Эффективный запуск compute-шейдеров в Adept

Кирилл Колодяжный

Ведущий инженер по разработке ПО

- Разработка систем хранения данных
- Машинное обучение
- 3Д-реконструкция
- Книга «Hands-On Machine Learning with C++»
- Курс «Архитектура платформ машинного обучения»

YADRO

Что мы рассмотрим?

Реализацию конвейера запуска вычислительных ядер для Vulkan

01

Модели конкурентности на GPU

CUDA, OpenCL, Vulkan

02

Архитектура Adept

Ключевые компоненты и их задачи

03

Доступ к памяти

Автоматическая генерация барьеров

04

Управление ресурсами

Кэширование, отслеживание, пакетирование

05

Производительность

Настройка параметров и сравнение

Проект Adept

Automatic Differentiation Engine for Tensor Processing

Библиотека для построения и обучения ML-моделей

Dynamic graphs
Eager-mode как в PyTorch: отладка, условия, циклы

Tensors + Autograd
Авто-дифференцирование с ускорением на GPU

Vulkan backend
NVIDIA, AMD, Intel, Qualcomm

Unified Python/C++ semantics
Легко адаптировать и переносить код

Adept: образовательный проект

01

Основные архитектурные блоки платформ машинного обучения

02

Разные уровни абстракции

03

Взаимодействие нескольких языков программирования

04

Реализация математических абстракций

05

Метод автоматического дифференцирования

06

Реализация распространённых модулей ML-моделей

Vulkan — единый API для различных GPU

Vulkan®

Embedded GPU | Mobile SoC GPU | Server GPU | Desktop GPU

Integrated graphics → Embedded GPU

Adreno, Mali, PowerVR → Mobile SoC GPU

NVIDIA, AMD, Intel → Server GPU, Desktop GPU

Embedded GPU, Mobile SoC GPU, Server GPU, Desktop GPU → Vulkan

Vulkan — тренд для Inference на устройствах

ExecuTorch

Instagram[1], WhatsApp

TFLite / LiteRT

Airbnb, PayPal

NCNN

QQ, Qzone, WeChat, Pitu

MNN

Taobao, Tmall, Youku

[1] Деятельность организации запрещена на территории РФ

Концепция конвейера вычислений

Управление запуском вычислений и синхронизация на хосте

GPU

Host Memory → Load → Device Memory → Compute Kernel → Device Memory → Compute Kernel → Device Memory → Store → Host Memory

```
auto a = tensor({1.0f,
2.0f, 3.0f});
auto b = tensor({4.0f,
5.0f, 6.0f});
auto c = tensor({0.0f,
0.0f, 0.0f});
```

```
a = a.gpu();
b = b.gpu();
c = c.gpu();
```

```
c = a + b;
```

```
c = c * 2.0f;
```

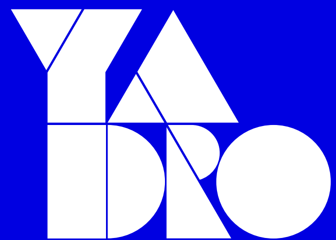
```
cout << c.item();
```

Host / CPU

barrier | barrier | barrier

Program Flow

Модели конкурентности в разных API



CUDA stream: семантика одного потока

Host | Stream | GPU

Host → Stream: Kernel A
Stream → GPU: Execute A
GPU ⇢ Stream: A Complete
Host → Stream: Kernel B
Stream → GPU: Execute B

Host | Stream | GPU

Strict FIFO Order

CUDA: видимость памяти внутри потока

Дополнительный код не нужен: неявная синхронизация внутри потока

Kernel A	Memory	Kernel B
Write Data →		
	Visibly Guaranteed	
Completion Signal - - - →		→
	← Read Data	
		Sees Fresh Data

OpenCL: упорядоченные vs неупорядоченные очереди

Характеристика	Упорядоченная очередь	Неупорядоченная очередь
Исполнение	Строгий FIFO	Зависит от событий
Зависимости	Неявные	Явные (события)
Сложность	Низкая	Высокая
Параллелизм	Ограничен	Высокий

Vulkan Compute vs. Graphics Pipeline

Вычислительный конвейер

Shader Module (Compute) → Compute Pipeline → Dispatch

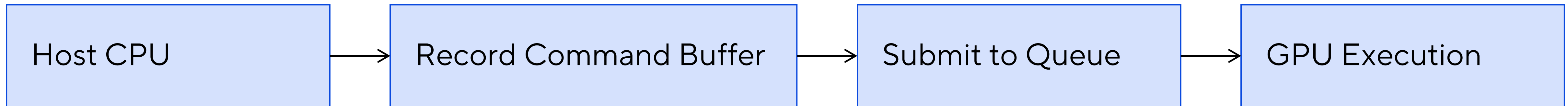
Общие ресурсы
Image Attachments
(color, depth, storage)

