LUDI VULKANALICI

BUILDING A MODERN C++ FORGE FOR COMPUTE AND GRAPHICS

Alexandru Voicu

alex_voicu@outlook.com

PART (I – I): GOALS

Present a minimal but sufficient C++ framework for driving Vulkan:

- Abstract model
- Initialisation
- Data movement
- Compute
- Graphics
- Testing

Different from what e.g. the SDK suggests.

I came, I saw, I shuddered:

- SDK samples are somewhat daunting for the neophyte
- The other sources of information – discussions, implementations, IHV advice – did not prove to be more encouraging
- Rendering two triangles shouldn’t be hard…right?
- “High performance code is always convoluted, deal with it”☹

PART I: FOUNDATION

Let us try to start from the bottom: `return_type foo(workload_type bar);`

Let us think:

- `return_type` ∈ {`void`, `future<void>`}?
- `workload_type` ∈ {`pipeline<compute>`, `pipeline<graphics>`}?
- we probably need some I/O…possibly `Buffer<T>` and `Texture<T>`?
- perhaps it would be worthwhile to choose the locus of execution? Therefore:

```
void run_pipeline(Pipeline<T> p, locus_type l);
future<void> run_pipeline(Pipeline<T> p, locus_type l);
```

PART I: FOUNDATION

Some objectives:

- `Regular` types by default
- If possible, `TotallyOrdered`
- If unfortunate, `SemiRegular`
- If no other way, deep thought
- `Swappable` types by default (follows from the above)

The locus of execution should be (associated) with some sort of accelerator – let's see what Vulkan offers here:

- `VkInstance` exposes {$\texttt{VkPhysicalDevice}_0$, …, $\texttt{VkPhysicalDevice}_n$}
- `VkInstance` lifetime under programmer control, `VkPhysicalDevice` can be regarded as immutable
- `VkInstance` exposes useful controllable knobs: layers, extensions
- `VkPhysicalDevice` can be queried for important properties of the accelerator – don't lose it!

```
const Vulkan_handle<VkInstance>& default_instance(bool debug_on) {
    static constexpr VkApplicationInfo ai = { /* … */};
    static constexpr const char* l[] =
                       {"VK_LAYER_LUNARG_standard_validation"};
    static const vector<VkExtensionProperties> es = extensions();
    static const vector<const char*> e = names(es);
    static const VkInstanceCreateInfo d = { /* … */ };
    static const VkInstanceCreateInfo r = { /* … */ };
    static const Vulkan_handle<VkInstance> i_dbg{make_instance(d)};
    static const Vulkan_handle<VkInstance> i_rel{make_instance(r)};

    return debug_on ? i_dbg : i_rel;
}
```

```
template<typename T>
class Vulkan_handle:private Equality_comparable<Vulkan_handle<T>>,
                    private Less_than_comparable<Vulkan_handle<T>>,
                    private Swappable<Vulkan_handle<T>>,
                    private TotallyOrdered_check<Vulkan_handle<T>>{
/* … */
};
```

```
friend T handle(const Vulkan_handle& x) { return x.handle(); }

T h_ = nullptr;
Deleter<T> d_;

bool equal_to_(const Vulkan_handle& x) const { return h_ == x.h_; }
bool less_than_(const Vulkan_handle& x) const { return h_ < x.h_; }
void swap_(Vulkan_handle& x) {
    using std::swap;
    swap(h_, x.h_);
    swap(d_, x.d_);
}
public: /* … */
```

```
Vulkan_handle(const Vulkan_handle&) = ?;
Vulkan_handle(Vulkan_handle&&)      = ?;
template<typename... Us>
    requires(is_constructible<Deleter<T>, Us...>)
Vulkan_handle(T h, Us&&... deleter_args)
    : h_{h}, d_{std::forward<Us>(deleter_args)...} {}

Vulkan_handle& operator=(Vulkan_handle x) { swap(*this, x); return *this; }

T handle() const { return h_; }
explicit operator bool() const { return h_ != nullptr; }

~Vulkan_handle() { if (h_) d_(h_); h_ = nullptr; }
```

PART I: FOUNDATION

Minimal set of types:

