ВЫЧИСЛЕНИЯ НА GPU

OpenCL и обёртки вокруг него. Основные концепции GPU compute.

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

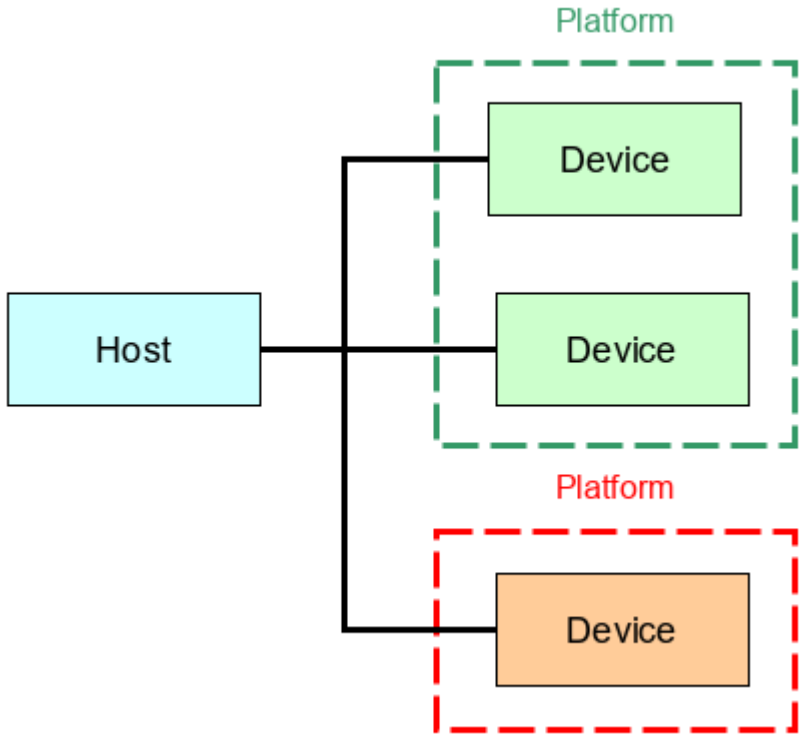
- Логическая модель OpenCL
- Память и синхронизация
- OpenCL C++ API
- Битоническая сортировка

Гетерогенные системы

- Когда мы говорим о вычислениях, мы часто имеем в виду CPU.
- Но в жизни существуют также:
  - Видеокарты.
  - Графические ускорители.
  - Карты для машинного обучения.
  - И многое другое.
- Существует ряд API для работы с ними. В основном всех их поддерживает Khronos.
- OpenGL, WebGL – графика.
- OpenVG – векторная графика
- Vulkan – графика и вычисления.
- OpenCL – вычисления.
- OpenVX – компьютерное зрение и обработка изображений.
- OpenXR – дополненная и виртуальная реальность.

Модель вычислений OpenCL

- В модели OpenCL разделены **host** и **device**.
- Host это та машина, на которой выполянется программа-драйвер.
- Device (устройство) это та машина, на которой проводятся инициированные драйвером вычисления.
- Ничего не мешает им физически быть одним и тем же, скажем, микропроцессором.



OpenCL 2.1

Запросить платформы и устройства

Полезные API для того, чтобы получить информацию о платформах и устройствах, доступных на вашей системе

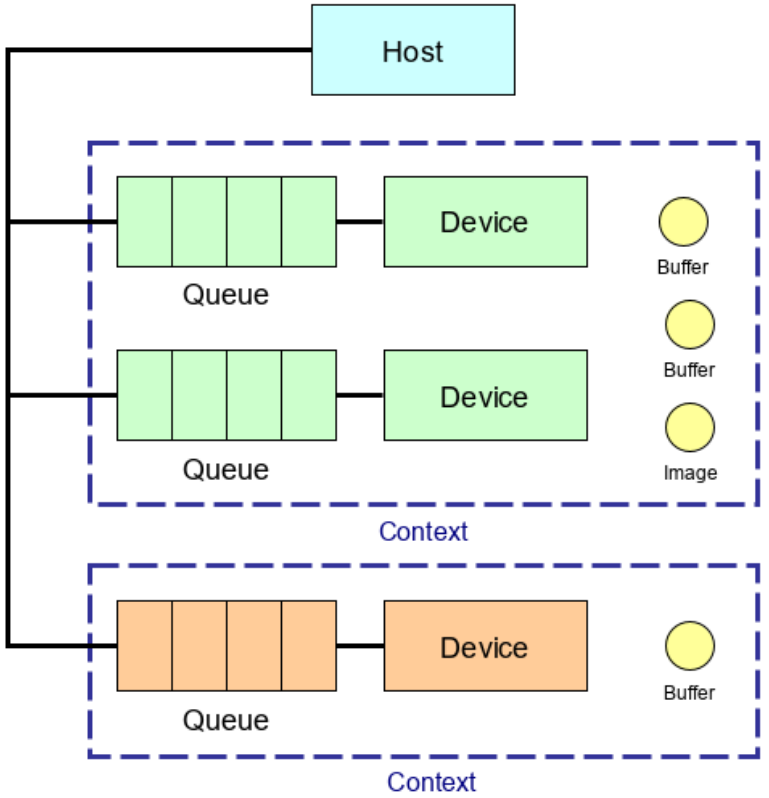
- **clGetPlatformIDs** чтобы получить список платформ
- **clGetPlatformInfo** чтобы получить информацию о платформе
- **clGetDeviceIDs** чтобы получить список устройств на данной платформе
- **clGetDeviceInfo** чтобы получить информацию об устройстве

Чтобы посылать запросы к устройствам, инициировать вычисления и получать результаты, для них надо создавать **контексты**

cl_platform.c

Модель владения OpenCL

- Ключевое понятие это контекст, который создаётся для коммуникации с конкретным устройством
- Контекст владеет очередями запросов, а также дополнительными объектами: buffer, image, pipe, sampler, etc…
- Несколько контекстов могут владеть одним устройством с разными очередями запросов



Подсчёт ссылок

- Многие объекты (памяти и проч.) выделяются на стороне хоста.

```
cl_mem buf = clCreateBuffer( .... ); // счётчик ссылок = 1
```

- При этом этот объект является ref-counted. Рантайм сам его уничтожает когда количество ссылок становится равным нулю.

```
clRetainMemObject(buf); // увеличили счётчик ссылок до 2
clReleaseMemObject(buf); // уменьшаем счётчик ссылок до 1
clReleaseMemObject(buf); // уменьшаем счётчик ссылок до 0
```

- В этот момент буфер совобождён и попытка его использовать это UB.