Графический конвейер (много этапов)

Render Pass → Framebuffer → Graphics Pipeline → Input Assembly → Vertex Shader → Tessellation → Geometry Shader → Rasterizer → Fragment Shader → Color Blend/Depth Stencil → Draw Call

Отличие запуска вычислений в Vulkan

Ключевое отличие:
нет подхода «запустил и забыл» — требуется явная отправка



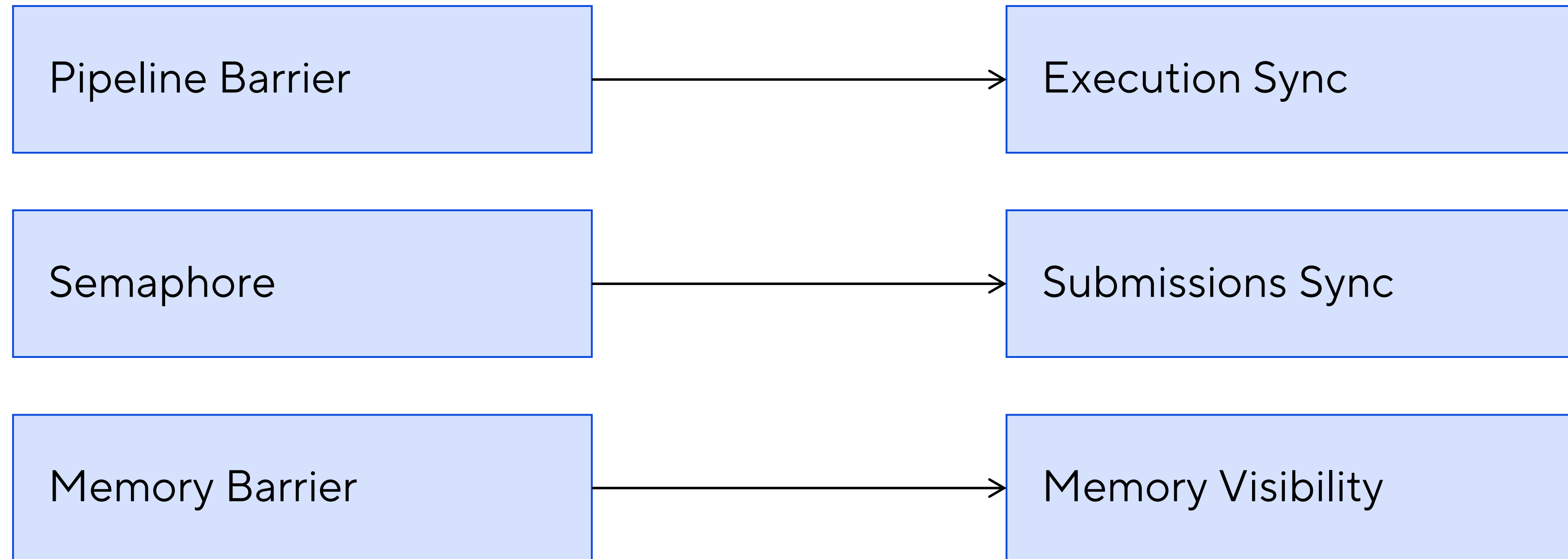
Семейства очередей Vulkan: гетерогенные пути исполнения

GPU Hardware

Compute Engine	Transfer Engine	Graphics Engine
Queue Family 0	Queue Family 1	Queue Family 2
Compute Queue Family	Transfer Queue Family	Graphics Queue Family

Основные примитивы синхронизации Vulkan

API ничего не предполагает — разработчик объявляет все зависимости

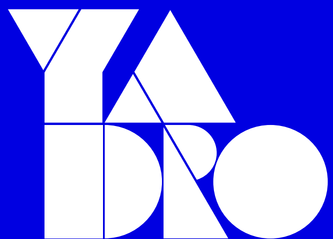


Итоги рассмотрения моделей конкурентности

API/Модель	Порядок исполнения	Видимость памяти
CUDA stream	Строгий FIFO	Автоматическая (внутри потока)
OpenCL Упорядоченная очередь	Строгий FIFO	Автоматическая
OpenCL Неупорядоченная очередь	Зависит от событий	Зависит от событий
Vulkan (без барьеров)	Только порядок отправки	Только порядок отправки
Vulkan (с барьерами)	Явный контроль	Явный контроль

