

ВЫЧИСЛЕНИЯ НА GPU

OpenCL и обёртки вокруг него. Основные концепции GPU compute.

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com

➤ Логическая модель OpenCL

- ❑ Память и синхронизация

- ❑ OpenCL C++ API

- ❑ Битоническая сортировка

Гетерогенные системы

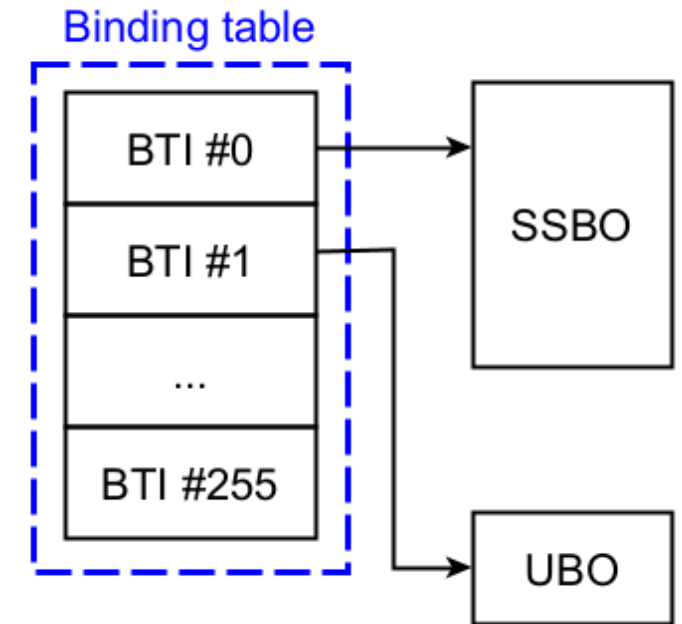
- Когда мы говорим о вычислениях, мы часто имеем в виду CPU.
- Но в жизни существуют также:
 - Видеокарты.
 - Графические ускорители.
 - Карты для машинного обучения.
 - И многое другое.
- Существует ряд API для работы с ними. В основном всех их поддерживает Khronos.
- OpenGL, WebGL – графика.
- OpenVG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

Обсуждение

- До некоторой степени Vulkan можно использовать для вычислений, там есть даже специальные compute queues.
- Зачем нам специализированный compute?

Stateful память в GLSL: binding table

```
struct Particle { vec4 pos, vel; };  
  
layout(std140, binding = 0) buffer Pos {  
    Particle particles[];  
};  
  
layout (binding = 1) uniform UBO {  
    float deltaT;  
    int count;  
} ubo;  
  
for (int i = 0; i < ubo.count; i += DataSz)  
    if (i + x < ubo.count)  
        someData[x] = particles[i + x].pos;
```



Binding table и структура указателя

- "Указатель" particles, таким образом, – это индекс в BTI и смещение в буфере.

```
for (int i = 0; i < ubo.count; i += DataSz)
    if (i + x < ubo.count)
        someData[x] = particles[i + x].pos;
```

- То, что выглядит как доступ к массиву, является доступом к памяти через два уровня.
- Такая память называется **stateful** (пример state – это базовый адрес в памяти).
- В таких языках, как GLSL, просто нет концепции **stateless** (настоящего) указателя.

Настоящего?

- Слово "настоящий" в отношении указателя может показаться казуистикой.
- Но давайте посмотрим на вот такой простой пример.

```
int *uboPtr = &ubo.deltaT; // так сделать нельзя
```

```
int *ssboPtr = &particles[i + x].pos; // и так тоже
```

```
int *somePtr = CUnknownCondition() ? ssboPtr : uboPtr;
```

```
*somePtr = 8; // store в какой буфер? По какому индексу?
```

- По определению BTI задаётся именем буфера. Stateful pointer – это индекс.

```
int * addrspace(binding(1)) ssboPtr = offset(ubo.deltaT);
```

Typed и untyped память

- Концепция stateful памяти есть и в compute языках при работе со сложно структурированными изображениями.

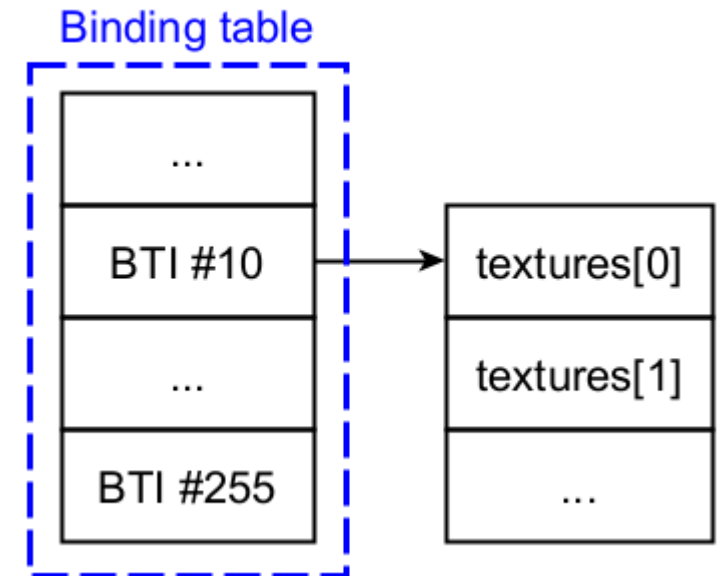
```
accessor<float4, 2, sycl_read, sycl_image> Img(Src, Cgh);  
sampler Sampler(unnormalized, clamp, nearest);  
....
```

```
Value = Img.read(Coords, Sampler);
```

- Здесь state – это уже не только базовый адрес, но и детали Image: формат, длина и ширина, питч. По определению typed память является stateful.
- Напротив, обычный буфер – это **untyped**, и он может быть и stateful, и stateless.