- Accelerator_pool
- Accelerator – exposes interface for working with a physical device
- Accelerator_view – exposes interface for a locus of execution
- Command_buffer
- Shader<…>
- Buffer<T>, Texture<T>
- Pipeline<…> - associates I/O (Buffer, Texture, constants) and compute (Shader)

```
template<TotallyOrdered M>
class Accelerator_view_concept
               : public Enable_downcast<M>,
                 private Equality_comparable<Accelerator_view_concept<M>>,
                 private Less_than_comparable<Accelerator_view_concept<M>>,
                 private Swappable<Accelerator_view_concept<M>> {
    friend class Equality_comparable<Accelerator_view_concept>;
    /* … */
    template<FunctionalProcedure F>
        requires(Domain<F> == void)
    friend
    decltype(auto) command_pool(const Accelerator_view_concept& x, F f) {
        return x.command_pool(f);
    }
    /* … */
```

```
    using Enable_downcast<M>::model;
    /* … */
    void swap_(Accelerator_view_concept& x) {
        model().sw_(x.model());
    }
public:
    /* … */
    template<FunctionalProcedure F>
        requires(Domain<F> == void)
    decltype(auto) command_pool(F f) const {
        return model().cp_(f);
    }
};
```

```
    using Enable_downcast<M>::model;
    /* … */
    void swap_(Accelerator_view_concept& x) {
        model().sw_(x.model());
    }
public:
    /* … */
    template<FunctionalProcedure F>
        requires(Domain<F> == void)
    decltype(auto) command_pool(F f) const {
        return model().cp_(f);
    }
};
```

```
class Vulkan_accelerator_view
        : public Accelerator_view_concept<Vulkan_accelerator_view>,
          private TotallyOrdered_check<Vulkan_accelerator_view> {
    friend class Accelerator_view_concept<Vulkan_accelerator_view>;

    Vulkan_accelerator const* a_ = nullptr;
    vector<VkExtensionProperties> e_;
    vector<VkQueueFamilyProperties> q_;
    VkPhysicalDeviceFeatures f_ = {};
    Vulkan_handle<VkDevice> d_ = nullptr;
    vector<pair<Vulkan_handle<VkCommandPool>,
                vector<Vulkan_handle<VkQueue>>>> pq_;
    /* … */
```

```
template<FunctionalProcedure F>
    requires(Domain<F> == void && Codomain<F> == VkQueueFlags)
decltype(auto) q_idx_(F f) const {
    return min_element(std::cbegin(q_),
                       std::cend(q_),
                       [=](auto&& x, auto&& y) {
      if (x.queueFlags - (x.queueFlags ^ f()) != f()) return false;
      if (y.queueFlags - (y.queueFlags ^ f()) != f()) return true;
      return x.queueFlags < y.queueFlags;
    }) - cbegin(q_);
}
template<FunctionalProcedure F>
    requires(Domain<F> == void && Codomain<F> == VkQueueFlags)
Cmd_pool_t_ cp_(F f) const {return pq_[q_idx_(f)].first.handle();}
```

```
Vulkan_accelerator_view(const Vulkan_accelerator_view&) = ?;
Vulkan_accelerator_view(const Vulkan_accelerator& a)
    : a_{&a}, e_{extensions(a_->handle())}, q_{queues(a_->handle())},
      f_{features(a_->handle())}, d_{make_device(a_->handle(), e_, q_, f_)},
      pq_{size(q_)} {
     for (decltype(size(q_)) i = 0u; i != size(q_); ++i) {
         Vulkan_handle<VkCommandPool> p{make_command_pool(d_.handle(), i),
                                         d_.handle()};
         vector<Vulkan_handle<VkQueue>> q{q_[i].queueCount};
         for (auto j = 0u; j != q_[i].queueCount; ++j) {
             q[j] = make_queue(d_.handle(), i, j);
         }
         pq_[i] = make_pair(move(p), move(q));
     }
}
```

```
VkDevice make_device(VkPhysicalDevice pd, /*…*/ es, /*…*/ qs, /*…*/ df) {
    VkDevice d = nullptr;
    if (pd) {
        const auto e = names(es);
        vector<VkDeviceQueueCreateInfo> qci;
        static const vector<float> t{64u, 1.0f};
        for (decltype(size(qs)) i = 0u; i != size(qs); ++i) {
            VkDeviceQueueCreateInfo qi = {/* … */qs[i].queueCount,data(t)};
            qci.push_back(move(qi));
        }
        VkDeviceCreateInfo dci = {/*…*/size(e), data(e), &df};
        vkCreateDevice(pd, &dci, nullptr, &d);
    }
    return d;
}
```

Vulkan exposes multiple memory spaces, which can be synthesised as:

- Accelerator exclusive
- (Fast) Accelerator-Host shared
- Host exclusive

PART II: DATA MOVEMENT

Two main container types:

- Buffer<T> - contiguous sequence of bytes
- Texture<T> - contiguous or swizzled sequence of bytes

Let us focus on Buffer<T> (why?)

