

# Видео-карточки. Теория

Александр Сергеев

## 1 Общие штуки

Платформы(Intel, AMD, NVIDIA) содержат девайсы(видяха 1, видяха 2)

```
1 clGetPlatformIDs(NULL, 0, &sz); //return memory size needed
   in sz
2 clGetPlatformIDs(buffer, buffer_size, NULL); //return
   platform list in buffer
```

Возвращает переменное количество аргументов

```
3 #include <CL/cl.h> //minimal needed header
```

Виды функций в cl:

1. возвращает код ошибки
2. функции *clCreate\**: возвращает объект, код ошибки по указателю

```
4 clGetPlatformInfo(...); //get platform info
5 clGetDeviceIDs(platform, ...);
6 clGetDeviceInfo(device, ...);
```

## 2 Создание контекста

```
7 clCreateContext(...);
8 id = clCreateProgramWithSource(...); //load files in
   context
9 err = clBuildProgram(id, device_list, build_options, ...);
   //compile file $id for devices from $device_list and link
   it to program $id, build_options = "...", not NULL
10 clGetProgramBuildInfo(...);
```

## 3 Код

```
1 kernel void add(global const int *a, global const int *b,  
    global int *c) {  
2     size_t x = get_global_id(0);          //0 -- номер координаты  
3     c[x] = a[x] + b[x];  
4 }
```

*kernel* – точка входа

```
11 clCreateKernel(...);
```

## 4 Память

size\_t на девайсе != size\_t на хосте

```
12 cl_mem buf = clCreateBuffer(...);  
13 clCreateCommandQueue(device, flags); //flags: profiling_info  
    -- enable stats, out_of_order_execution_enable -- do not  
    use it  
14 clSetKernelArg(id, arg_n, buf, buf_size);  
15 clEnqueueWriteBuffer(buf, data, data_size, ...); //flag:  
    blocking_write. If not,  
16 clEnqueueNDRangeKernel(dimensions, &global_work_size, offset=  
    NULL, local_work_size=NULL);          //dimensions = 1,  
    global_work_size = 1, local_work_size=NULL --  
    автоматическое разбиение  
17 clEnqueueReadBuffer(...); //flag: blocking_read  
18 clReleaseMemObject(buf);
```

Можно сделать запись и исполнение неблокирующими, а чтение – блокирующим

Блокирующие операции запускают очередь

Т.к. действия выполняются последовательно, то мы заблокируемся до конца исполнения

CreateBuffer – ленивый, т.е. память создается в момент использования

## 5 Понятия

Work Item – логический исполнитель

SINT – single instruction, multiple threads

Ядра в видеяхи ~ конвейеры в процессоре: умеют считать, но не более

Пачка тредов(Warp) исполняется с единым IP  
Если треды наткнулись на if, то исполняется и if, и else, но результат применяется только в тех тредах, для которых if актуален (остальные треды простаивают)  
Если ни одному треду не надо входить в if, то ок  
В видяхах есть кэш, который используется, чтобы уменьшить передачу данных и сократить энергопотребление (дешевле, чем жирная шина)  
У видеокарт есть новый тип памяти: shared  
Эта память доступна для всего исполнителя  
Ее больше, чем регистров, но меньше, чем оперативки  
Локальная группа – это набор тредов, исполняемых на одном исполнителе  
Локальная группа состоит из нескольких warp'ов. Из них в каждый момент времени исполняется только один  
Смена warp'ов происходит в момент обращения к памяти и прочей тяжелой фигуре  
(хотя на одном исполнителе может быть несколько локальных групп)  
Все треды, относящиеся к одной локальной группе, будут исполняться на одном локальном исполнителе  
Shared память запрашивается в момент запуска: если попросил слишком много, то локальная группа даже не запустится  
Заметим, что в видяхах память оптимизирована на запись, а не на чтение, поэтому оптимальнее иметь больше warp'ов на исполнителе  
Заметим, что регистры 32битные регистры, а значит использование 64битных значений съедает регистры (ценный ресурс так-то)  
В OpenCL регистры – приватная память  
Заметим, что регистры общие на исполнитель. Если мы «съедаем» слишком много, то вместо регистров начинает использоваться память, а она медленная  
Поэтому регистры надо экономить  
Нумерация тредов может быть 1-3 мерной  
*Occupancy* – параметр, равный отношению количества загруженных ядер к количеству ядер вообще  
При низком occupancy и высоком