Дивергенция 3D: bindless

- Есть некая проблема в классическом BTI: что делать если у нас 2^{20} дескрипторов?
- Bindless память позволяет привязывать к одной точке целый массив текстур.
- Увы, такая память до сих пор не доступна в compute.
- Но есть расширение GLSL (и HLSL).



```
#extension GL_EXT_nonuniform_qualifier : enable
layout (set = 1, binding = 10) uniform sampler2D textures[];
vec4 color = texture(textures[albedo_id], final_uv.xy)
```

Конвергенция 3D: device pointers

- Нечто вроде настоящих указателей начинает появляться на GPU.

```
// forward declaration
layout(buffer_reference) buffer blockType;

layout(buffer_reference, std430, buffer_reference_align = 16)
buffer blockType { int x; blockType next; };

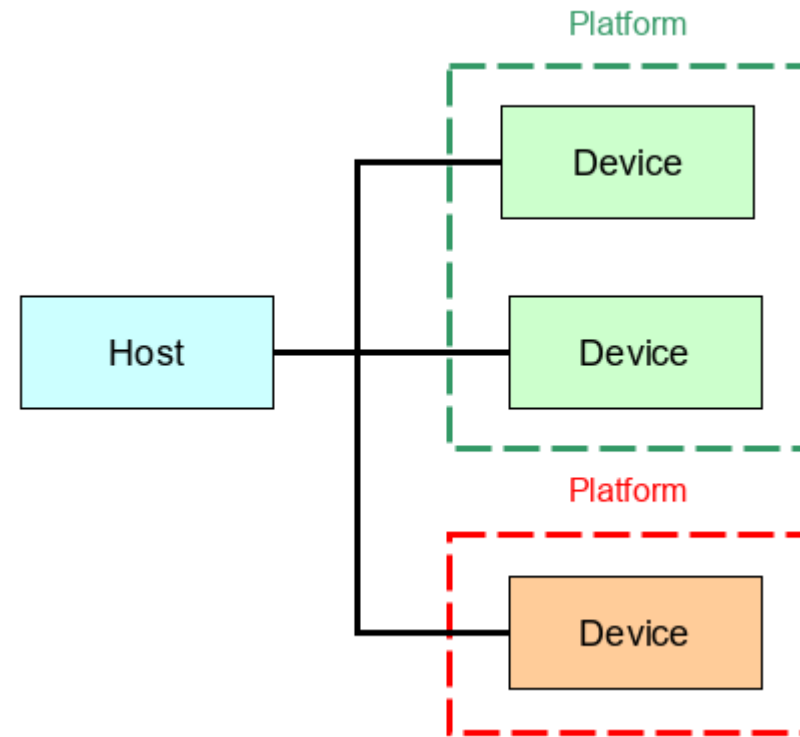
layout(std430) buffer rootBlock { blockType root; } r;

void main() {
    blockType b = r.root;

    // "pointer chasing" through a linked list
    b = b.next.next.next.next.next;
}
```

Модель вычислений OpenCL

- В модели OpenCL разделены **host** и **device**.
- Host это та машина, на которой выполняется 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.



Запросить платформы и устройства

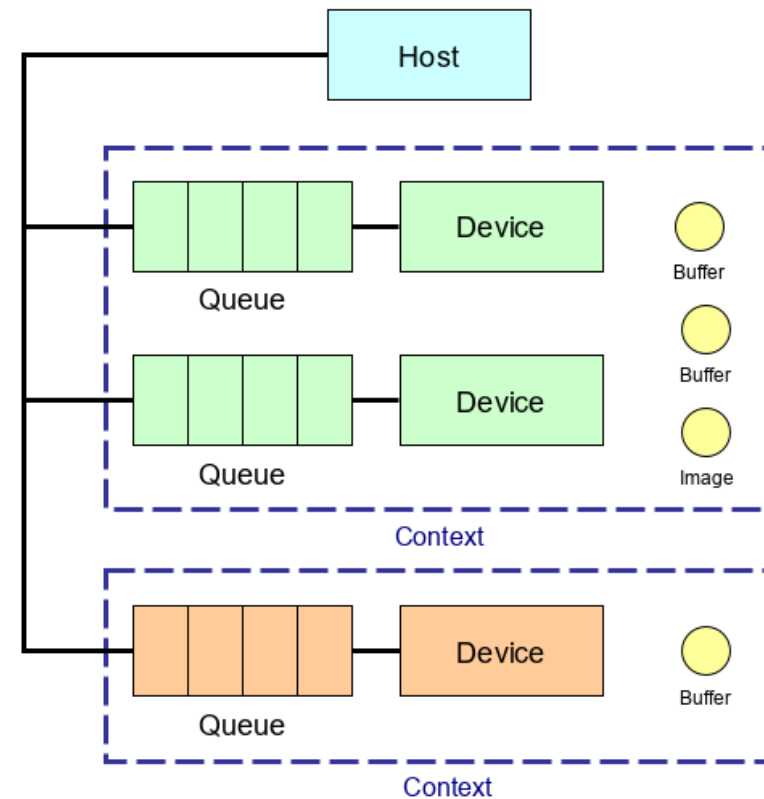
Полезные API для того, чтобы получить информацию о платформах и устройствах, доступных на вашей системе

