Лекция 4: Использование графических процессоров для ускорения вычислений. OpenCL

Н.Д. Смирнова

Санкт-Петербургский государственный Политехнический университет

22.10.2011

Содержание

Содержание

OpenCL - Open Computing Language

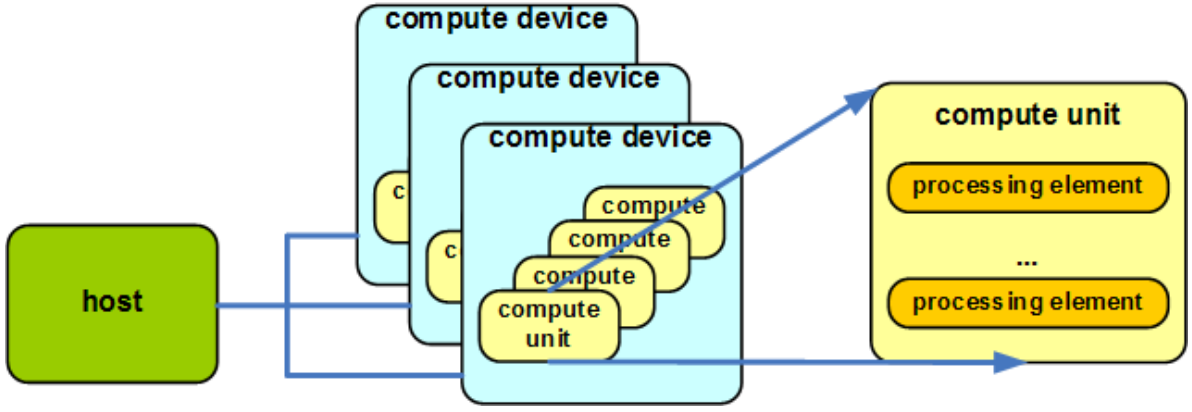
OpenCL - открытый стандарт для параллельного программирования на различных платформах[1]

- разработан Apple свместно с Khronos Group [2]
- бесплатный
- кросс-платформенный: платформа может состоять из разного количества CPU, GPU и других процессоров
- язык программирования: C99 с ограничениями и дополнениями
  - используется для написания kernel
- API
  - создание интерфейса для общения с GPU, CPU, ...
  - управление вычислениями: старт, синхронизация, ...
  - пересылка данных в/из памяти GPU

[1] http://en.wikipedia.org/wiki/OpenCL
[2] http://www.khronos.org/opencl/

Содержание

OpenCL - модель платформы (platform model)



- Host управляет одним или несколькими *вычислительныеми устройствами* (Compute Device)
  - Host - Windows, Linux, embedded system, ...
  - Kernel пишется на языке OpenCL и исполняется на девайсе
- Вычислительное устройство содержит один или несколько процессоров (Compute Unit)
- Процессор состоит из нескольких *обрабатывающих элементов* (Processing Element - исполняет код по принципу SIMD)

Содержание

OpenCL - модель исполнения команд(execution model)

Kernel

- C-функция, выполняемая на Compute Device
- точка входа (entry point), аргументы, **нет** возвращаемого значения

Program

- набор Kernel'ов и функций
- по сути, динамически подгружаемая библиотека

Command Queue

- организует очередь kernel'ов и других OpenCL команд (например, команды работы с памятью)
- можно запустить выполнение команд в очереди как по порядку так и не очень

Event

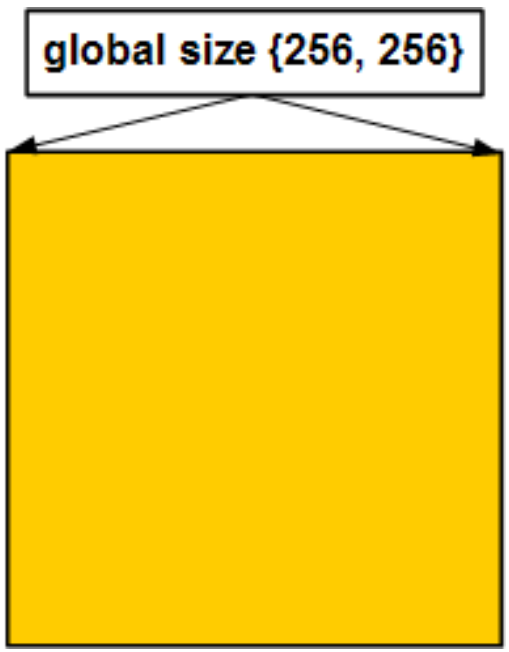
- событие, используемое для синхронизации как команд внутри очереди, так и нескольких очередей между собой

