( program,
"vecadd", NULL );
//设置Kernel 参数
cl_int clnum = NWITEMS;
clSetKernelArg(kernel, 0, sizeof(cl_mem),
(void*) &clbuf1);
clSetKernelArg(kernel, 1,
sizeof(cl_mem), (void*) &clbuf2);
clSetKernelArg(kernel, 2,
sizeof(cl_mem), (void*) &buffer);
//执行kernel
cl_event ev;
size_t global_work_size =
NWITEMS;
clEnqueueNDRangeKernel( queue,
kernel,1,NULL,&global_work_size,
NULL, 0, NULL, &ev);
clFinish( queue );
```

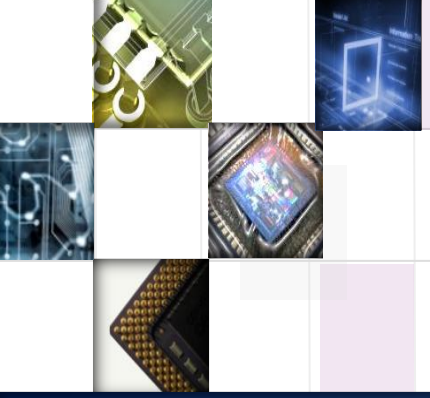
```
//数据拷回host 内存
cl_float *ptr;
ptr = (cl_float *) clEnqueueReadBuffer( queue,
buffer,
CL_TRUE,
CL_MAP_READ,
0,
NWITEMS * sizeof(cl_float),
0, NULL, NULL, NULL );

if(buf)
free(buf);
if(buf1)
free(buf1);
if(buf2)
free(buf2);
```

```
//删除OpenCL 资源对象
clReleaseMemObject(clbuf1);
clReleaseMemObject(clbuf2);
clReleaseMemObject(buffer);
clReleaseProgram(program);
clReleaseCommandQueue(q
ueue);
clReleaseContext(context);
return 0;
}
```

```
__kernel void vecadd(__global const float* A,
__global const float* B, __global
float* C)
{
int id = get_global_id(0);
C[id] = A[id] + B[id];
}
```

保存到add.cl文件中



LEADCORE
联芯科技

Thank You!