- **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**

Модель владения OpenGL

- 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 **clReleaseContext** является **clRetainContext**
- Чтобы записать или прочитать буфер на устройстве мы должны поставить запись в очередь (как в вулкане но без CmdBuffer посередине)
 - **clEnqueueWriteBuffer** / **clEnqueueReadBuffer**

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 `opengl.h`

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};

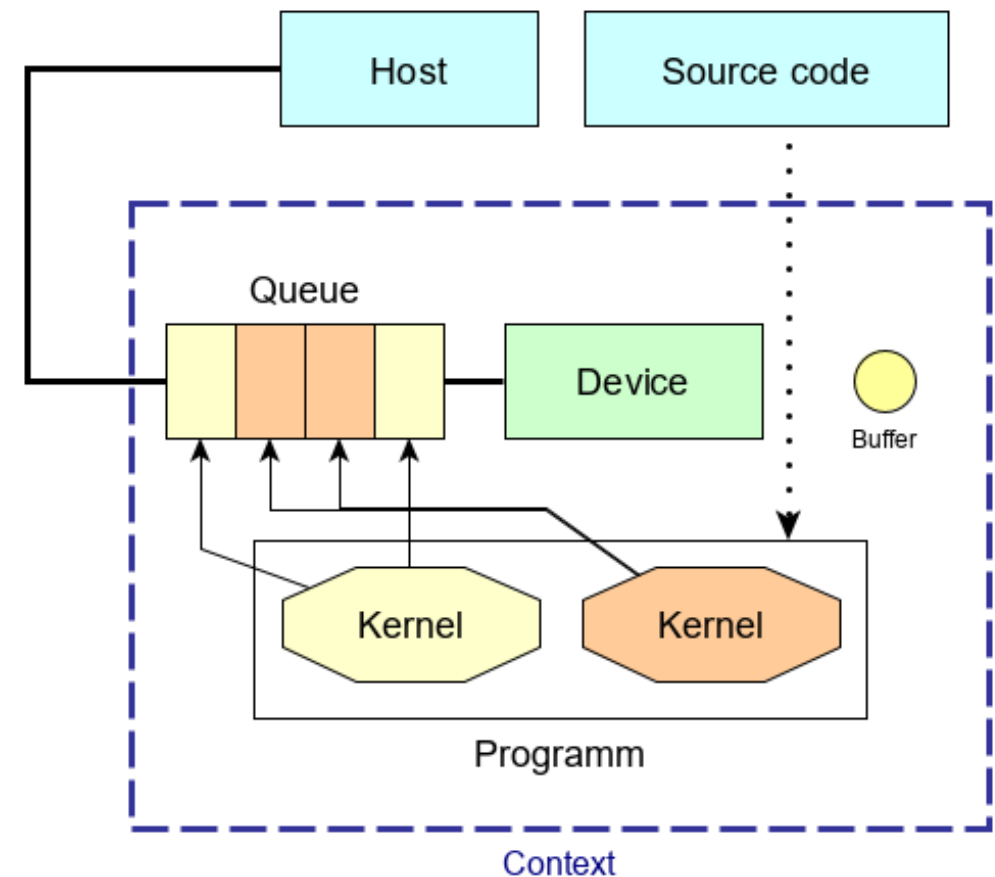
// Буфер D на устройстве
cl::Buffer D{Context, CL_MEM_READ_WRITE, BUFSZ * sizeof(int)};