Архитектура Adept

Automatic Differentiation Engine
for Tensor Processing



Архитектурные блоки

CPU:
C++, SIMD intrinsics

GPU:
C++, Shaders, Kernels

Dispatching
C++

NN layers
C++

Autograd
C++

Linear algebra
C++

Top level API
Python / C++

1. Modules
2. Operators
3. Expressions
4. ...

Архитектурные блоки

CPU:
C++, SIMD intrinsics

GPU:
C++, Shaders, Kernels

Dispatching
C++

NN layers
C++

Autograd
C++

Linear algebra
C++

Top level API
Python / C++
1. Modules
2. Operators
3. Expressions
4. ...

Компиляция шейдеров в SPIR-V

Shader Template → Python Generator → Specialized .comp → glslangValidator → SPIR-V .spv → C++ Header → Runtime Cache

Трёхуровневая модель архитектуры

Developer View

Tensor Operation → RUN_SHADER Macro

Adept Layer

Shader Cache → Access Tracking → Barrier Generation → Completion Tracking → Resource Pooling

Vulkan Layer

Command Buffer → Pipeline → Descriptor Set → Queue Submit

Прикладной интерфейс — Developer View

```
// User code: Simple tensor arithmetic
Tensor a = tensor({1.0f, 2.0f, 3.0f});
Tensor b = tensor({4.0f, 5.0f, 6.0f});
Tensor c = tensor({0.0f, 0.0f, 0.0f});

// Sequential operations with automatic dependencies
c = a + b;       // Kernel 1: element-wise add
c = c * 2.0f;    // Kernel 2: scalar multiply
c = c - a;       // Kernel 3: element-wise subtract
```

Прикладной интерфейс — Developer View

```
// User code: Simple tensor arithmetic
Tensor a = tensor({1.0f, 2.0f, 3.0f});
Tensor b = tensor({4.0f, 5.0f, 6.0f});
Tensor c = tensor({0.0f, 0.0f, 0.0f});

// Sequential operations with automatic dependencies
c = a + b;        // Kernel 1: element-wise add
c = c * 2.0f;     // Kernel 2: scalar multiply
c = c - a;        // Kernel 3: element-wise subtract
```

Прикладной интерфейс — Developer View

```
// User code: Simple tensor arithmetic
Tensor a = tensor({1.0f, 2.0f, 3.0f});
Tensor b = tensor({4.0f, 5.0f, 6.0f});
Tensor c = tensor({0.0f, 0.0f, 0.0f});

// Sequential operations with automatic dependencies
c = a + b;      // Kernel 1: element-wise add
c = c * 2.0f;   // Kernel 2: scalar multiply
c = c - a;      // Kernel 3: element-wise subtract
```

Внутренняя реализация — Adept Layer

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Внутренняя реализация — Adept Layer

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Внутренняя реализация — Adept Layer

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Внутренняя реализация — Adept Layer

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Основные компоненты

VkManager

1

*

VkDevice

1 1 1 1

* 1 * *

VkSubmit

<<Resourse Pools>>

VkShader

MemoryPool

1

tracks via storage_ptr

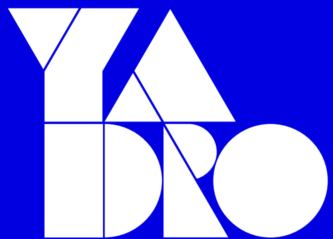
1

VkCommandBuffer

VkBufferBase

Буферы данных

Сокращение количества выделений памяти



Управление буферами данных

CPU

Host Memory

load: memcpy

store: memcpy

GPU

Staging Buffer
Host-Visible

load: vkCmdCopyBuffer

story: vkCmdCopyBuffer

Device Buffer
Device-Local

Staging-буфер — это память, видимая хосту, к которой могут обращаться и CPU, и GPU
Команды копирования выполняются асинхронно на GPU — CPU не блокируется во время передачи

Архитектура буферов данных

<<abstract>>
Buffer

+size() : const
+is_shared() : const
+is_aligned() : const

<<tracking>>
VkBufferBase

+last_access_type_
+last_pipline_stage_

+buffer_for(...) : const

<<concrete>>
VkBuffer

+buffer_
+memory_
+staging_buffer_
+mapped_data_

+load() / store()

allocates from

<<pooling>>
MemoryPool

+allocate(size)
+free(buffer)

Работа LRU пула памяти

Allocate size → Search cache

Search cache → Cache HIT → Remove from Cache → Wrap in PoolStorage

Search cache → Cache MISS → Cache Full?