Создать контекст, очередь, буфер

- Существуют API для того, чтобы создать контекст, очередь и буферы в нём
  - **`clCreateContext`** чтобы создать контекст
  - **`clCreateCommandQueue`** чтобы создать очередь (до OpenCL 2.0)
  - И так далее, они приблизительно однотипные
- Все такого рода вещи тоже создаются refcounted.
  - Например парным к **`clRealeaseContext`** является **`clRetainContext`**
- Чтобы записать или прочитать буфер на устройстве мы должны поставить запись в очередь (как в вулкане но без CmdBuffer посередине)
  - **`clEnqueueWriteBuffer / clEnqueueReadBuffer`**

cl_justbuf.c

Обсуждение

- Также как в Vulkan многовато кода, который просто должен быть (boilerplate).
- Какое у нас должно быть первое побуждение, когда мы видим **такое** C API?
- Правильно: написать C++ wrapper.
- Далее мы разберём как может выглядеть и что делать такой враппер на примере стандартного `opencl.hpp`

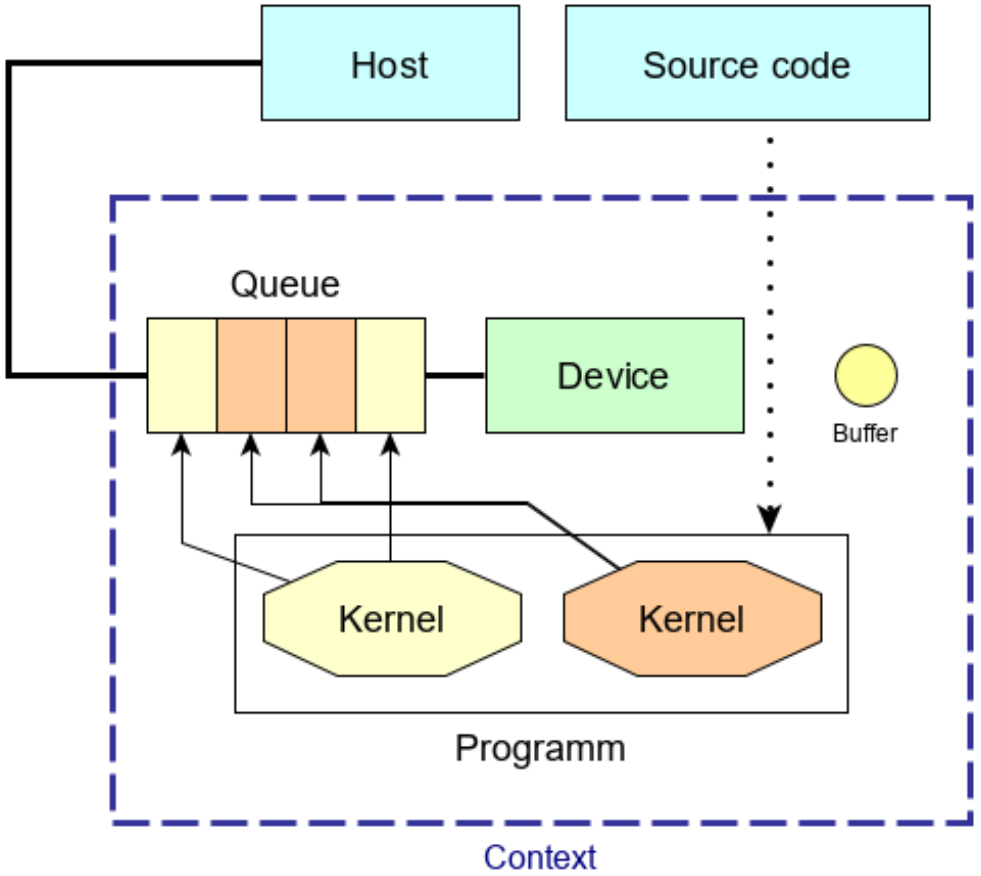
Пересылка буфера на OpenCL C++

```
// Буферы A и B на хосте
cl::vector<int> A(BUFSZ), B(BUFSZ);

cl::Context Context{CL_DEVICE_TYPE_GPU, properties};
cl::CommandQueue Queue{Context};

// Буфер D на устройстве
cl::Buffer D{Context, CL_MEM_READ_WRITE, BUFSZ * sizeof(int)};

// Пересылка A → D
cl::copy(Queue, A.begin(), A.end(), D);

// Пересылка D → B
cl::copy(Queue, D, B.begin(), B.end());
```


cl_justbuf.cc

Модель вычислений OpenCL

- Пересылать данные хорошо, но хотелось бы что-то считать.
- Устройства исполняют ядра (kernels), которые на них отсылаются, попадая в их очередь
- Исходный код совокупности ядер называется программой (program) и компилируется на устройстве
- И вот те данные над которыми ядра работают уже пересылаются.