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

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

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];  
}
```

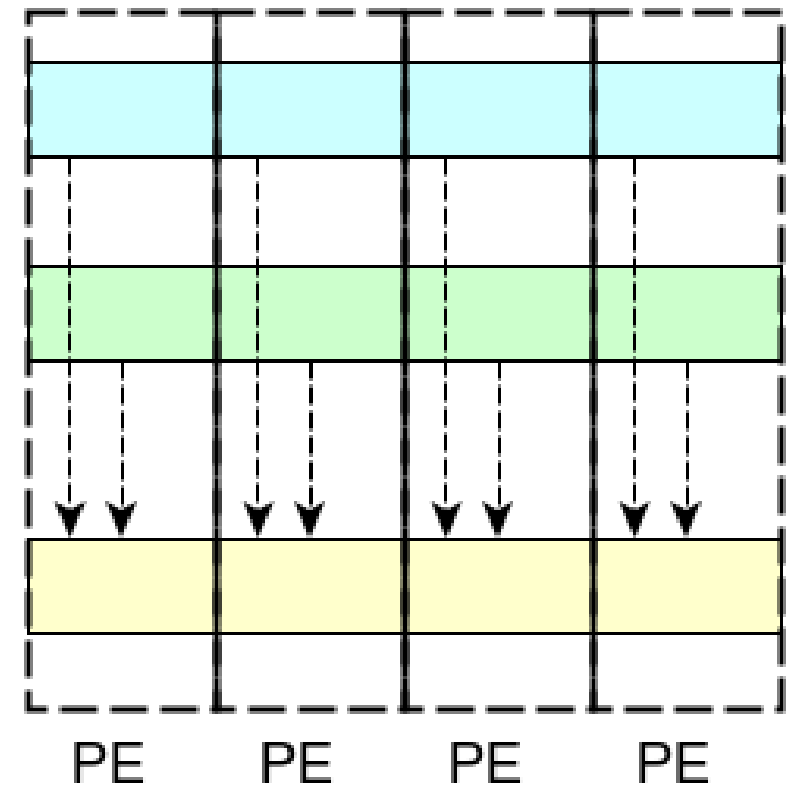
- Мы пока плохо понимаем что такое `__global`
- Кроме того, такое чувство, что тут написано сложение одного элемента векторов
- Это связано с тем, что устройство OpenCL это **SIMT**

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**.

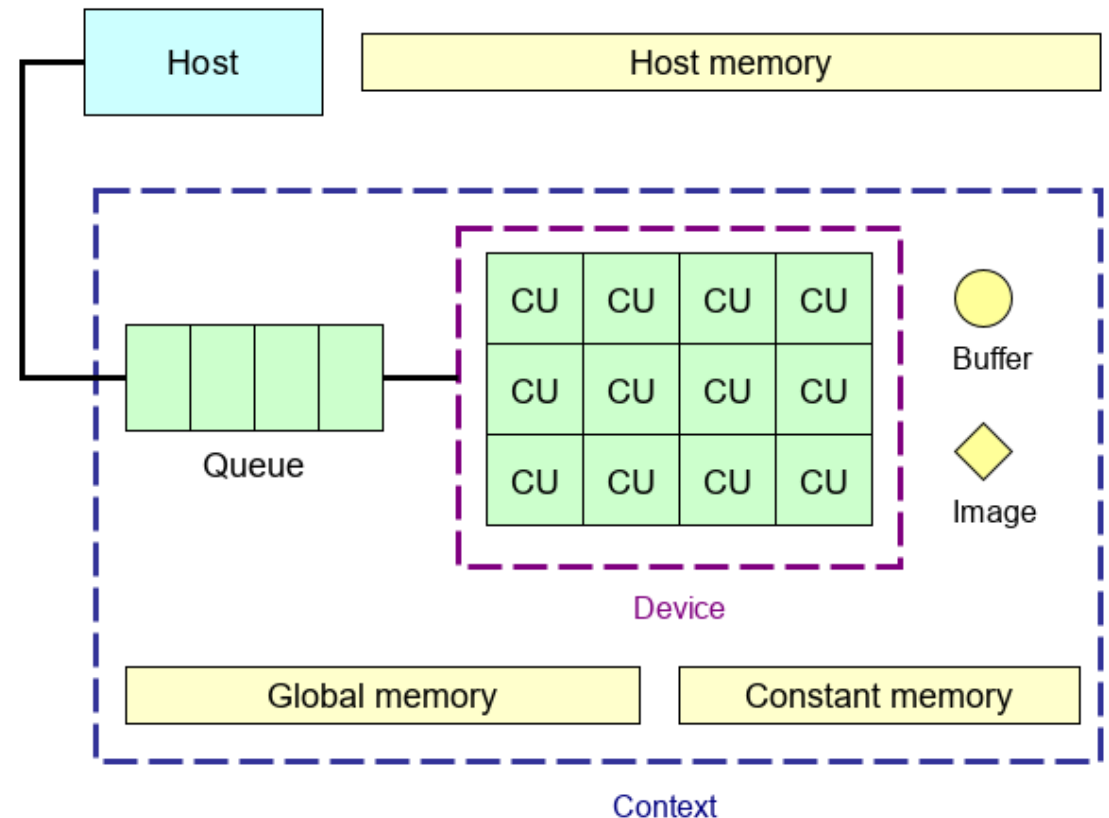
Итерационное пространство задачи

- 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.

C++ API: kernel functor

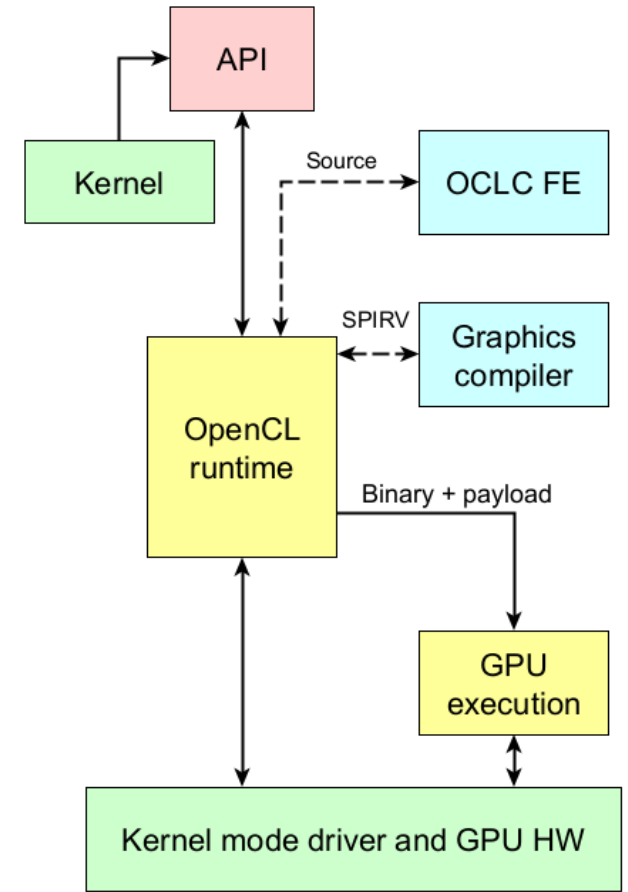
```
cl::Context Context{CL_DEVICE_TYPE_GPU, prop};
cl::CommandQueue Queue{Context};

