Программирование на видеокартах

Начало

Как всё это компилировать

Хост-код

`.h` файлы для C/C++ есть на официальном сайте, правда, там есть задепрекейченные полезные функции. Придётся задефайнить версию. Достаточно хедера `<CL/cl.h>`. Можно — `<CL/opencl.h>`

Чтобы слиноваться, надо где-то взять либы. Под линуксом — некий магический package. Под виндоусом — откуда-то взять. На официальном сайте Кроноса есть. Обратите внимание на битность!

Девайс-код

Само разберётся. Обычно.

API

Большинство функций возвращают код ошибки (как обычно 0 — успешно, не 0 — не успешно). Надо проверять! Во всяком случае, в дебаге, в учебных целях. Написать макрос? Какой ужас.

Это за исключением функций типа `create`, которые возвращают то, что они `create`. Тогда код ошибки, если нужен, по ссылке, передаваемой в аргумент… Такой типичный C.

- `clGetPlatformIDs` — API для получения списка доступных платформ. Принимает… Указатель, размер, и указатель на размер… Буффер, размер буффера, и то, куда записывать, сколько.

  Понятное дело, память она не выделяет. Так как надо, чтобы освобождал тот, кто выделил. Такой типичный C…

  Так, а какого размера выделять буффер? Для этого есть специальный вариант вызова: `(nullptr, 0, &x)` — тогда в `x` нам запишут то, сколько на самом деле вариантов. Такой типичный C…

  Правда, в отличие от типичного C, если буфера мало, то это не ошибка “буфера мало”, а нам запишут, сколько есть места.

  Можно сказать, “возвращает” эта функция список id платформ, где можно запустить opencl.

TODO!
show rule 'типичный C'

TODO!
пояснения к коду

TODO!
getDeviceIDs

Делаем что-то полезное

Допустим, мы выбрали девайс, который нам нравится, с которым мы хотим работать.

Теперь нам нужно создать контекст.

- `clCreateContext`

Контекст — это своего рода `globalThis` от мира OpenCL. Он инкапсулирует в себе всё, что мы хотим из хост-кода делать с девайс-кодом.

Теперь нужно в этом контексте создать тот код, который мы будем запускать в этом контексте:

- `clCreateProgramWithSource`

  “Принимает” массив указателей на source.

  TODO!
  Наебаться на сто дурных

  Обычно мы хотим не извращаться с C-style строчками, а иметь отдельный файл с исходным кодом (обычно расширение `.cl`). То есть, надо открыть файл, прочитать и скормить (компиляция девайс-кода будет уже в рантайме). Предлагается использовать обычные C’шные функции fread и прочее. Открыть в `binary`, `seek` до конца и так далее. Типичный C…

  Эта функция не делает ничего особенного. Если мы тут зафейлились, мы где-то конкретно налажали — передали кривой буфер или ещё что-нибудь. Ошибки внутри девайс-кода будут обнаруживаться уже потом.

- `clBuildProgram`

  Вот это уже серьёзно: компиляция нашего исходника. Передаём сюда `id` нашего `program`’а, который мы по’create’или, и девайс, под который надо скомпилиться. Или передать null, и тогда будет скомпилированно под все девайсы, привязанные к контексту.

  Компиляция может занимать продолжительное время! Имеет смысл локально закэшировать. Но так как результат сильно зависит от драйверов, девайсов, версий, погоды, фазы луны; распространять прекомпилированную версию не имеет смысла.

  Если в девайс-коде есть какая-нибудь синтаксическая (или ещё что-нибудь) ошибка, то она вылезет здесь. Обязательно проверять результат здесь! Можно сделать `clGetBuildInfo`, чтобы получить билд-лог (своего рода compilation error), ему нужно давать честный `id` девайса.

  build-options — не должен быть null! Некоторые платформы могут его не пережить. Передайте пустую строчку. Или, лучше, используйте по назначению! Например, можно с помощью `-D` передавать дефайнами переменными.