Cache Full? — No → Allocate New

Cache Full? — Yes → Evict LRU → Allocate New

Allocate New → Wrap in PoolStorage

RAII в PoolStorage

```
struct PoolStorage : public Storage {
  std::weak_ptr<MemoryPool> parent_pool_;
  buffer_ptr_t original_buffer_;

 ~PoolStorage() override {
    auto pool = parent_pool_.lock();
    if (pool) {
      pool->free(std::move(original_buffer_));
    }
  }
};
```

RAII в PoolStorage

```
struct PoolStorage : public Storage {
  std::weak_ptr<MemoryPool> parent_pool_;
  buffer_ptr_t original_buffer_;

 ~PoolStorage() override {
    auto pool = parent_pool_.lock();
    if (pool) {
      pool->free(std::move(original_buffer_));
    }
  }
};
```

RAII в PoolStorage

```
struct PoolStorage : public Storage {
  std::weak_ptr<MemoryPool> parent_pool_;
  buffer_ptr_t original_buffer_;

 ~PoolStorage() override {
    auto pool = parent_pool_.lock();
    if (pool) {
      pool->free(std::move(original_buffer_));
    }
  }
};
```

RAII в PoolStorage

```
struct PoolStorage : public Storage {
  std::weak_ptr<MemoryPool> parent_pool_;
  buffer_ptr_t original_buffer_;

 ~PoolStorage() override {
    auto pool = parent_pool_.lock();
    if (pool) {
      pool->free(std::move(original_buffer_));
    }
  }
};
```

RAII в PoolStorage

```
struct PoolStorage : public Storage {
  std::weak_ptr<MemoryPool> parent_pool_;
  buffer_ptr_t original_buffer_;

 ~PoolStorage() override {
    auto pool = parent_pool_.lock();
    if (pool) {
      pool->free(std::move(original_buffer_));
    }
  }
};
```

MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;       // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;      // Active buffers
  std::mutex guard_;                            // Thread-safe access
  allocator_ptr_t allocator_;               // Backend-specific allocator
  size_t max_size_{1000};                   // Cache size
  float32_t size_diff_ratio_{0.3f};         // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;        // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;       // Active buffers
  std::mutex guard_;                             // Thread-safe access
  allocator_ptr_t allocator_;                // Backend-specific allocator
  size_t max_size_{1000};                    // Cache size
  float32_t size_diff_ratio_{0.3f};          // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;         // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;        // Active buffers
  std::mutex guard_;                              // Thread-safe access
  allocator_ptr_t allocator_;              // Backend-specific allocator
  size_t max_size_{1000};                  // Cache size
  float32_t size_diff_ratio_{0.3f};        // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;        // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;       // Active buffers
  std::mutex guard_;                             // Thread-safe access
  allocator_ptr_t allocator_;                 // Backend-specific allocator
  size_t max_size_{1000};                     // Cache size
  float32_t size_diff_ratio_{0.3f};           // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;       // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;      // Active buffers
  std::mutex guard_;                            // Thread-safe access
  allocator_ptr_t allocator_;               // Backend-specific allocator
  size_t max_size_{1000};                   // Cache size
  float32_t size_diff_ratio_{0.3f};         // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

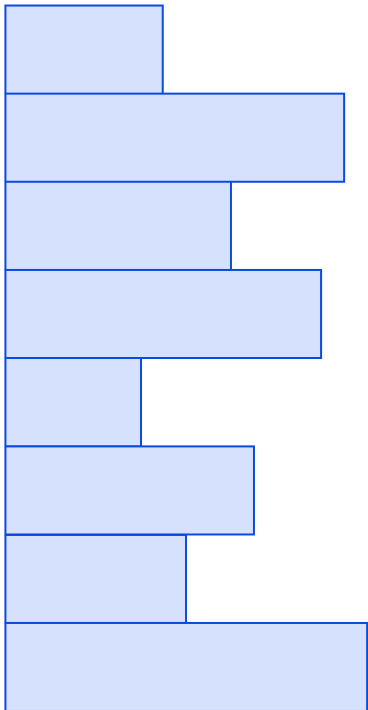
MemoryPool: LRU-кэш

LRU-кэш для буферов памяти GPU с доступом по размеру

```
class MemoryPool {
  // Size-indexed lookup
  std::multimap<size_t, iterator> cache_lookup_;
  std::list<buffer_ptr_t> buffers_cache_;       // LRU order
  std::list<buffer_ptr_t> buffers_in_use_;      // Active buffers
  std::mutex guard_;                            // Thread-safe access
  allocator_ptr_t allocator_;                // Backend-specific allocator
  size_t max_size_{1000};                    // Cache size
  float32_t size_diff_ratio_{0.3f};          // 30% size tolerance
  // ...
  storage_ptr_t allocate(size_t size);
  void free(buffer_ptr_t&& buffer);
};
```