// 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"};

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



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

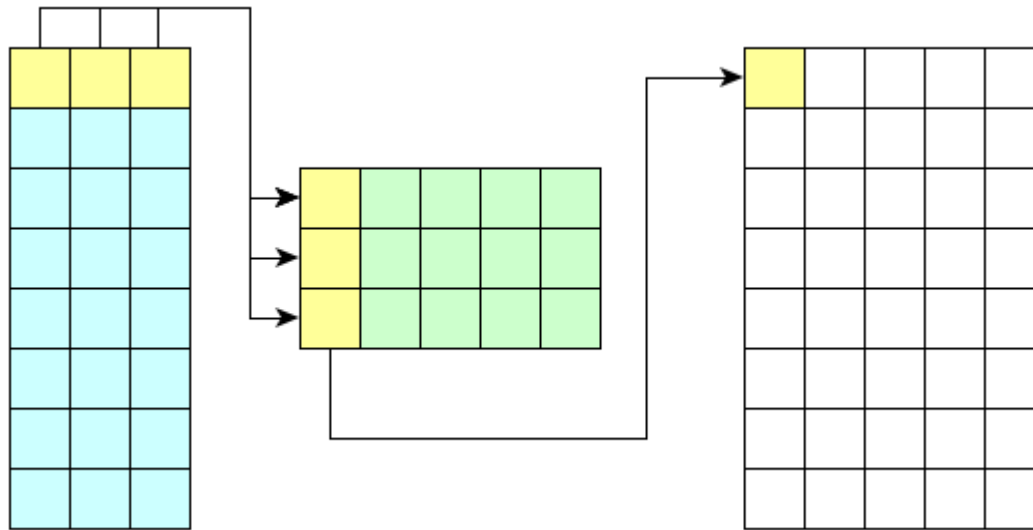
- Память и синхронизация

- ❑ OpenCL C++ API

- ❑ Битоническая сортировка

Обсуждение: а что для матриц?

- Вам нужно умножить матрицу ($N \times M$) на матрицу ($M \times K$)
- 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.