## 6 Продвинутое произведение матриц

Написав примитивное произведение матриц, мы получим маленькие флорс  
Это нормально

Давайте оптимизировать

В тупом ядре каждый тред будет читать по столбику и строке

Т.е. требуется  $\Omega(n)$  памяти для вычисления  $n$  столбцов

Давайте кэшировать строки и столбцы, чтобы соседние треды использовали этот кэш

Поместим их в локальную память перед использованием

Локальной памяти  $\geq 32KB$  с версии 1.2

Однако этого может не хватить даже на строку

Поэтому давайте кэшировать блоками (TILE). Сначала первые  $m$  элементов строки и столбца, потом следующие  $m$  блоков

1. Разобьем треды на локальные группы  
Удобнее всего делать группу квадратной
2. Задефайним размер в девайс-коде (дефайн удобно, т.к. можно менять при компиляции)
3. Теперь наш внутренний цикл по  $k$  разобьется на два: читаем данные в кэш, потом вычисляем
4. Переносить из глобальной памяти в локальную можно всей локальной группы. Если завести буфер размера  $TILE * TILE$ , то каждый поток перенесет ровно по одному элементу
5. Не забываем ставить барьер после переноса

В OpenCL требуется, чтобы размер глобальной группы был кратен размеру локальной группы

Если размеры матрицы не кратны размеру локальной группы, то можно расширить матрицу

Другой вариант – написать if при переносе из глобальной памяти в локальную

Теперь оптимизируем сами обращения

Вспоминаем, что при запросе к памяти обычно возвращается не байтик, а сразу пачка

Если несколько соседних тредов в варпе посылают запросы на соседние

ячейки, то они объединяются в один запрос  
 В OpenCL треды с соседними  $x$  обычно расположены рядом  
 Для локальной памяти справедливы те же соображения  
 Локальная память состоит из банков памяти  
 Они чередуются где-то через 4 байта  
 Лучше ходить в разные банки  
 Отсюда следует, что лучше читать сплошные куски памяти (т.к. в таком случае распределение будет равномернее)  
 Еще одна причина не итерироваться по  $y$  (шанс попасть в один банк)  
 Однако если несколько тредов делают запрос в одну ячейку, то это все еще один запрос

//todo:

1. zero cost copy
2. host memory
3. const memory

## 7 Векторные типы данных

*float4* – векторный тип данных из 4 флотов  
 Нотации:

1.  $b.x, b.y, b.z, b.w$
2.  $b.r, b.g, b.b, b.a$
3.  $b.s0, b.s1, b.s2, b.s3$

Классический способ обращения по индексам:

```

1  union {
2      float4 vect;
3      float[4] arr;
4  }
```

Есть поддержка операторов:

```

1  float4 + float4; //vector sum
2  float4 + float = float + float4; //add float to each
   coord
```

Не забываем ставить  $f$  в конце констант! Double работают очень медленно, не используй их

```
1 float4.xy == float2; //first 2 coords
2 float4.zxy;
3 float.xxz;
```

Аналогично для остальных типов

Можно вызывать арифметические операции, sin

При вызове  $\langle == \rangle$  на float4 получаем int4, где char.si = 0 при a.si  $\langle == \rangle$  b.si == false и -1 при true

При сравнении double4 получаем long4 и т.д., т.е. sizeof(x) == sizeof(x  $\langle == \rangle$  x)

На видеокарточке эти типы нужны чисто для удобства: выигрыша в производительности нет

На процессоре эти операции действительно могут преобразоваться в векторные инструкции(не факт)

Замечание: по стандарту алайнмент элемента должен быть кратен размеру

Т.е. float\* -> float4\* - ub

Поэтому при перемещении данных из глобальной в локальную память надо использовать специальные методы: vloadn, vstoren

Еще замечание: в opencl есть 3 адресных пространства: private, global, local

Поэтому и указатели бывают 3 типов