Архитектура MemoryPool

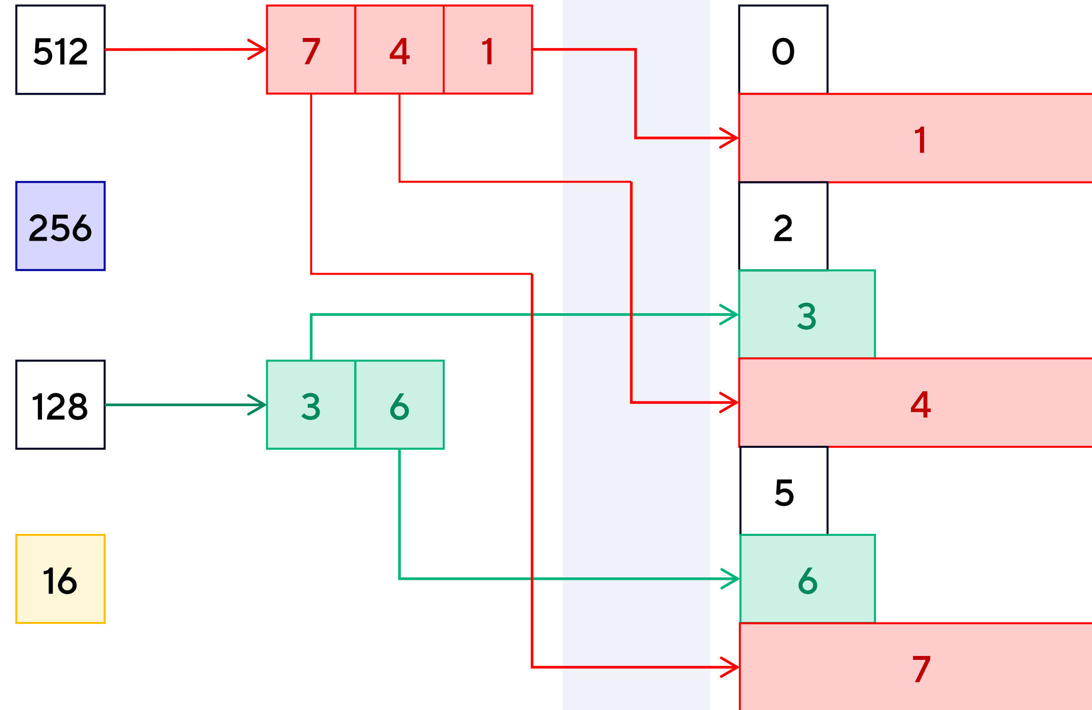
Список используемых буферов



Кеш

Таблица поиска:
Размер — список указателей буферы

Список доступных буферов



Часто используются

Редко используются

Логика MemoryPool::allocate()

```
storage_ptr_t MemoryPool::allocate(size_t size) {
  auto lower_it = cache_lookup_.lower_bound(size);
  // Search with tolerance
  if (lower_it != cache_lookup_.end() &&
      lower_it->first <= size +
                     static_cast<size_t>(size * size_diff_ratio_))) {
    return move_from_cache(lower_it);  // Cache HIT
  }

  if (buffers_cache_.size() >= max_size_) {
    evict_oldest();
  }
  return allocate_new(size);  // Cache MISS
}
```

Логика MemoryPool::allocate()

```
storage_ptr_t MemoryPool::allocate(size_t size) {
  auto lower_it = cache_lookup_.lower_bound(size);
  // Search with tolerance
  if (lower_it != cache_lookup_.end() &&
      lower_it->first <= size +
                     static_cast<size_t>(size * size_diff_ratio_))) {
    return move_from_cache(lower_it);  // Cache HIT
  }

  if (buffers_cache_.size() >= max_size_) {
    evict_oldest();
  }
  return allocate_new(size);  // Cache MISS
}
```