Обсуждение

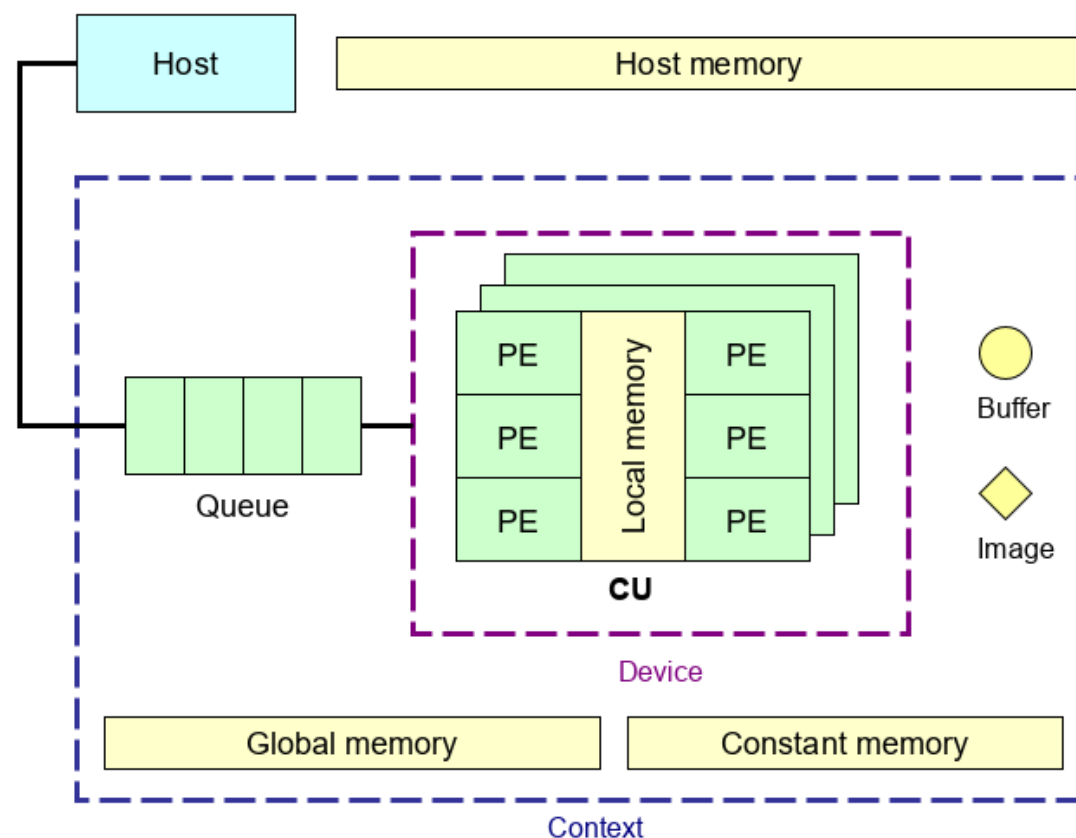
- Работа с памятью это всегда проблемы производительности

```
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 буффер 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

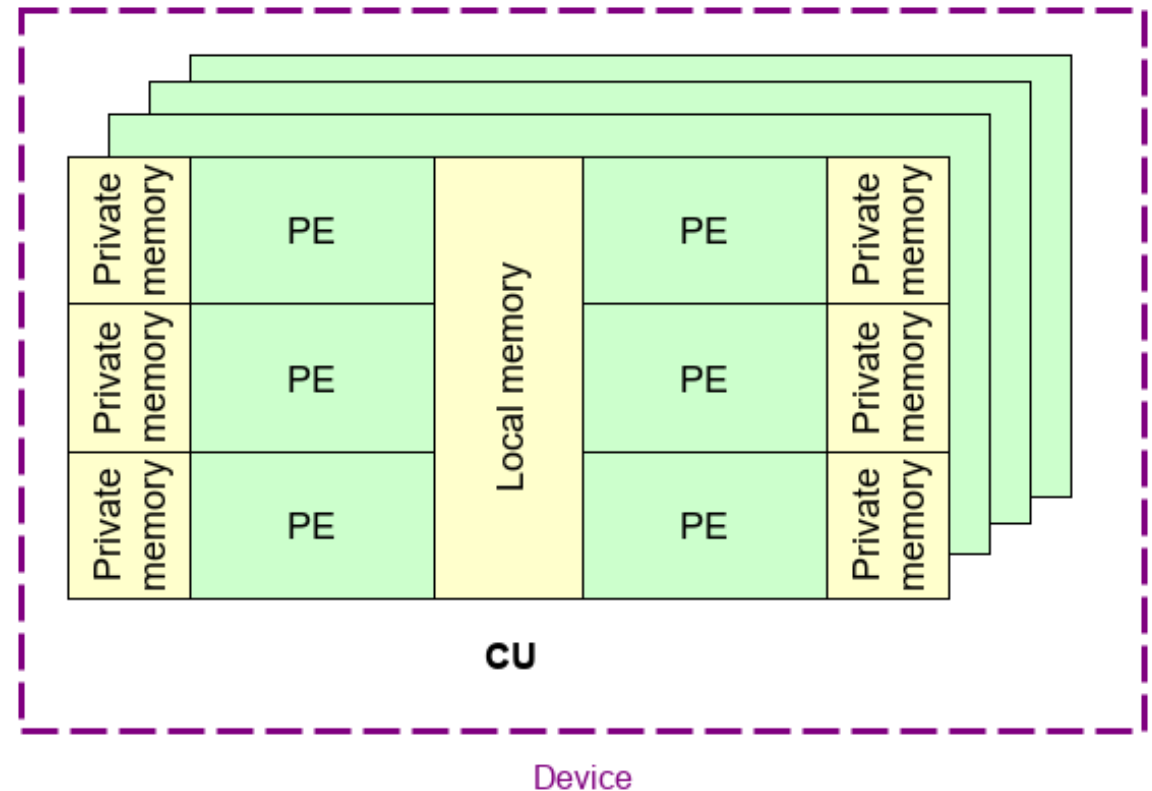
```
__kernel void matrix_multiply(__global float *A,  
    __global float *B, __global TYPE *C, int AX, int AY, int BY) {  
    const int row = get_local_id(0); // Local row ID (max: TS)  
    const int col = get_local_id(1); // Local col ID (max: TS)  
    ...  
  
    __local TYPE Asub[TS][TS]; // 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).

Вложенный параллелизм

```
parallel for workgroup:
  private_memory<int, 2> S;
  parallel for workitem: S(wi) = 0;

  for (T = 0; T < NT; ++T) {
    parallel for workitem:
      ALocal[...] = A[...]; BLocal[...] = B[...];

    parallel for workitem:
      for (K = 0; K < LSZ; K++)
        S(wi) += ALocal[...] * BLocal[...];
  }
  parallel for workitem: C[...] = S(wi)
```