Итерационное пространство задачи

- Посмотрим на kernel для сложения векторов

```
__kernel void
vector_add(__global int *A, __global int *B, __global int *C) {
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

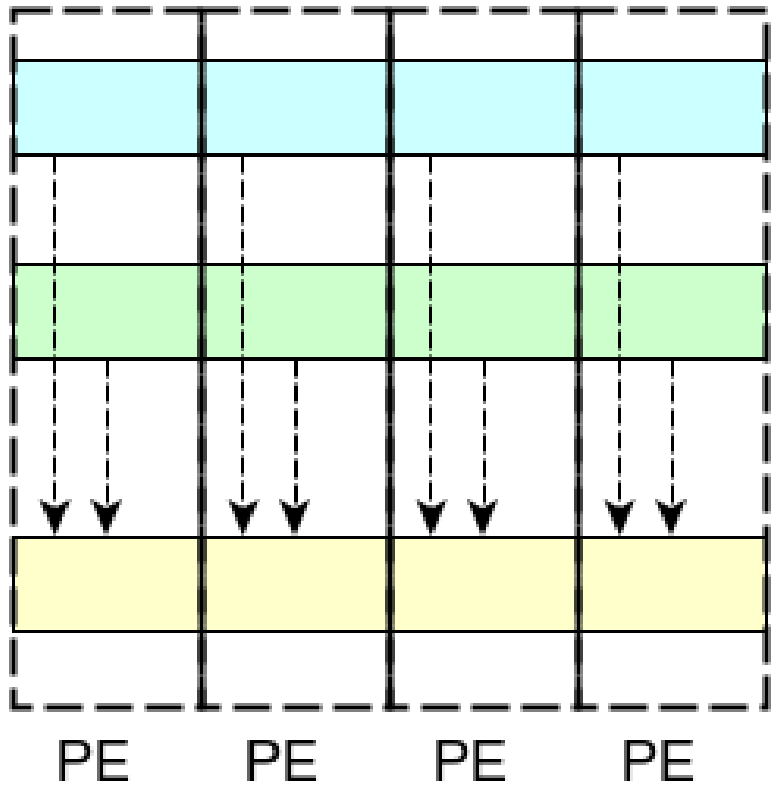
- Мы пока плохо понимаем что такое `__global`
- Кроме того, такое чувство, что тут написано сложение одного элемента векторов
- Это связано с тем, что устройство OpenCL это **SIMT**

Одна инструкция – много потоков

- Аббревиатура SIMT похожа на известную многим SIMD, но разница в нюансах.
- Single instruction – multiple data имеется в виду, что **одна инструкция** работает с большим количеством данных.
- Single instruction – multiple threads имеется в виду, что **с одной инструкцией** работает большое количество потоков.
- Модель SIMT характерна для сверхпараллельных устройств (GPU, APU) и реализует идею throughput-first вычисления.
- В обычных CPU победила latency-first модель. Процесс посылки задачи на обработку стороннему вычислителю называется **offload**.

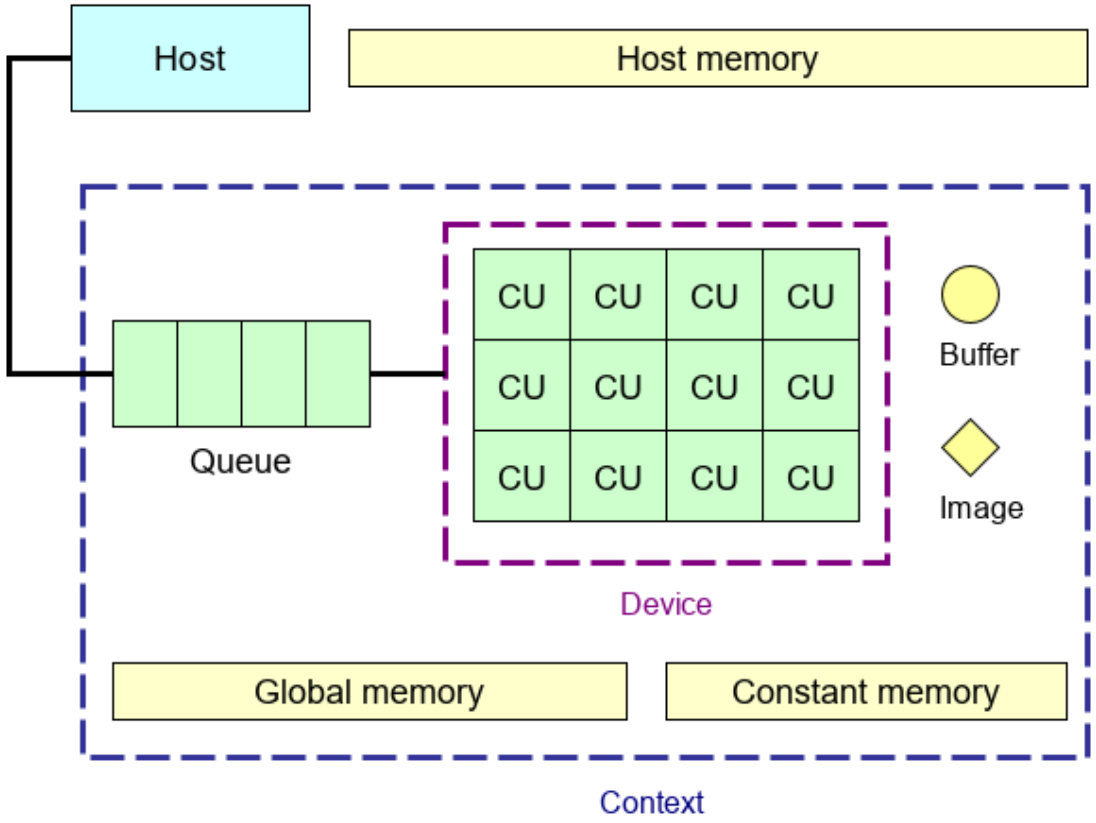
Итерационное пространство задачи

- В парадигме SIMT, одно ядро исполняется на многих элементах итерационного пространства (range) в параллель.
- В примере vector add мы имели дело с одномерным пространством, но оно может быть и двумерным и трёхмерным.
- Из-за этого самая лучшая задача для OpenCL это **та, которая лучше всего параллелится**.



Идея глобальной памяти

- Хостовая память это привычная программисту RAM
- Устройства состоят из computing units (CU, вычислительные модули)
- Вычислительные модули поддерживают фиксированное количество потоков исполнения
- Все вычислительные модули внутри устройства имеют общий доступ к его глобальной памяти



Где мы в глобальной памяти

- По сути указатель в глобальную память это uniform переменная.
- А вот global id это уже varying.

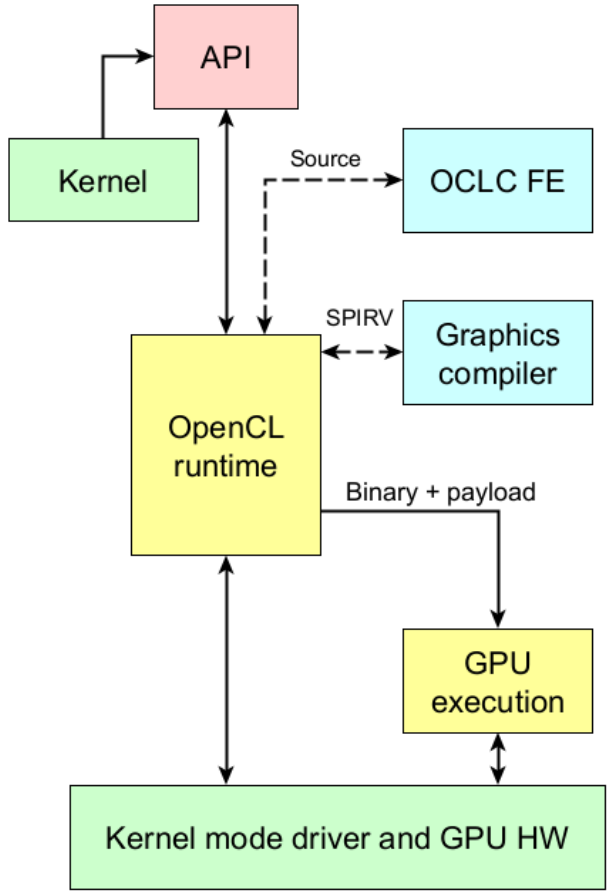
```
__kernel void
vector_add(__global int *A, __global int *B, __global int *C) {
  int i = get_global_id(0); // varying i
  C[i] = A[i] + B[i];
}
```

- Kernel это точка входа вроде функции main.
- Буфер в глобальной памяти устанавливается на хосте как **аргумент** кернела.

clSetKernelArg.html

C++ API: kernel functor

```
cl::Context Context{CL_DEVICE_TYPE_GPU, prop};
cl::CommandQueue Queue{Context};

// true == build immediately
cl::Program program{Context, vakernel, true};

cl::NDRange GlobalRange{Sz};
cl::EnqueueArgs Args{Queue, GlobalRange};

cl::KernelFunctor<cl::Buffer, cl::Buffer,
cl::Buffer> add_vecs{program, "vector_add"};

// enque, execute, wait
cl::Event evt = add_vecs(Args, A, B, C);
```



cl_vectoradd.cc

Информация о выполнении

- Результатом выполнения функтора является Event.

```
cl::Event evt = add_vecs(Args, A, B, C); evt.wait();
```

- Его можно использовать чтобы подождать результат, а можно для профилировочной информации.