Логика MemoryPool::allocate()

```
storage_ptr_t MemoryPool::allocate(size_t size) {
  auto lower_it = cache_lookup_.lower_bound(size);
  // Search with tolerance
  if (lower_it != cache_lookup_.end() &&
      lower_it->first <= size +
                           static_cast<size_t>(size * size_diff_ratio_))) {
    return move_from_cache(lower_it);  // Cache HIT
  }

  if (buffers_cache_.size() >= max_size_) {
    evict_oldest();
  }
  return allocate_new(size);  // Cache MISS
}
```

Логика MemoryPool::allocate()

```
storage_ptr_t MemoryPool::allocate(size_t size) {
  auto lower_it = cache_lookup_.lower_bound(size);
  // Search with tolerance
  if (lower_it != cache_lookup_.end() &&
      lower_it->first <= size +
                         static_cast<size_t>(size * size_diff_ratio_))) {
    return move_from_cache(lower_it);  // Cache HIT
  }

  if (buffers_cache_.size() >= max_size_) {
    evict_oldest();
  }
  return allocate_new(size);  // Cache MISS
}
```

Логика MemoryPool::allocate()

```
storage_ptr_t MemoryPool::allocate(size_t size) {
  auto lower_it = cache_lookup_.lower_bound(size);
  // Search with tolerance
  if (lower_it != cache_lookup_.end() &&
      lower_it->first <= size +
                         static_cast<size_t>(size * size_diff_ratio_))) {
    return move_from_cache(lower_it);  // Cache HIT
  }

  if (buffers_cache_.size() >= max_size_) {
    evict_oldest();
  }
  return allocate_new(size);  // Cache MISS
}
```

Итоги раздела

Задачи, которые решает **MemoryPool**

01

Снижение накладных расходов выделения памяти

02

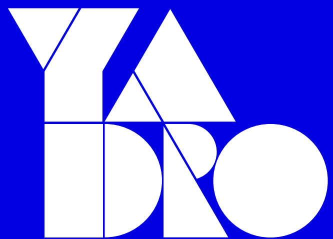
Автоматическое управление временем жизни буферов (RAII)

03

Балансировку hit rate через допуск по размерам

Вычислительные шейдеры

- Упрощение запуска
- Повторное использование
- Уменьшение количества системных вызовов



Запуск вычислительного ядра в CUDA — это просто!

```
int N = 1024; // Аргумент вызова

float *d_A, *d_B, *d_C; // Данные

// Размеры сетки
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Запуск
addVectors<<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C,
                                               N);
```

Запуск вычислительного ядра в CUDA — это просто!

```
int N = 1024; // Аргумент вызова

float *d_A, *d_B, *d_C; // Данные

// Размеры сетки
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Запуск
addVectors<<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C,
                                               N);
```

Запуск вычислительного ядра в CUDA — это просто!

```
int N = 1024; // Аргумент вызова

float *d_A, *d_B, *d_C; // Данные

// Размеры сетки
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Запуск
addVectors<<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C,
                                               N);
```

Запуск вычислительного ядра в CUDA — это просто!

```
int N = 1024; // Аргумент вызова

float *d_A, *d_B, *d_C; // Данные

// Размеры сетки
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Запуск
addVectors<<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C,
                                               N);
```

Запуск вычислительного ядра в CUDA — это просто!

```
int N = 1024; // Аргумент вызова

float *d_A, *d_B, *d_C; // Данные

// Размеры сетки
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Запуск
addVectors<<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C,
                                               N);
```