PART II: DATA MOVEMENT

`Buffer<T>`:

- Can live in any of the memory spaces exposed by an `Accelerator_view` e.g.:

```
Buffer<int> a{av, cnt, dptr, Generic_buffer{}, Accelerator_memory{}};
Buffer<int> b{some_av, cnt, dptr, Generic_buffer{}, Fast_shared_memory{}};
```

- Can be generic (see above) or special purpose e.g.:

```
Buffer<int> c{some_av, cnt, dptr, Index_buffer{}, Accelerator_memory{}};
```

- Can be copied between memory spaces and `Accelerator_views` e.g.:

```
copy(a, b);
copy_async(b, c);
```

All `Buffer<T>`s that are in Accelerator-Host shared memory are mapped for the entirety of their lifetime – let's automate this via `Mapped_pointer<T>`:

```
template<typename T> class Vulkan_mapped_pointer : /*…*/ {
    VkDevice d_ = nullptr;
    VkDeviceMemory m_ = nullptr;
    vector<char> b_;
    void* pb_ = nullptr;
    decltype(size(b_)) s_ = 0u;
    void* p_ = nullptr;
public:
    Vulkan_mapped_pointer(const Vulkan_mapped_pointer&) = ?;
    Vulkan_mapped_pointer(Vulkan_mapped_pointer&& x) = ?;
    /*…*/
```

```
Vulkan_mapped_pointer(const Accelerator_view& d,
                      const Vulkan_handle<VkDeviceMemory>& m)
    : d_{handle(d)},
      m_{handle(m)},
      b_(map_alignment(accelerator(d))),
      pb_{reinterpret_cast<void*>(data(b_))},
      s_{size(b_)},
      p_{align(map_alignment(accelerator(d)), sizeof(p_), pb_, s_)} {
        vkMapMemory(d_, m_, 0u, VK_WHOLE_SIZE, 0u, &p_);
    }
~Vulkan_mapped_pointer() {
    if (d_ && m_) vkUnmapMemory(d_, m_);
    p_ = nullptr;
}
```

`Mapped_pointer<T>` also enables familiar RAII based scoped mapping:

```
Buffer<int> a{av,
              cnt,
              dptr,
              Generic_buffer{},
              Fast_shared_memory{}};
{
    Mapped_pointer<int>  p{accelerator_view(a),  memory(a)};
    generate_n(p, p + size(a), rand);
} // Unmapped at p’s destruction.
```

Having said this, how do we actually copy something?

In Buffer_concept:

```
friend
void copy(const Buffer_concept& src, Buffer_concept& dst)
    src.copy_to_(dst);
}
void copy_to_(Buffer_concept& x) const { model().cp_(x.model()); }
```