В общем случае

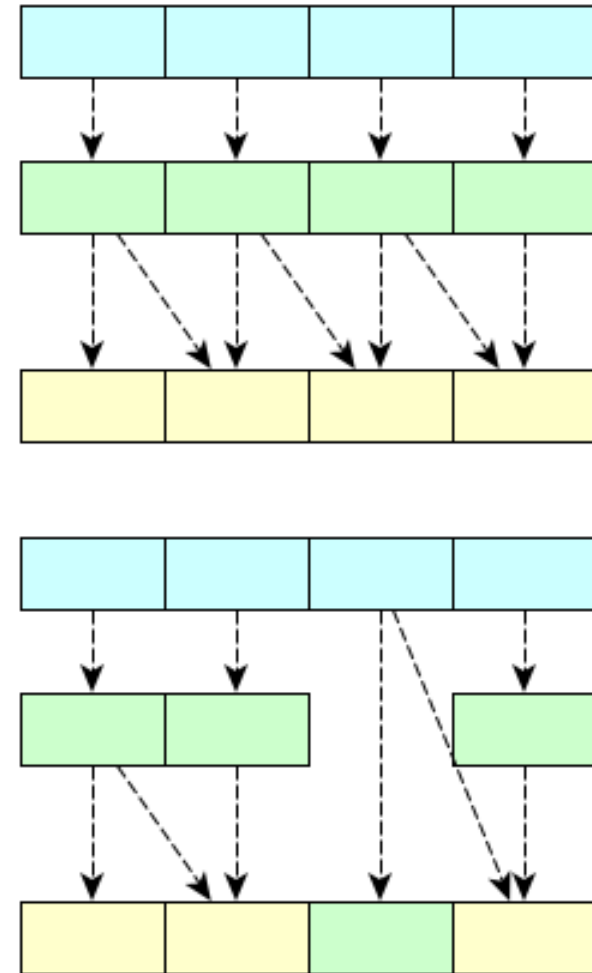
```
for (auto i :  $N_i$ )  
  for (auto j :  $N_j$ )  
    parallel for (auto k :  $N_k$ )  
      do(i, j, k); // порядок по k не важен, и  $N_k$  не велико
```

- Трансформируется в следующий более выгодный вид:

```
parallel for (auto k :  $N_k$ )  
  for (auto i :  $N_i$ )  
    for (auto j :  $N_j$ ) {  
      do(i, j, k); // порядок по k не важен, и  $N_k$  не велико  
      barrier(k); // тут все work items ждут перед итерацией  
    }
```

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



Упражнение в инверсии параллелизма

```
private S = 0;
for (Tile = 0; Tile < NumTiles; ++Tile) {
    parallel for(auto Wi: Workitems)
        copy_global_to_local(Tile, Wi);

    parallel for(auto Wi: Workitems)
        S = calculate_local_mmult(wi);
}

parallel for(auto Wi: Workitems)
    update_global_memory(S, Wi);
```

- Как мы это развернём далее?

Вводим барьеры

```
parallel for(auto Wi: Workitems)
    for (Tile = 0; Tile < NumTiles; ++Tile) {
        copy_global_to_local(Tile, Wi);
        barrier;
    }
```

```
parallel for(auto Wi: Workitems)
    for (Tile = 0; Tile < NumTiles; ++Tile) {
        S = calculate_local_mmult(wi);
        barrier;
    }
```

- И теперь поскольку это параллельные циклы их можно смержить

Волшебство барьеров

```
parallel for(auto Wi: Workitems) {  
    private S = 0;  
    for (Tile = 0; Tile < NumTiles; ++Tile) {  
        copy_global_to_local(Tile, Wi);  
        barrier;  
        S = calculate_local_mmult(wi);  
        barrier;  
    }  
    update_global_memory(S, Wi);  
}
```

- Осталось понять какие фенсы ставить на барьерах. Отложим это пока.

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);

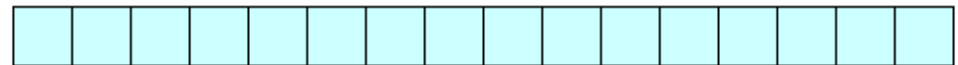
for (k = 0; k < TS; k++)
    acc += Asub[k][row] * Bsub[col][k];

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

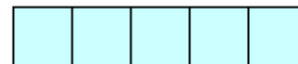
Гистограмма

- 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.

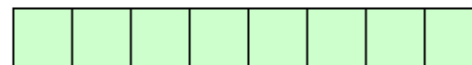
Data array



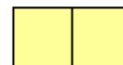
Hist array



Global iteration size



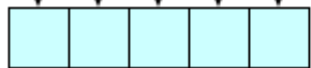
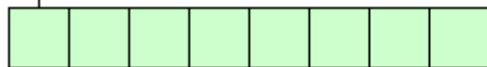
Local iteration size



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?
```

Data array



Hist array

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?

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
```

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!

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

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

Второй шаг: wrapper

- Вrapper хранит в себе объект и вызывает release на уничтожении

```
template <typename cl_type> class Wrapper {  
protected:  
    cl_type obj_;  
  
public:  
    Wrapper(cl_type obj = NULL) : obj_(obj) {}  
    ~Wrapper() { if (obj_) release(); }  
  
    // делегирует к ReferenceHandler<cl_type>::release(obj_)  
    cl_int release() const;
```

- 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?

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 вращеру виртуальный деструктор?