OpenCL - пространство индексов

- исполнение kernel'а для одной группы данных происходит в *потоке*
- индекс потока берется из 1-, 2- или 3-мерного двухуровневого пространства индексов - **NDRange**
- поток называется *рабочим элементом* (work-item)
- рабочие элементы формируют *рабочие группы* (work-group)
- нет никакой синхронизации между рабочими группами
- но есть специальные события - *барьеры* - для синхронизации рабочих элементов внутри рабочей группы
- размерность NDRange лучше выбирать подходящей для задачи

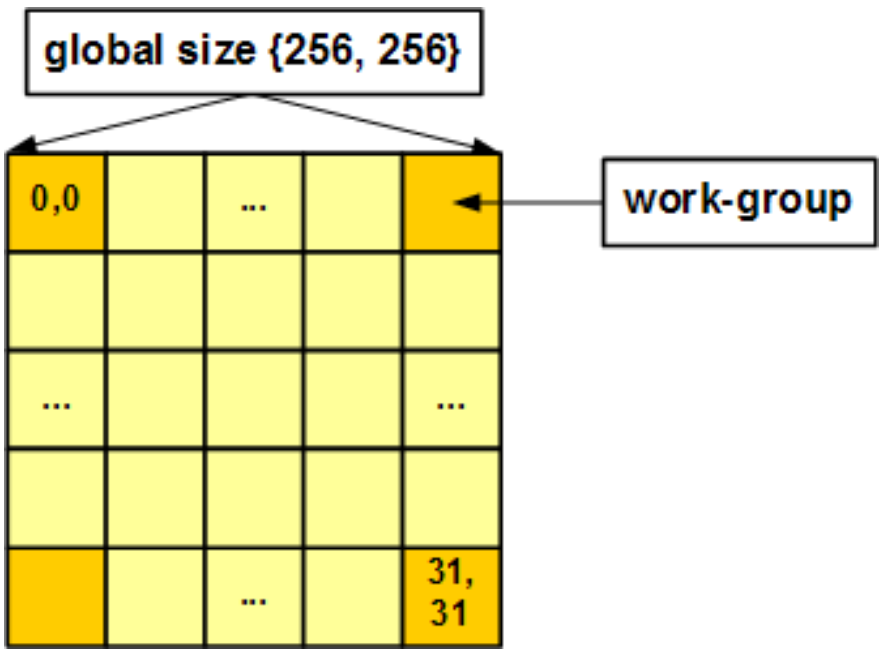
OpenCL NDRange - пример

- обработка изображения 256x256, 1 пиксель = 1 рабочий элемент



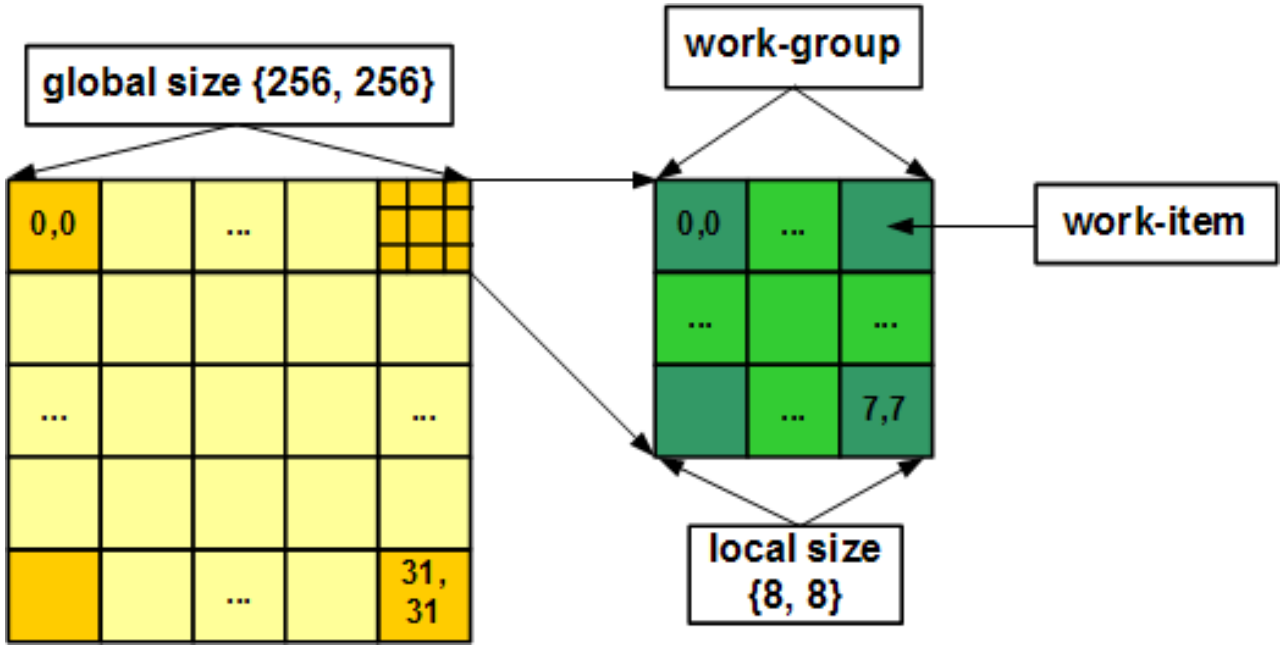
OpenCL NDRange - пример

- обработка изображения 256x256, 1 пиксель = 1 рабочий элемент



OpenCL NDRange - пример

- обработка изображения 256x256, 1 пиксель = 1 рабочий элемент



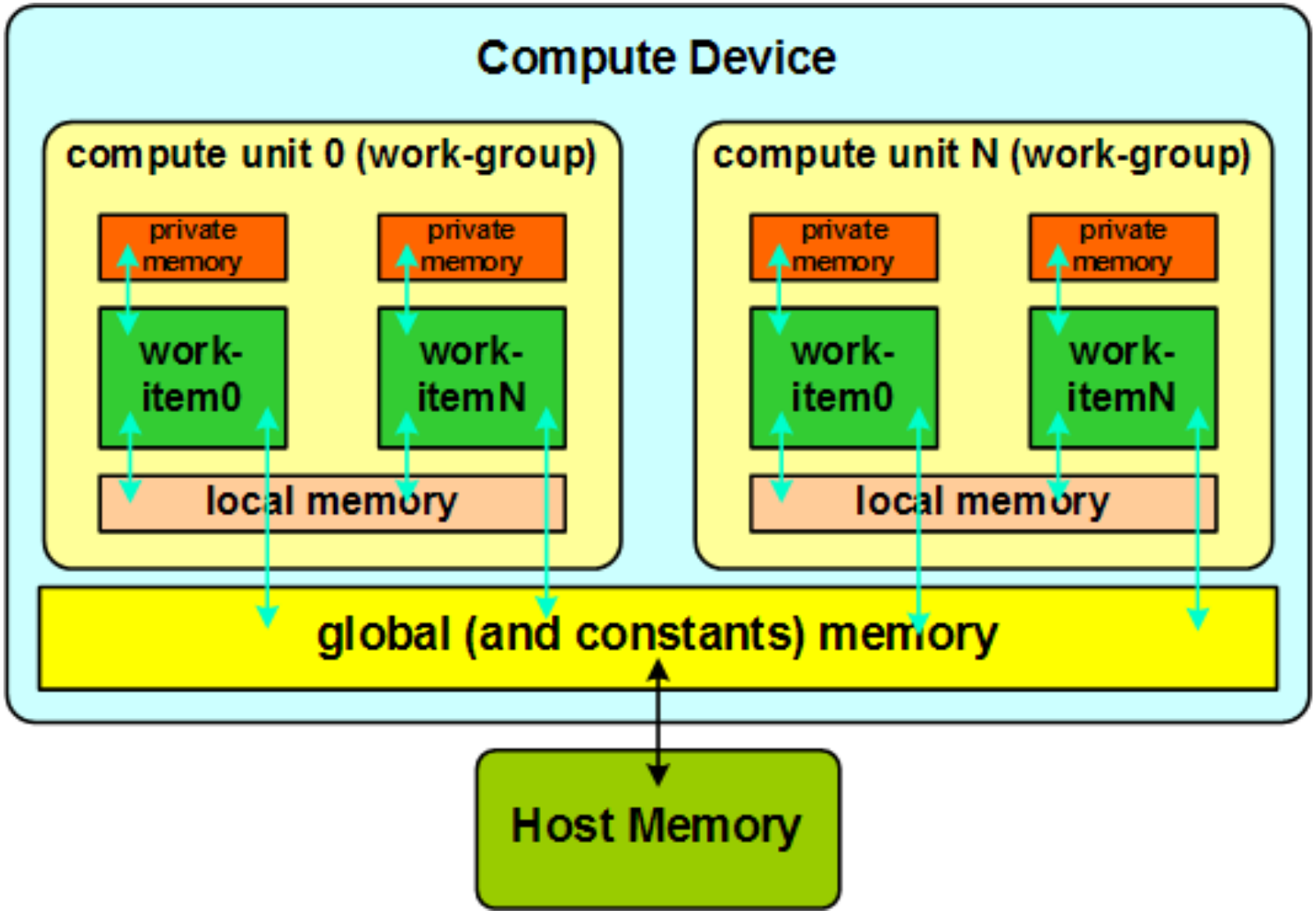
OpenCL NDRange

- рабочий элемент получает уникальный индекс внутри группы - *localID*
- рабочая группа получает уникальный индекс группы - *groupID*
- индексы доступны внутри Kernel'а через функции `get_()`
- размеры пространства индексов:

```
num_groups * local_size = global_size
local_id + group_id * local_size = global_id
global_size % local_size = 0
```

Содержание