In Vulkan_buffer:

```
void cp_(Vulkan_buffer& x) const {
    switch (buffers_location_(*this, x)) {
    case Host_host: cp_host_host_(x); break;
    case Host_accl: cp_host_accl_(x); break;
    case Accl_accl: cp_accl_accl_(x); break;
    case Accl_host: cp_accl_host_(x); break;
    }
}
void cp_host_accl_(Vulkan_buffer& x) const {
    constexpr size_t inline_copy = 65536; // Bytes.
    if (sz_() * sizeof(T) <= inline_copy) cp_host_accl_small_(x);
    else cp_host_accl_large_(x);
}
```

```
void cp_host_accl_small_(Vulkan_buffer& x) const {
    Command_buffer c{*x.s_.a_,
                     Transfer_queue{},
                     [this, &x](auto&& cb) {
        vkCmdUpdateBuffer(cb,
                          x.handle(),
                          0u,
                          this->sz_() * sizeof(T),
                          this->dptr_());
    }};
    c(Synchronous{});
}
```

```
void cp_host_accl_large_(Vulkan_buffer& x) const {
    Vulkan_buffer t{*x.s_.a_,
                    dptr_(),
                    sz_(),
                    Generic_buffer{},
                    Fast_shared_memory{}};
    Command_buffer c{*x.s_.a_,
                     Transfer_queue{},
                     [&t, &x](auto&& cb) {
        VkBufferCopy bc{0u, 0u, t.sz_() * sizeof(T)};
        vkCmdCopyBuffer(cb, t.handle(), x.s_.b_.handle(), 1u, &bc);
    }};
    c(Synchronous{});
}
```

```
class Vulkan_command_buffer : /*…*/ {
    static constexpr VkCommandBufferBeginInfo cbi_ = { /*…*/ };

    Vulkan_accelerator_view const* a_ = nullptr;
    Vulkan_handle<VkCommandBuffer> c_ = nullptr;
    VkQueueFlags qf_;
    function<void(VkCommandBuffer)> f_;
public:
    Vulkan_command_buffer(const Vulkan_command_buffer&) = ?;
    Vulkan_command_buffer(Vulkan_command_buffer&&) = ?
    /*…*/
};
```

```
template<FunctionalProcedure F, FunctionalProcedure G>
    requires(Domain<F> == void && Codomain<F> == VkQueueFlags &&
             Domain<G> == VkCommandBuffer && Codomain<G> == void)
Vulkan_command_buffer(const Vulkan_accelerator_view& a, F f, G g)
    : a_{&a},
      c_{make_command_buffer(a.handle(), command_pool(a, f)),
         a.handle()},
      qf_{f()},
      f_{std::move(g)} {
    vkBeginCommandBuffer(handle(), &cbi_);
    f_(handle());
    vkEndCommandBuffer(handle());
}
```

```
VkFence submit(VkDevice d, VkCommandBuffer b, VkQueue q,
               const std::vector<VkSemaphore>& s_pre = {},
               const std::vector<VkPipelineStageFlags>& p = {},
               const std::vector<VkSemaphore>& s_post = {}) const {
    VkFence f = make_fence(d);
    if (q && f) {
        VkSubmitInfo si = {/*…*/ 1u, &b, /*…*/};
        vkQueueSubmit(q, 1u, &si, f);
    }
    return f;
}
```

```
future<void> operator()(Asynchronous) const {
    Vulkan_handle<VkFence> f{submit(/*…*/), a_->handle()};
    return std::async(std::launch::deferred,
                      [this](Vulkan_handle<VkFence> f) {
        const VkFence fs[] = {f.handle()};
        vkWaitForFences(a_->handle(), size(fs), fs, true,UINT_MAX);
    }, move(f));
}

void operator()(Synchronous) const {
    operator()(Asynchronous{}).get();
}
```

PART III: COMPUTE

Now that we know how to move data, perhaps we can do something interesting.

Let us try to sort it – any ideas about (simple) algorithms we might use?

```
layout(std430, binding = 0) buffer a { int data[]; };
layout(std430, binding = 1) buffer b { uint unsorted; };
layout(push_constant) uniform c { uint odd; } constants;

shared uint unsorted_cnt;
layout(local_size_x = /*…*/) in;
void main() { /*…*/ }
```

```
void main() {
    unsorted_cnt = 0u;
    const uint tidx = gl_WorkGroupID.x * twice(gl_WorkGroupSize.x);
    const uint lidx = gl_LocalInvocationID.x;
    const uint gidx = tidx + (twice(lidx) + constants.odd);

    if ((successor(gidx) < data.length())) {
        const uint u = uint(compare_exchange(data[gidx],
                                             data[successor(gidx)]));
        atomicAdd(unsorted_cnt, u);
    }
    if (positive(unsorted_cnt) && zero(lidx)) atomicAdd(unsorted, 1u);
}
```

What we need from the host:

- Mutable `Buffer<T>` holding the data to be sorted;
- A `Buffer<unsigned int>` for holding the continuation condition;
- Constant controlling whether it's an odd or an even pass;
- Submission to a Compute or Generic Queue;
- Status check.

Where should we put the `Buffers`?

Should we wait or should we go?