```
time_start = evt.getProfilingInfo<CL_PROFILING_COMMAND_START>();
time_end = evt.getProfilingInfo<CL_PROFILING_COMMAND_END>();
```

- Это позволяет чётко понимать сколько мы провели собственно на GPU, выполняя задачи (из всего потраченного времени).

Взаимозависимость кернелов

- В структуре EnqueueArgs мы можем сконфигурировать систему Events.

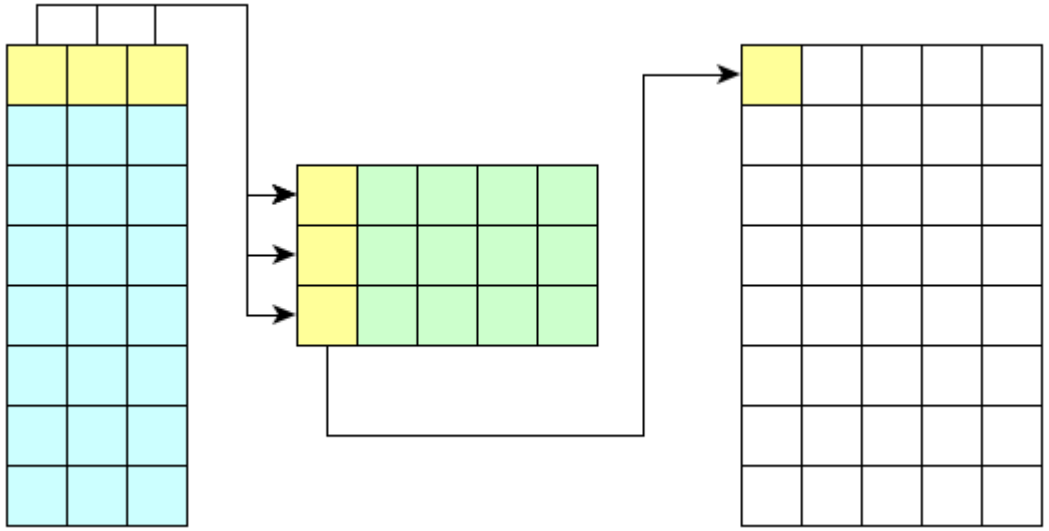
```
cl::EnqueueArgs Args{Queue, GlobalRange};
cl::Event Evt = add_vecs(Args, A, B, C); // C = A + B
cl::EnqueueArgs DepArgs{Queue, Evt, GlobalRange};
cl::Event Evt = add_vecs(DepArgs, A, C, B); // B = A + C
```

- Здесь мы сказали запускать второе ядро только после выполнения первого
- Одна из сложностей OpenCL: мы всегда должны думать в терминах асинхронных очередей.

- Логическая модель OpenCL
  - Память и синхронизация
- OpenCL C++ API
- Битоническая сортировка

Обсуждение: а что для матриц?

- Вам нужно умножить матрицу (N x M) на матрицу (M x K)
- Что будет элементом итерационного пространства этой задачи?



Case study: перемножение матриц

- Простейшее ядро для перемножения

```
__kernel void simple_multiply(__global int *A,
  __global int *B, __global int *C, int AX, int AY, int BY) {
  int row = get_global_id(0);
  int col = get_global_id(1);
  int sum = 0;

  for (int k = 0; k < AY; k++)
    sum += A[row * AY + k] * B[k * BY + col];
  C[row * BY + col] = sum;
}
```

- Уже даёт ощутимый выигрыш над аналогичной программой для CPU.

gemm_simple.cl

Обсуждение

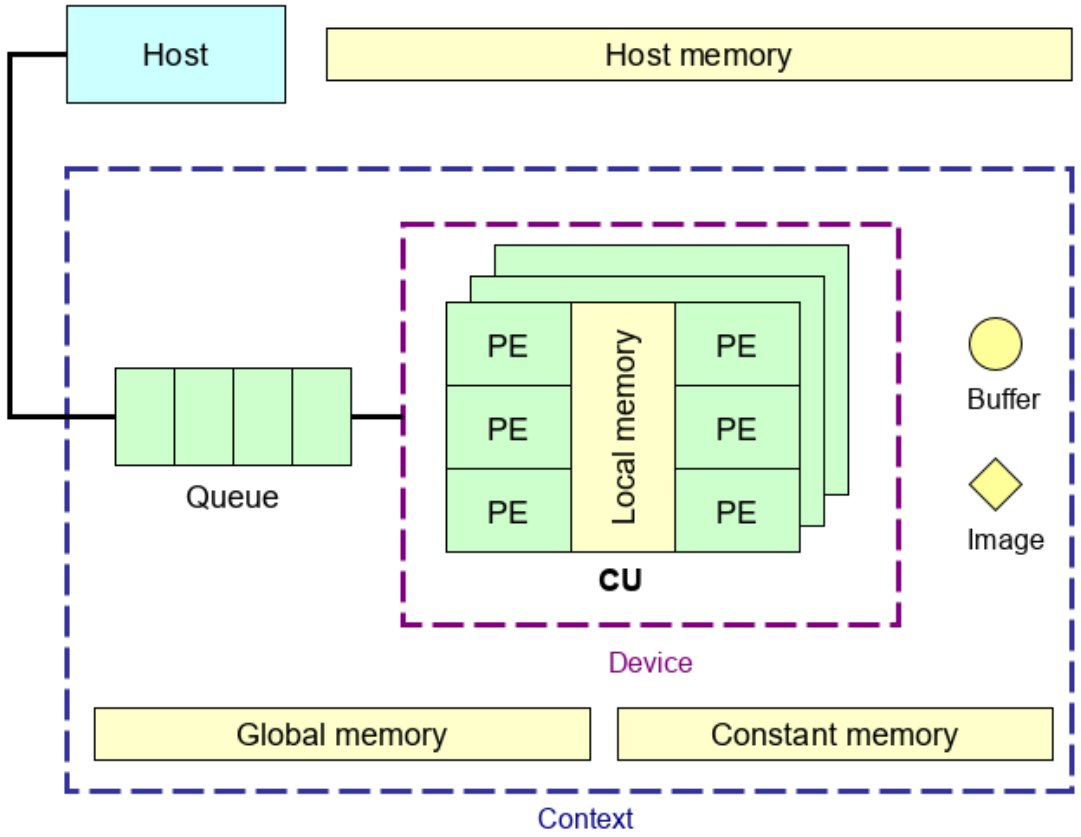
- Работа с памятью это всегда проблемы производительности

```
for (int k = 0; k < AY; k++)
   sum += A[row * AY + k] * B[k * BY + col];
```

- Можно ли здесь как-то всё улучшить?
- Подумайте как бы вы подошли к задаче если бы вы писали программу для обычного CPU?

Локальная память в OpenCL

- Каждое вычислительное устройство делится на Processing Elements (в терминологии CUDA threads).
- Все потоки внутри вычислительного устройства имеют общий доступ к локальной памяти.
- Думайте о локальной памяти, как о кэше.



Локальная память: управление с хоста

- Для OpenCL C используется модификатор local

```
__kernel void histogram(__global uchar *data, int nelts,
  __global int *histogram, __local int *local_hist, int bins) {
....
  int lid = get_local_id(0); // varying внутри local space
  int gid = get_global_id(0); // varying внутри global space
```

- В хостовом C++API есть специальный псевдо-буффер, обозначающий "тут локальная память" (это буфер с памятью null pointer, ненулевого размера).