Запуск compute shader Vulkan — это сложно!

```
vkGetDeviceQueue(device, vulkanDevice->queueFamilyIndices.compute, 0, &compute.queue);

VkCommandPoolCreateInfo cmdPoolInfo = {};
cmdPoolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
cmdPoolInfo.queueFamilyIndex = vulkanDevice->queueFamilyIndices.compute;
cmdPoolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

VK_CHECK_RESULT(vkCreateCommandPool(device, &cmdPoolInfo, nullptr, &compute.commandPool));
VkCommandBufferAllocateInfo cmdBufAllocateInfo = vks::initializers::commandBufferAllocateInfo(compute.commandPool, VK_COMMAND_BUFFER_LEVEL_PRIMARY, 1);
for (auto& commandBuffer : compute.commandBuffers) {
  VK_CHECK_RESULT(vkAllocateCommandBuffers(device, &cmdBufAllocateInfo, &commandBuffer));
}
std::vector<VkDescriptorSetLayoutBinding> setLayoutBindings = {
  vks::initializers::descriptorSetLayoutBinding(VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_SHADER_STAGE_COMPUTE_BIT, 0),
  vks::initializers::descriptorSetLayoutBinding(VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_SHADER_STAGE_COMPUTE_BIT, 1),};

VkDescriptorSetLayoutCreateInfo descriptorLayout = vks::initializers::descriptorSetLayoutCreateInfo(setLayoutBindings);
VK_CHECK_RESULT(vkCreateDescriptorSetLayout(device, &descriptorLayout, nullptr, &compute.descriptorSetLayout));

VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo = vks::initializers::pipelineLayoutCreateInfo(&compute.descriptorSetLayout, 1);
VK_CHECK_RESULT(vkCreatePipelineLayout(device, &pipelineLayoutCreateInfo, nullptr, &compute.pipelineLayout));

VkDescriptorSetAllocateInfo allocInfo = vks::initializers::descriptorSetAllocateInfo(descriptorPool, &compute.descriptorSetLayout, 1);
VK_CHECK_RESULT(vkAllocateDescriptorSets(device, &allocInfo, &compute.descriptorSet));

std::vector<VkWriteDescriptorSet> computeWriteDescriptorSets = {
  vks::initializers::writeDescriptorSet(compute.descriptorSet, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 0, &textureColorMap.descriptor),
  vks::initializers::writeDescriptorSet(compute.descriptorSet, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 1, &storageImage.descriptor) };

vkUpdateDescriptorSets(device, static_cast<uint32_t>(computeWriteDescriptorSets.size()), computeWriteDescriptorSets.data(), 0, nullptr);
VkComputePipelineCreateInfo computePipelineCreateInfo = vks::initializers::computePipelineCreateInfo(compute.pipelineLayout, 0);

std::string fileName = getShadersPath() + "computeshader/" + shaderName + ".comp.spv";
computePipelineCreateInfo.stage = loadShader(fileName, VK_SHADER_STAGE_COMPUTE_BIT);
VkPipeline pipeline; VK_CHECK_RESULT(vkCreateComputePipelines(device, pipelineCache, 1, &computePipelineCreateInfo, nullptr, &pipeline));
compute.pipelines.push_back(pipeline);
```

Стоимость создания «конвейера» запуска шейдера

Для создания конвейера шейдера требуется создание дополнительных объектов

Shader Module	Descriptor Set Layout	SpecializationInfo
PushConstantRage	Pipeline Layout	Compute Layout

Проблема

Вычислительные шейдеры запускаются тысячи раз в секунду, а создание «конвейера» может занимать десятки миллисекунд

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C, // Memory pointers
                                               N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),   // spec constants
           make_shader_consts(N),                 // push constants
           blocksPerGrid);
```

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C, // Memory pointers
                                               N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),   // spec constants
           make_shader_consts(N),                 // push constants
           blocksPerGrid);
```

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C, // Memory pointers
                                               N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),   // spec constants
           make_shader_consts(N),                 // push constants
           blocksPerGrid);
```

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                                 d_A, d_B, d_C, // Memory pointers
                                                 N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),    // spec constants
           make_shader_consts(N),                  // push constants
           blocksPerGrid);
```

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                              d_A, d_B, d_C, // Memory pointers
                                              N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),   // spec constants
           make_shader_consts(N),                 // push constants
           blocksPerGrid);
```

Запуск вычислительного шейдера в Adept

CUDA

```
addVectors<dtype><<<blocksPerGrid, threadsPerBlock>>>(
                                               d_A, d_B, d_C, // Memory pointers
                                               N);            // Arguments
```

Adept

```
RUN_SHADER(device_id, dtype, ADD_SPV,
           buffers,
           make_shader_consts(threadsPerBlock),   // spec constants
           make_shader_consts(N),                 // push constants
           blocksPerGrid);
```

Push vs. Specialization константы

	Push-константы	Константы специализации
Аналогия CUDA	Аргументы вычислительного ядра	Конфигурация запуска
Когда задаются	Каждый вызов dispatch()	При создании конвейера
Производительность	Нулевые накладные расходы, быстрое обновление	Встроены в конвейер
Лимит размера	Макс. 128–256 байт	Ограничены размером SPIR-V модуля
Вариант использования	Форма тензора, счётчики, смещения	Тип данных, размер подгруппы

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Контейнер для констант

```
struct VkShaderConstants final {
  VkShaderConstantsDesc desc_; // sizes of each element
  std::vector<std::byte> data_;
  ...

  template <typename... Types>
  VkShaderConstants(Types&&... args) {
    desc_ = VkShaderConstantsDesc({sizeof(args)...});
    data_.resize(desc_.total_size());
    std::size_t offset = 0;
    (std::memcpy(data_.data() + (offset += sizeof(args)) - sizeof(args),
                 std::addressof(args), sizeof(args)),
     ...);
  }
  ...
};
```

Функции работы с константами

Разные типы и значения