OpenCL - модель памяти



Содержание

Пример - версия программы для CPU

простейшая программа, складывающая поэлементно два массива вещественных чисел [3]

- создать (прочитать) 2 массива вещественных чисел
- в цикле по количеству элементов
  - сложить 2 соответствующих элемента
  - результат сложения записать в соответствующий элемент результирующего массива

[3] https://sites.google.com/site/primatgpgpu/fajly-i-ssylki

Содержание

Содержание

Platform, Device

API - библиотека, реализующая общение Host'а и Девайса(ов)

- **Platform** - это реализация стандарта OpenCL для некоторого типа устройства, на котором возможны параллельные вычисления (NVidia, ATI, ...)
  - может быть несколько в одной системе
  - все они будут управляться OpenCL
  - `clGetPlatformIDs()`
  - запросить информацию по платформе можно с помощью `clGetPlatformInfo()`
  - ID платформы по умолчанию = 0
- **Device** - конкретная плата, например, графическая карта
  - может быть несколько на одной платформе
  - список можно получить с помощью `clGetDeviceIDs()`
  - `clGetDeviceInfo()` запрашивает свойства (максимальные размеры рабочих групп, количество доступной памяти,...) девайса

Context

- **Context** - объект управления одним или несколькими девайсами. Остальные OpenCL объекты создаются с помощью Context