```
cl::KernelFunctor hist<cl::Buffer, cl_int, cl::Buffer,
  cl::LocalSpaceArg, cl_int>(program, "histogram");
```

Локальная память внутри ядра

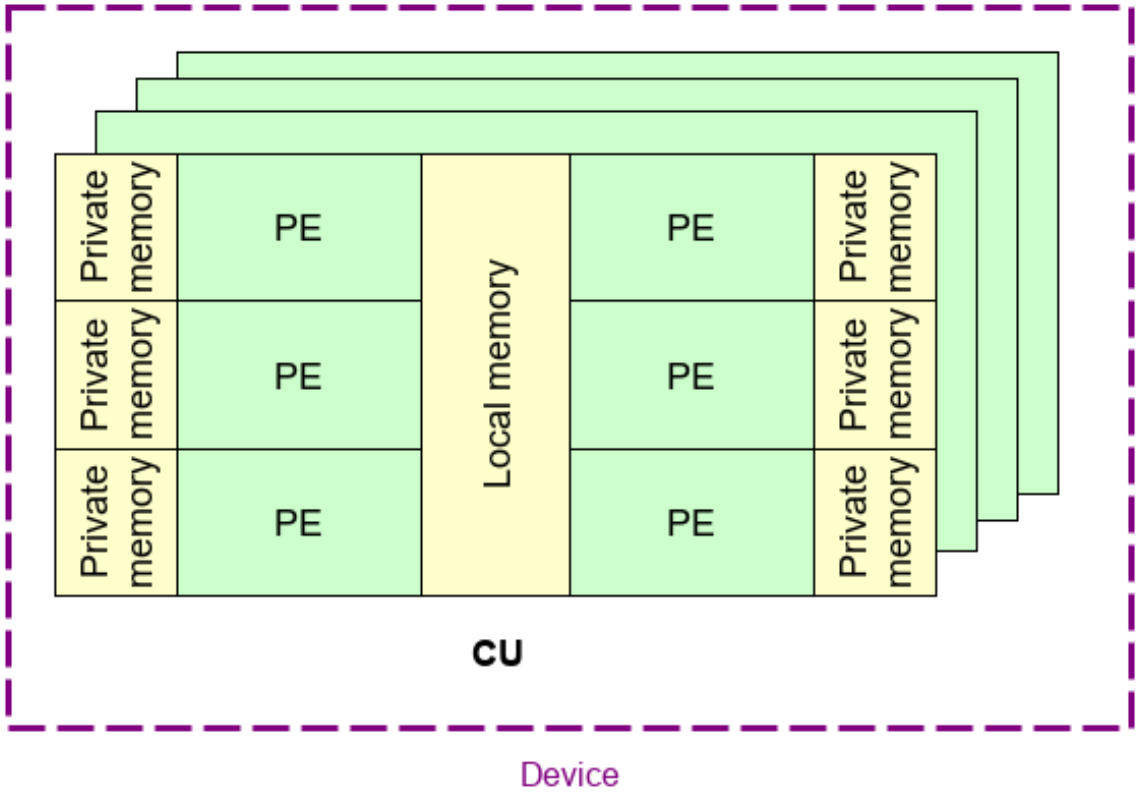
```
__kernel void matrix_multiply(__global float *A,
  __global float *B, __global TYPE *C, int AX, int AY, int BY) {

const int row = get_local_id(0); // Local row ID (max: TS)
const int col = get_local_id(1); // Local col ID (max: TS)
...

__local TYPE Asub[TS][TS]; // local memory buffer
__local TYPE Bsub[TS][TS];
```

- С хоста в любом случае нужно передать размеры локальной памяти в описании аргументов

```
cl::EnqueueArgs Args(Queue, GlobalRange, LocalRange);
```

Приватная память в OpenCL

- Каждый поток обладает приватной памятью (пока её мало, думайте о ней как о регистрах).
- Соотношение скорости локальной и приватной памяти это сложный вопрос.
- Работа с локальной и приватной памятью это высший пилотаж OpenCL.



Summary: память

- Хостовая память.
- Разновидности памяти на устройстве
  - Private memory (просто переменная внутри ядра).
  - Global memory (обозначается __global).
  - Constant memory (обозначается __constant).
  - Local memory (обозначается __local).
- Shared virtual memory (SVM).
  - Хостовая память, видимая с устройства. Нужна для динамических структур данных (деревья, списки).

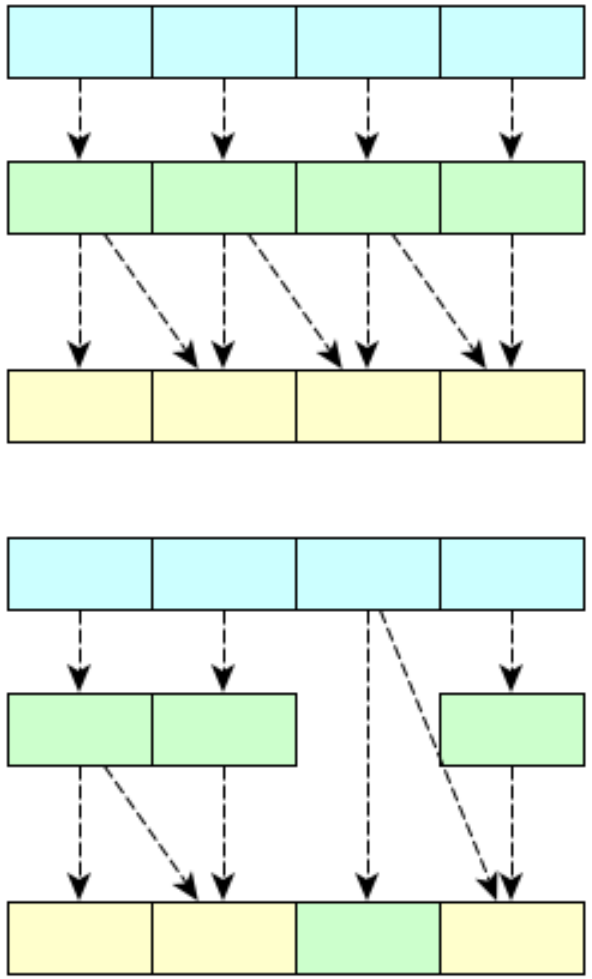
Case study: улучшаем матрицы

```
// переменные в приватной памяти (на регистрах)
int tiledRow = TS * t + row, tiledCol = TS * t + col;

// временные буферы в локальной памяти
Asub[col][row] = A[globalRow * AY + tiledCol];
Bsub[col][row] = B[tiledRow * BY + globalCol];

// в цикле используем локальную память и приватную память
for (k = 0; k < TS; k++)
  acc += Asub[k][row] * Bsub[col][k];
```

- Увы, как написано на слайде это не будет работать.
- Дело в том, что потоки бегут с немного разной скоростью.

Идея барьеров

- На картинке сверху показано желаемое поведение
- Снизу вариант реального поведения из-за рассинхронизации потоков в пределах CU
- Для того чтобы все потоки внутри CU ждали друг друга используется барьер (т. е. разновидность семафора)
- Барьеры могут быть и по глобальной и по локальной памяти



Case study: улучшаем матрицы

```
const int tiledRow = TS * t + row, tiledCol = TS * t + col;

Asub[col][row] = A[globalRow * AY + tiledCol];
Bsub[col][row] = B[tiledRow * BY + globalCol];

// Синхронизируем чтобы гарантировать что всё загружено
barrier(CLK_LOCAL_MEM_FENCE);

for (k = 0; k < TS; k++)
  acc += Asub[k][row] * Bsub[col][k];