Третий шаг: девайс

- Теперь конкретный класс для девайса может быть унаследован от wrappers

```
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;  
}
```

- 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

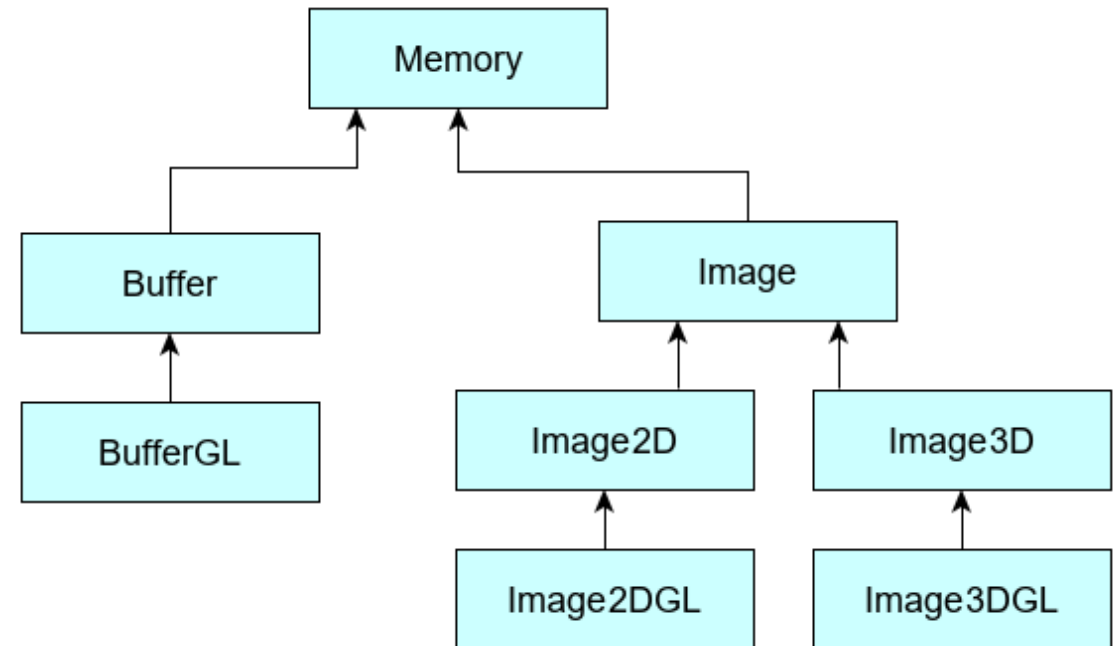
- Теперь используем здесь наши type traits

```
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

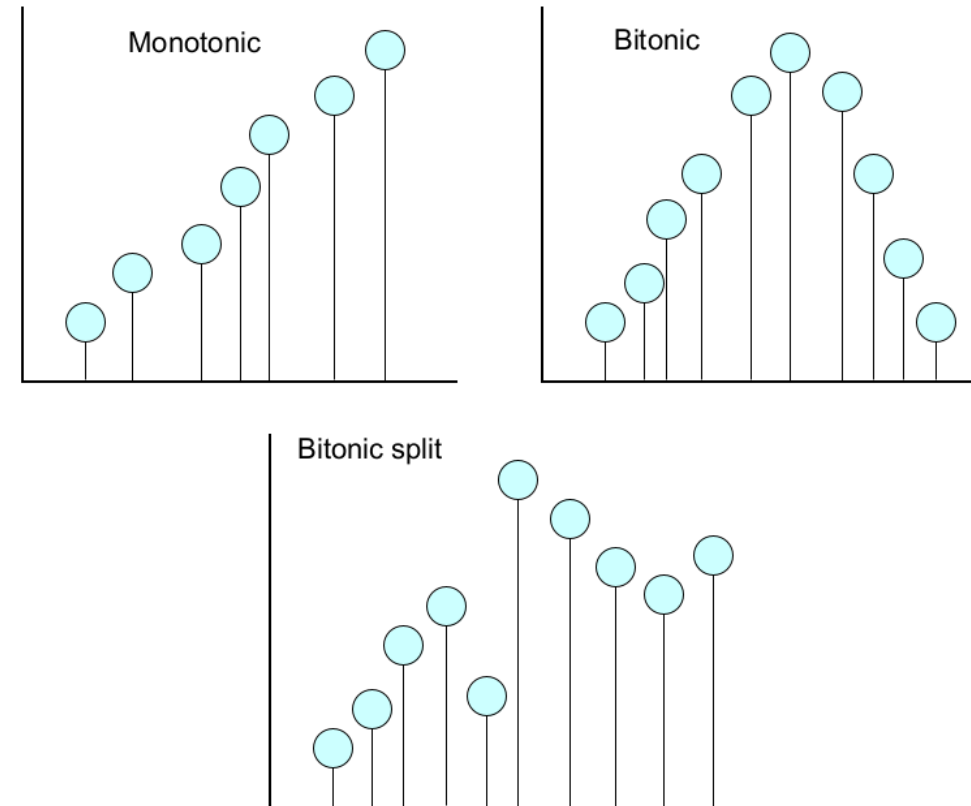
- ❑ Память и синхронизация

- ❑ 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

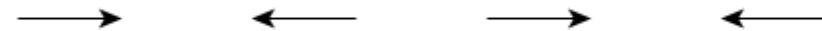


Bitonic merge

- 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.

6	1	4	5	7	2	3	8
---	---	---	---	---	---	---	---

1	6	5	4	7	2	3	8
---	---	---	---	---	---	---	---



1	4	5	6	8	7	3	2
---	---	---	---	---	---	---	---



НWB: 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

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_Gen12LP1p.bin`
 - Изучение ассемблера.
- Дополнительные темы в C++
 - `chrono`, `random`, `charconv`, `system_error`