  - `clCreateContext()` создает контексты
  - можно смешивать девайсы от различных платформ в одном контексте (правда ни к чему хорошему не приводит)
  - programs, kernels, buffers, events - создаются в контексте
  - buffers, images, events доступны всем девайсам в контексте

Mem objects

- **Memory object** (Buffers, Images) - создаются на всех девайсах, внутри контекста
  - нет malloc'a, глобальная память выделяется только через `clCreateBuffer()`
  - важно назначение памяти `MEM_READ_ONLY`, `MEM_WRITE_ONLY`
  - картинки можно создать через `clCreateImage{2D|3D}()`
  - ну и конечно не забыть очистить память `clReleaseMemObject()`
  - можно инициализировать и получить к ним доступ, используя **host_ptr**, **map/unmap**, **read/write/copy**

Programs

- **Programs** - программы, создаются из текстовых или бинарных источников (файлов, памяти)
  - `clCreateProgramWithSource()`, `clCreateProgramWithBinary()`
  - являются набором функций, kernel'ов
  - компилируются для девайсов `clBuildProgram()`
  - возможна подгрузка компилятора как dll, тогда `clUnloadCompiler()`
  - Kernels создаются из программы с помощью `clCreateKernel()` и `clCreateKernelsInProgram()`
  - построение программы и создание kernel'а может занимать существенное время
  - установка аргументов kernel'а `clSetKernelArg()`