// Синхронизируем прежде чем начать следующую загрузку
barrier(CLK_LOCAL_MEM_FENCE);
```


gemm_localmem.cl

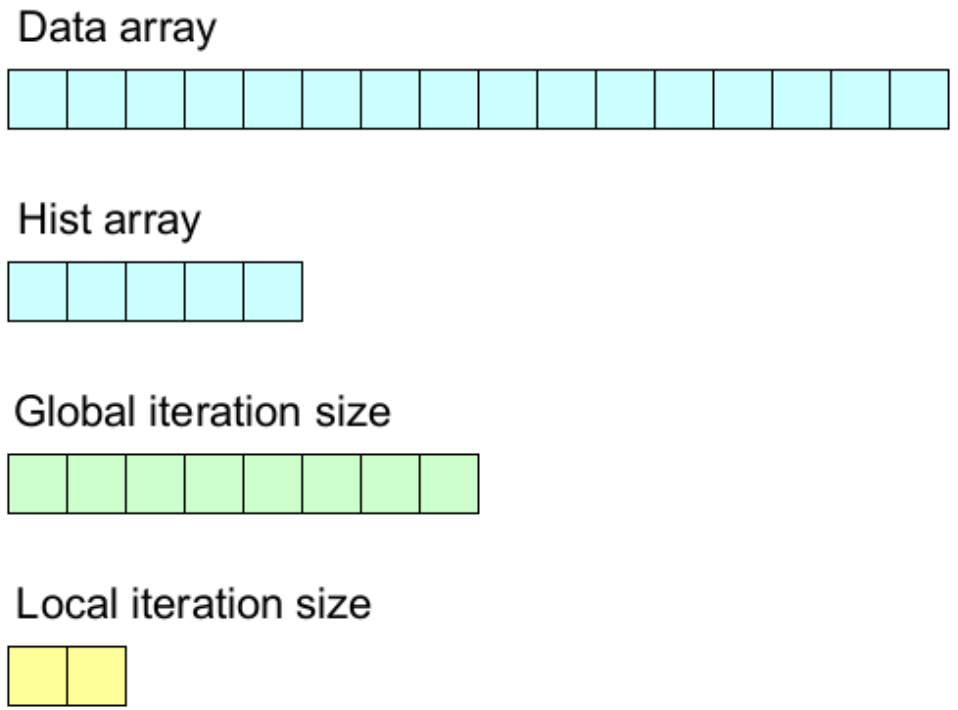
Демонстрация

- Пока что мы говорили о преимуществах, но не показывали их.
- Кажется настало время.
- Дополнительные темы:
  - Отладка INVALID_MEMORY_OBJECT
  - `ocloc.exe compile -device TGLLP -file gemm_simple_modif.cl`
  - `ocloc.exe disasm -file gemm_simple_modif_Gen12LPlp.bin`
  - Изучение ассемблера.
- Дополнительные темы в C++
  - chrono, random, charconv, system_error

cl_gemm.cc

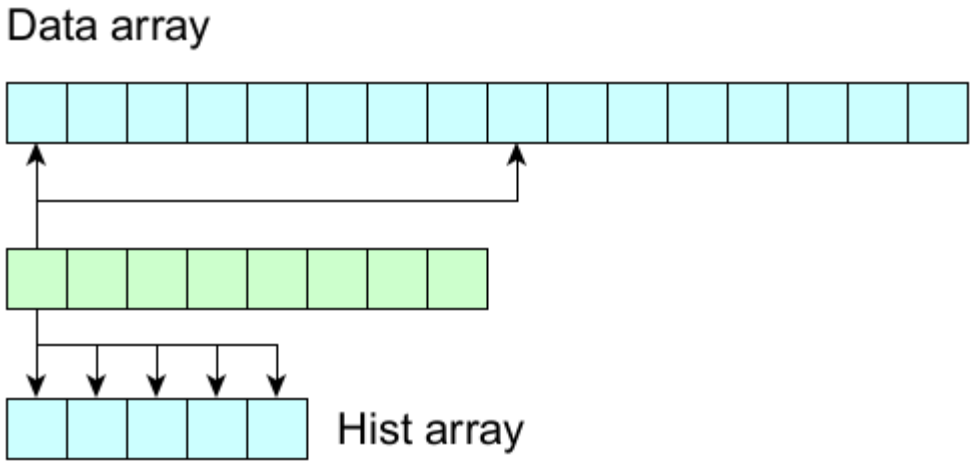
Гистограмма

- Идея гистограммы это сложить в histarray[n] количество элементов из dataarray со значением n.
- Главная проблема в том, что массив данных может быть куда больше чем количество доступных нам потоков даже на GPU.
- И нам как-то надо его поделить, ну и локальную память использовать разумно.



Идея: делим ответственность

```
int gid = get_global_id(0), gsize = get_global_size(0);

for (i = gid; i < num_data; i += gsize)
  histogram[data[i]] += 1; // всё ли видят проблему?
```



hist-onlyg.cl

Сложение должно быть атомарным

```
int gid = get_global_id(0), gsize = get_global_size(0);

for (i = gid; i < num_data; i += gsize)
  atomic_add(&histogram[data[i]], 1);
```

- Теперь всё будет работать, но медленно
- У кого есть идеи, как это ускорить, подключив локальную память?

hist-onlyg-corr.cl

Идея: подключаем локальную память

```
int lid = get_local_id(0), gid = get_global_id(0),
    lsize = get_local_size(0), gsize = get_global_size(0);

for (i = lid; i < num_bins; i += lsize)
  local_hist[i] = 0; // зануляем только то, с чем работаем

barrier(CLK_LOCAL_MEM_FENCE);

for (i = gid; i < num_data; i += gsize)
  atomic_add(&local_hist[data[i]], 1); // локальная часть

barrier(CLK_LOCAL_MEM_FENCE);

for (i = lid; i < num_bins; i += lsize)
  atomic_add(&histogram[i], local_hist[i]); // собираем
```


hist.cl

Демонстрация и обсуждение

- Надо всегда понимать:
  - Сколько у вас глобальное итерационное пространство?
  - Сколько у вас глобальной памяти (обычно больше)
  - То же самое про локальную память и локальное итерационное пространство
- Синхронизация и атомарность внутри OpenCL C могут показаться сложными вещами, но в реальности они там проще, чем на CPU.
- Разумеется мы могли бы разбить гистограмму на подзадачи на CPU и синхронизировать евентами. Было бы это лучше? Попробуйте!

cl_hist.cc

Обсуждение

- Чтобы упрощать наши программы мы использовали C++ API
- Но пока что мы не заглядывали в то, как именно оно устроено.
- А там есть несколько очень важных уроков.

- Логическая модель OpenCL
- Память и синхронизация
- OpenCL C++ API
- Битоническая сортировка

Первый шаг: reference handler

- Почти любая сущность в OpenCL является ref-counted и имеет два специальных метода: retain и release
- `clRetainMemObject(cl_mem), clReleaseMemObject(cl_mem)`
- `clRetainDevice(cl_device_id), clReleaseDevice(cl_device_id)`
- `clRetainContext(cl_context), clReleaseContext(cl_context)`
- И так далее
- Разумно сделать некий класс, абстрагирующий это. Но увы, нет никакой системности в этих функциях

clReleaseMemObject

Идея специализации

- Шаблон класса может быть специализирован, то есть его частный случай для конкретного типа может быть указан непосредственно