Наконец, девайс-код

```
kernel void add(global const int *a, global const int *b, global int *c) {
  *c = *a + *b;
}
```

Здесь немного нового по сравнению с C.

- `kernel` означает, что это точка входа в программу. Их может быть несколько — это не совсем то же, что `main`.
- `clCreateKernel` — создаёт идентификатор, через который мы сможем вызывать `kernel`. Передаём туда имя.

Теперь мы хотим передать ему аргументы и запустить его! Проблемы? Э-э-э…… Указатели на память, что вы думаете, сработает? Ха. Где память? На девайсе. Мы её выделили? Нет. Очень жаль. Давайте выделять…

Here we go again

- Выделяем память на девайсе: `clCreateBuffer`. Нам надо только передать размер буфера, и то, как мы хотим к нему обращаться из кернела (read only, write only, read-write). В данном случае, мы хотим три буфера: под `a` и под `b` — read-only, под `c` — write-only. Ну, мы можем оба подписать read-write, но лучше так — так что-нибудь оптимизировать (не надо про протокол когерентности, кэши, всё остальное думать).

  Понятно, что доступ распространяется только на кернел, с хоста-то мы и так записать/прочитать сможем.

  `clCreateBuffer` возвращает `cl_mem`, это своего рода указатель, но не указатель. Это хендл, арифметику с ним делать нельзя.
- А мы написали `int`… а что это? Понятное дело, что `int` на девайсе и `int` на хосте могут отличаться. Ну так для этого может быть `cl_int`. На самом деле, они даже фиксированы, не зависят от девайса. Так что это всегда 32 бита. Лучше, чем в C! А вот `size_t` девайса мы не знаем. Точнее, можем спросить, но это будет уже в рантайме и у конкретного девайса. Так что кернелам просто запрещено принимать `size_t`.
- Наконец, можем накормить кернел аргументами: `clSetKernelArg`, указываем туда идентификатор кернела, номер аргумента, значение аргумента (оно — как адрес + количество байтов).
- В наших буферах лежит какой-то мусор, надо его наполнить. `clEnqueueWriteBuffer`. Чувствуется в названии подвох.
- `clCreateCommandQueue` — очередь команд, чего мы хотим. Принимает контекст и девайс. А ещё принимает *флажки*. Один из них полезный — profiling info (или как-то так), причём почти ничего не стоит. Рассказывает, сколько времени уходит на процессы. Второй — `out of order executionary order`. Не влезай, убьёт.
- `clEnqueueWriteBuffer`, да. Принимает cl_mem, указатель наш (откуда брать данные), флажок `blocking write` (позже). Про блокирующее чтение. Мы ставим *задание на постановку в очередь*. Если мы не поставим флажок, то оно вернётся мгновенно! И ничего не дождётся. Имеет смысл делать передачу данных **на** девайс не блокирующей, а **с** девайса — блокирующей. Очередь ленивая, не будет ничего исполнять, пока мы не пнём её. Ну, или можно пнуть через “подождать выполнения всех функций”. Или спросить “~~а не в омах ли измеряется сопротивление~~ а не закончилось ли исполнение”.

- Спойлер: В конце захотим `clEnqueueReadBuffer`.
- `clEnqueueNDRangeKernel` — ураааа! Запуск кернела. Передаём туда, очевидно, очередь и кернел. Принимает также dimensions (who? пока передаём 1); global work size, причём через указатель (пока тоже 1); local work size — смело кормим null'ами, так же поступаем с event'ами.
- Наконец, делаем `clEnqueueReadBuffer`, и читаем нашу замечательную сумму двух чисел. Будем на это, во всяком случае, надеяться.
- Замечание к окончанию: всё, что мы `create`, хорошо бы потом `release`. А то может быть грустно.
- А если что-нибудь криво работает на девайсе… Ну, упадёт видео-драйвер. Винда его обычно поднимает через пару секунд, остальные — … поэкспериментируйте!