A (suboptimal) solution:

```
Accelerator_view av{Accelerator::get_default()};
vector<int> d(sz);
generate_n(data(d), sz, rand);

Buffer<int> in{av, data(d), sz, Generic_buffer{}, Accelerator_memory{}};
Buffer<unsigned> unsorted{av, 1u, Generic_buffer{}, Fast_shared_memory{}};
Pipeline<Compute> p0{av, {in, “a”}, {unsorted, “b”},{1u, “odd”},R”(/**/)”};
Pipeline<Compute> p1{av, {in, “a”}, {unsorted, “b”},{0u, ”odd”},R”(/**/)”};
// Perhaps the Pipeline<Compute> constructor is missing something?
do { run_pipeline(p0, av); run_pipeline(p1, av); } while (unsorted[0]);
```

Vulkan and GLSL – it’s complicated…

Perhaps we should rewrite that directly in SPIR-V…

It might be nicer to just use Shaderc, i.e.:

```
template<Shader_type t> class Shader : /*…*/ {
    Vulkan_accelerator_view const* a_ = nullptr;
    string s_;
    Vulkan_handle<VkShaderModule> m_ = nullptr;
    /*…*/
public:
    Shader(const Accelerator_view& a, string s)
        : a_{&a}, s_{move(s)}, m_{make_shader(handle(a), s_, t),
                                  handle(a)}
    {}
    /*…*/
};
```

```
VkShaderModule make_shader(VkDevice d, const string& src,
                           Shader_type t) {
    VkShaderModule s = nullptr;
    if (d) {
        shaderc::Compiler c;
        shaderc::Compiler o;
        const auto m = c.CompileGlslToSpv(src, t, “”, o);
        const size_t sz = (cend(m) – cbegin(m)) * sizeof(uint32_t);
        VkShaderModuleCreateInfo si = {/*…*/ m, cbegin(m)};
        vkCreateShaderModule(d, &si, nullptr, &s);
    }
    return s;
}
```

Note that the `Pipeline` constructor takes care of patching the source string:

- As it traverses its arguments it associates unique binding indexes with e.g. `Buffer<T>`s, and records the name
- It associates unique push_constant offsets for (Vulkan compatible) constants of trivial type, and records the name
- Once traversal is complete it patches the string before passing it to the `Shader<T>` constructor

PART IV: GRAPHICS

At this point, handling a graphics workload is just more of the same:

- Pipeline<Graphics> type which works equivalently but has different invariants and some extra bindings (e.g. for the Vertex_buffer and Index_buffer)
- Submission is done against a graphics capable queue
- Necessary to interact with native windowing systems – if we want to see the results

If curious / interested, please peek at the code sample that will be added to the presentation slightly later

Now we are going to discuss something slightly more philosophical: Async *Compute*

As we've already seen, it's possible for an Accelerator to expose multiple queue types, with differing capabilities

It's possible to submit work against these various queues or just hit the guaranteed to exist Universal_queue – what is the optimal choice?

As is usually the case…it depends!

It's perfectly fine for an implementation to serialise all work, even if submitted against different queues – there's no promise of parallel execution

There are implementations / GPUs which do benefit from this overlap:

- Wide GPUs that have trouble scheduling sufficient work for their sea of ALUs
- GPUs with decoupled DMA engines which can thus decouple all work submitted against the Transfer_queue
- Etc.

The more profound question is, should this (submitting against multiple queues) be done by default or not? In my opinion, yes!

PART IV: GRAPHICS

The programmer has an algorithmic / logical view:

- If you have a Compute shader, why run it anywhere but on a Compute queue?
- If you're just reading or writing a buffer, why do it on the Universal queue?
- Ignore potential overheads (unless you can prove them to be crushing)

Don't expect massive performance wins by default:

- Only cases where there are idle resources benefit e.g. overlap Graphics work that is light in terms of ALU work (shadow pass) with arithmetically intensive Compute

So, does all of this stuff that we've seen help in any way with balancing queue payload?

```
template<FunctionalProcedure F>
    requires(Domain<F> == void && Codomain<F> == VkQueueFlags)
VkQueue Accelerator_view::qe_(F f) const {
    static std::vector<unsigned int> qid(std::size(q_));
    const auto q = q_idx_(f);
    return pq_[q].second[qid[q]++ % q_[q].queueCount].handle();
}
```

CONCLUSIONS

I had abandoned any hope of ever making it this far during – therefore no conclusions!

PART V: SOME USEFUL LINKS

Catch++: https://github.com/philsquared/Catch

GLI: https://github.com/g-truc/gli

GLM: https://github.com/g-truc/glm

LunarG Vulkan SDK: https://vulkan.lunarg.com/

RenderDoc: https://github.com/baldurk/renderdoc

Shaderc: https://github.com/google/shaderc/

?

THANK YOU!