CommandQueue

- **CommandQueue** - очередь команд
  - соответствует одному девайсу
  - каждая команда может ожидать какого-нибудь события и сама создавать событие (Event)
  - `clCreateCommandQueue()`
  - очереди могут выполнятся по порядку и не очень, что зависит от девайса
  - `clSetCommandQueueProperty()` используется для настройки очереди
  - для каждого девайса можно создать несколько очередей
- Пример:
  - записать первый буфер с вещественными числами
  - записать второй буфер с вещественными числами
  - исполнить kernel
  - прочитать буфер с результатом

Events

- **Event** - события для синхронизации очередей команд
  - может использоваться только внутри одного контекста, но в разных очередях
  - `clFlush()` - все предыдущие команды посылает реально выполняться (только посылает!)
  - `clFinish()` - ждет, когда все предыдущие команды реально(!) выполнятся
  - `clWaitForEvents()` ждет события host'а
  - `clEnqueueMarker*()` создает событие "все предыдущие команды в очереди завершены"
  - `clEnqueueWaitForEvents**()` - задержка выполнения очереди до определенного события
  - `clEnqueueBarrier*,**()` - задержка до выполнения всех предыдущих комманд

Running

- помещение комманд в очередь (не значит, что сразу будут выполняться)
  - `clEnqueueReadBuffer()`, `clEnqueueWriteBuffer()`, `clEnqueueCopyBuffer()` чтение, запись, копирование данных в созданные буферы
  - `clEnqueueMapBuffer()/clEnqueueUnmapMemObject()` - связывают кусок памяти хоста с буфером
  - `clEnqueueNDRangeKernel()` запуск ядра

Содержание

OpenCL - основные шаги для примера

простейшая программа, складывающая поэлементно два массива вещественных чисел [4]

- получить доступ к *платформе*
- получить доступ к *девайсу*
- создать Context для девайса
- создать CommandQueue в контексте
- создать буферы под данные: 2 для векторов, один для результата
- создать Kernel
- поместить комманды в очередь = запустить вычисления
- сравнить результат (данные, время выполнения) с полученным на CPU

[4] https://sites.google.com/site/primatgpgpu/fajly-i-ssylki

Содержание

Содержание

C99 с ограничениями

- нет рекурсии
- нет указателей на функции
- нет стандартных хедеров
- нет битовых полей
- нет спецификаторов extern, static, auto, register
- нельзя писать по указателю на данные меньше int

Содержание

Спецификаторы

- Спецификаторы адрессного пространства
  - `__global` - из глобальной памяти
  - `__constant` - тоже из глобальное памяти только read-only
  - `__local` - память, доступная всем членам рабочей группы
  - `__private` - память, доступная только рабочему элементу
  - `__readonly` - только для Image'ей
  - `__write_only` - только для Image'ей
  - аргументы kernel'а должны быть `global`, `local` или `constant`
- Функции
  - `__kernel` - только такие функции являются kernel'ами (т.е. entry point)

Функции, Типы данных

- Индексы потоков
  - `get_work_dim, get_global_size, get_global_id, get_local_size, get_local_id, get_num_groups, get_group_id` - получает индекс в соотвествущей пространстве
- Типы данных
  - `bool, char (cl_char), unsigned char, uchar(cl_uchar), short (cl_short), unsigned short, ushort(cl_ushort), int (cl_int), unsigned int, uint (cl_uint), long (64-bit) (cl_long), unsigned long, ulong (64-bit) (cl_ulong), float (cl_float), half (cl_half), size_t (CL_DEVICE_ADDRESS_BITS), ptrdiff_t (CL_DEVICE_ADDRESS_BITS), intptr_t, uintptr` - получает индекс в соотвествущей пространстве

Содержание

Kernel

C function

```
void vector_add(float *a, float *b, float *c, size_t
n)
{
   size_t i;
   for(i = 0; i < n; i++)
     c[i] = a[i] + b[i];
}
```

OpenCl kernel

```
__kernel void vector_add(__global float *a, __global
float *b, __global float *c)
{
   size_t i;
   i = get_global_id(0);
   c[i] = a[i] + b[i];
}
```

ПРОДОЛЖЕНИЕ СЛЕДУЕТ...