```
template <typename T> struct S {
  void dump() { std::cout << "for all\n"; }
};

template <> struct S<int> {
 void dump() { std::cout << "for int\n"; }
};

S<int> s1; s1.dump();    // используется ваша специализация
S<double> s2; s2.dump(); // специализацию делает компилятор
```

Собираем reference handler

- Общий случай:

```
template<typename T> struct ReferenceHandler { };
```

- Конкретные случаи:

```
template <> struct ReferenceHandler<cl_mem> {
  static cl_int retain(cl_mem memory)
    { return ::clRetainMemObject(memory); }
  static cl_int release(cl_mem memory)
    { return ::clReleaseMemObject(memory); }
};
```

- Теперь `ReferenceHandler<X>::release()` это либо release X либо ошибка

CL/opencl.hpp

Второй шаг: wrapper

- Враппер хранит в себе объект и вызывает release на уничтожении

```
template <typename cl_type> class Wrapper {
protected:
  cl_type obj_;

public:
  ~Wrapper() {  release(); }

  cl_int release() const {
    if (!obj_) return CL_SUCCESS;
    return ReferenceHandler<cl_type>::release(obj_);
  }
```

- Как бы вы написали копирование и присваивание?

Второй шаг: wrapper

- Также обратим внимание на перегруженные круглые скобки:

```
const cl_type& operator ()() const { return object_; }
cl_type& operator ()() { return object_; }
cl_type get() const { return object_; }
```

- У этого решения есть очевидная проблема:

```
Wrapper<T> a, b;
a() = b();
```

- Будет работать без release и retain. Чудовищное нарушение инкапсуляции.
- Но класс Wrapper находится в namespace detail, т.е. по конвенции не предназначен для использования извне

Обсуждение

- Эта идея завернуть всё лишнее в namespace detail пока что встречается в мире часто
- С распространением модулей она уйдёт в прошлое, так как модули позволяют определять классы не экспортируя их из модуля
- Кстати, как вы думаете, а делать ли врапперу виртуальный деструктор?

Третий шаг: девайс

- Теперь конкретный класс для девайса может быть унаследован от враппера

```
class Device : public detail::Wrapper<cl_device_id> {
```

- Возможно некоторое переиспользование копирования и присваивания

```
Device(const Device& dev) : detail::Wrapper<cl_type>(dev) {}

Device& operator = (const Device &dev)  {
  detail::Wrapper<cl_type>::operator=(dev);
  return *this;
}
```

- Увы, определять их приходится из-за контроля типа в rhs.

Третий шаг: девайс

- Однако у нас есть небольшая засада

```
vector<Device> devices;

vector<cl_device_id> ids(n);
::clGetDeviceIDs(platform, type, n, ids.data(), NULL);

devices.resize(n);

for (size_type i = 0, e = ids.size(); i < e; ++i)
  devices[i] = Device(ids[i]);
```

- Упс… кто видит тут возможную проблему?

Третий шаг: девайс

- Однако у нас есть небольшая засада

```
vector<Device> devices;

vector<cl_device_id> ids(n);
::clGetDeviceIDs(platform, type, n, ids.data(), NULL);

devices.resize(n);

for (size_type i = 0, e = ids.size(); i < e; ++i)
  devices[i] = Device(ids[i]); // copy и сразу release
```

- Таким образом мы теряем девайсы из-за сбоя в счётчике ссылок
- Что делать?

Выход: специальный конструктор

- Мы можем предусмотреть специальный retain-ctor

```
explicit Device(const cl_device_id &device,
                bool retainObject = false) :
        detail::Wrapper<cl_type>(device, retainObject) {}
```

- И если надо создать временный объект, создавать его с retain

```
for (size_type i = 0, e = ids.size(); i < e; ++i)
  devices[i] = Device(ids[i], true);
```

- Теперь всё хорошо, счётчик ссылок сходится

Этюд: получение информации

- У нас есть возможность запросить информацию о девайсе

```
char buf[STRING_BUFSIZE];
::clGetPlatformInfo(pid, CL_PLATFORM_NAME,
                    sizeof(buf), buf, NULL);

cl_uint ubuf;
::clGetDeviceInfo(devid, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
                  sizeof(ubuf), &ubuf, NULL);
```

- Мы бы хотели:

```
std::string pname = p.getInfo<CL_PLATFORM_NAME>();
unsigned md = d.getInfo<CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS>();
```

Этюд: получение информации

- Первый шаг очевиден: завести эту функцию

```
class Device : public detail::Wrapper<cl_device_id> {
// ....
  template <cl_device_info name>
  ???
  getInfo(cl_int* err = NULL) const {
    // делегация к detail::getInfo
  }
```

- Но как разобраться какой у неё должен быть возвращаемый тип?

Идея: type traits

- Заводим специальную структуру, чем-то похожую на ReferenceHandler

```
template <typename T, cl_int Name> struct param_traits {};
```

- Теперь можно как и раньше расписать специализации

```
template<> struct param_traits<cl_platform_info,
                               CL_PLATFORM_NAME> {
  enum { value = CL_PLATFORM_NAME };
  using type = std::string;
};
```

- Обратите внимание: теперь ключевую роль играют не вложенные функции, а вложенный тип

Этюд: получение информации

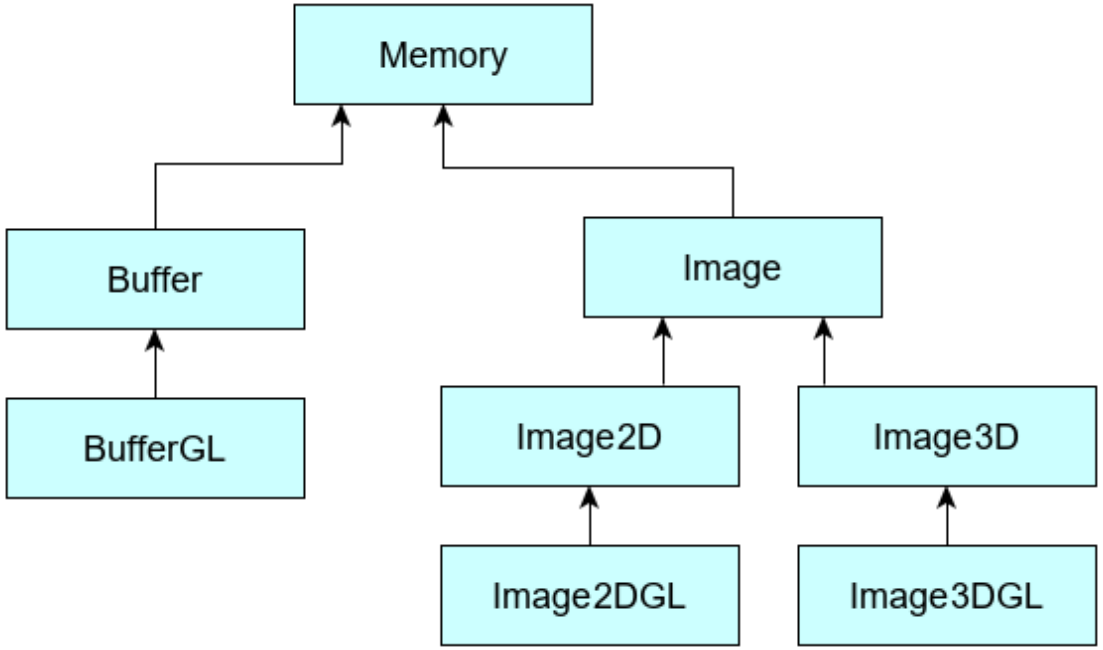
- Теперь используем здесь наши type traits

```
class Device : public detail::Wrapper<cl_device_id> {

// ....

  template <cl_device_info name>
  typename param_traits<cl_device_info, name>::type
  getInfo(cl_int* err = NULL) const {
    // делегация к detail::getInfo
  }
```

- Вот именно за это многие любят C++, а многие -- нет

Иерархия классов

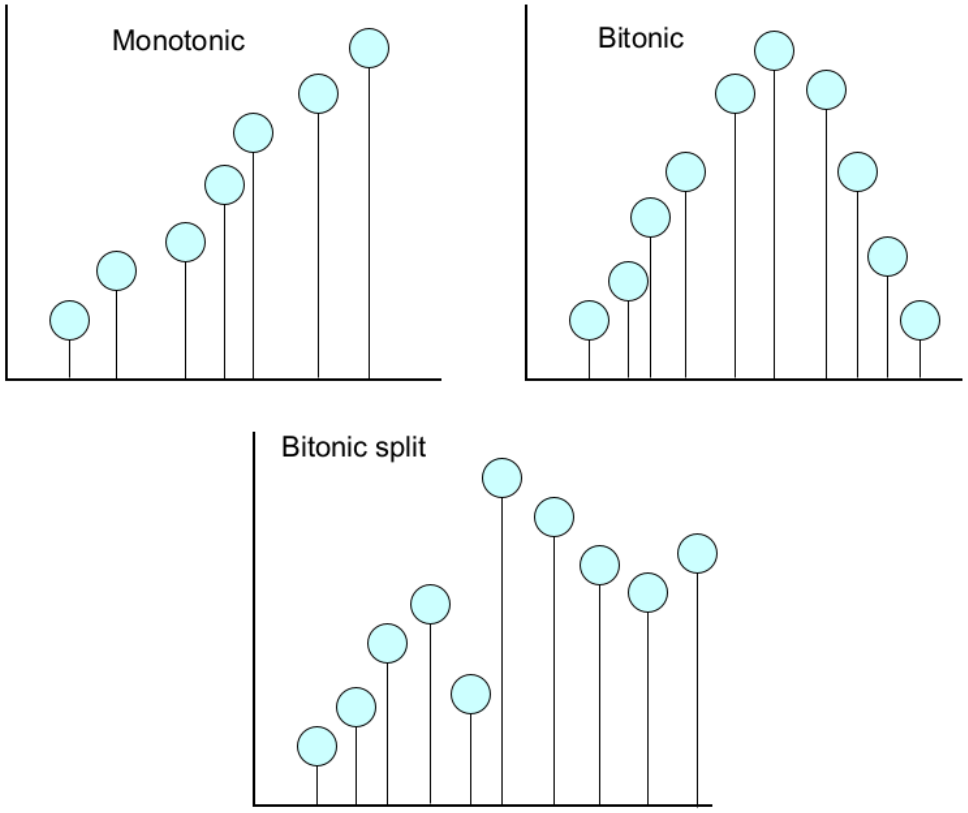
- Полученная иерархия классов отражает предметную область естественным образом
- Например на рисунке показана иерархия памяти
- Для C API у нас конечно такого не было, там мы просто использовали cl_mem небезопасным образом
- Обратите внимание на типы для OGL interop



- Логическая модель OpenCL
- Память и синхронизация
- OpenCL C++ API
- Битоническая сортировка

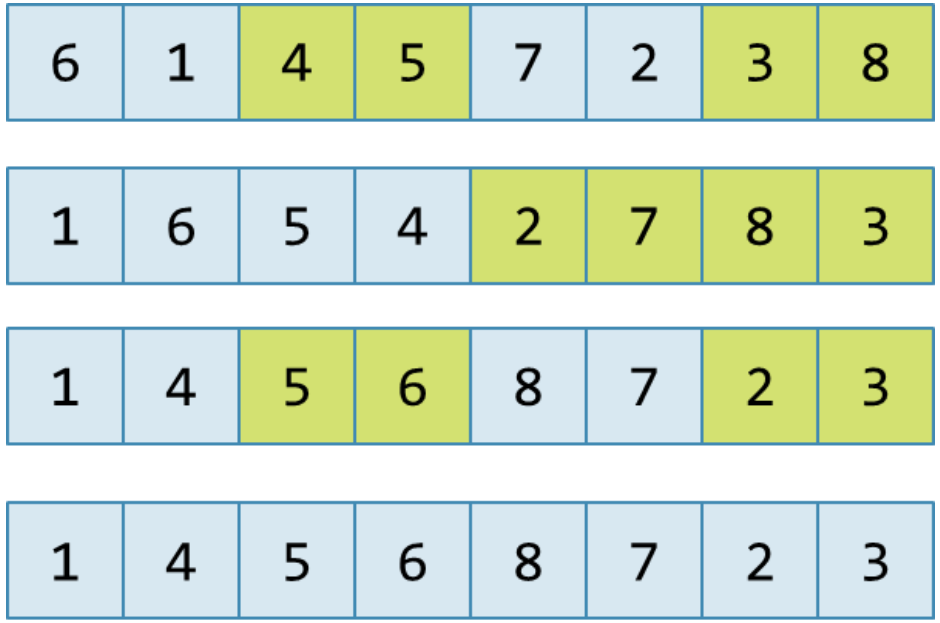
Монотонные последовательности

- Неубывающая (невозрастающая) последовательность монотонная.
- Битонная последовательность состоит из двух разных монотонных.
- Bitonic split: для каждого i от 0 до $N/2 - 1$ сравним его с $i + N/2$ и если он больше, обменяем их местами
- В итоге получится последовательность, каждая половина которой bitonic и правая строго больше левой



Bitonic merge

- Каждые два элемента уже bitonic.
- Отсортируем подпоследовательности из двух элементов, делая левую возрастающей а правую убывающей.
- Далее по четыре по восемь и т.п.
- Проведём bitonic splits до уровня двух элементов, после чего последовательность отсортирована.



HWB: битоническая сортировка

- Ваша задача: отсортировать входную последовательность на видеокарте, используя bitonic sort
- Получится ли у вас серьёзно обогнать CPU?
- Входные данные: число элементов N, далее N несортированных элементов
- Выходные данные: сортированная последовательность

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [$OCL2$] The OpenCL Specification v2.2, 2019
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [TM] Timothy Mattson – OpenCL Programming guide, 2011
- [MS] Matthew Scarpino – OpenCL in Action, 2011
- [RB] Ravishekhar Banger – OpenCL Programming by Example, 2013
- [DK] David Kaeli – Heterogeneous Computing with OpenCL 2.0, 2015
- [JK] John Kessenich – OpenGL Programming Guide, Version 4.5, 2016