```
template <typename... Args>
VkShaderConstants make_shader_consts(Args&&... args) {
  return VkShaderConstants(std::forward<Args>(args)...);
}
```

Значения одного типа

```
template <size_t N, typename T>
  requires(std::integral<T> || std::floating_point<T>)
VkShaderConstants make_uniform_shader_consts() {
  return VkShaderConstants(VkShaderConstantsDesc(N, sizeof(T)));
}
```

Функции работы с константами

Разные типы и значения

```
template <typename... Args>
VkShaderConstants make_shader_consts(Args&&... args) {
  return VkShaderConstants(std::forward<Args>(args)...);
}
```

Значения одного типа

```
template <size_t N, typename T>
  requires(std::integral<T> || std::floating_point<T>)
VkShaderConstants make_uniform_shader_consts() {
  return VkShaderConstants(VkShaderConstantsDesc(N, sizeof(T)));
}
```

Функции работы с константами

Разные типы и значения

```
template <typename... Args>
VkShaderConstants make_shader_consts(Args&&... args) {
  return VkShaderConstants(std::forward<Args>(args)...);
}
```

Значения одного типа

```
template <size_t N, typename T>
  requires(std::integral<T> || std::floating_point<T>)
VkShaderConstants make_uniform_shader_consts() {
  return VkShaderConstants(VkShaderConstantsDesc(N, sizeof(T)));
}
```

Инициализация констант значениями

```
const auto spec_consts = make_shader_consts(count, num_classes);
```

```
auto push_constants = make_uniform_shader_consts<14, uint32_t>();
push_constants.set(0, static_cast<uint32_t>(num_invokes));
push_constants.set(1, static_cast<uint32_t>(input_size[0]));
push_constants.set(2, static_cast<uint32_t>(input_size[1]));
push_constants.set(3, static_cast<uint32_t>(output_size[0]));
...
```

```
RUN_SHADER(device_id, dtype, shader,
           buffers,
           spec_constants,
           push_constants,
           work_group_size);
```

Инициализация констант значениями

```
const auto spec_consts = make_shader_consts(count, num_classes);
```

```
auto push_constants = make_uniform_shader_consts<14, uint32_t>();
push_constants.set(0, static_cast<uint32_t>(num_invokes));
push_constants.set(1, static_cast<uint32_t>(input_size[0]));
push_constants.set(2, static_cast<uint32_t>(input_size[1]));
push_constants.set(3, static_cast<uint32_t>(output_size[0]));
...
```

```
RUN_SHADER(device_id, dtype, shader,
           buffers,
           spec_constants,
           push_constants,
           work_group_size);
```

Инициализация констант значениями

```
const auto spec_consts = make_shader_consts(count, num_classes);
```

```
auto push_constants = make_uniform_shader_consts<14, uint32_t>();
push_constants.set(0, static_cast<uint32_t>(num_invokes));
push_constants.set(1, static_cast<uint32_t>(input_size[0]));
push_constants.set(2, static_cast<uint32_t>(input_size[1]));
push_constants.set(3, static_cast<uint32_t>(output_size[0]));
...
```

```
RUN_SHADER(device_id, dtype, shader,
           buffers,
           spec_constants,
           push_constants,
           work_group_size);
```

Инициализация констант значениями

```
const auto spec_consts = make_shader_consts(count, num_classes);
```

```
auto push_constants = make_uniform_shader_consts<14, uint32_t>();
push_constants.set(0, static_cast<uint32_t>(num_invokes));
push_constants.set(1, static_cast<uint32_t>(input_size[0]));
push_constants.set(2, static_cast<uint32_t>(input_size[1]));
push_constants.set(3, static_cast<uint32_t>(output_size[0]));
...
```

```
RUN_SHADER(device_id, dtype, shader,
           buffers,
           spec_constants,
           push_constants,
           work_group_size);
```

Новое описание констант = Новый конвейер шейдера

```
void VkShader::rebuild_pipeline(const VkShaderConstants& spec_consts) {
  // ...
  // Build specialization entries: index → offset → size
  std::vector<vk::SpecializationMapEntry> entries;
  // for (...) {
     entries.emplace_back(i, offset, spec_consts.item_size(i));
  // }...

  vk::SpecializationInfo spec_info(entries.size(), entries.data(),
                                   spec_consts.total_size(), spec_consts.data());

  compute_pipeline_ = device_.createComputePipeline(pipeline_cache_,
                                                    { ..., shader_module_, &spec_info, ... });
}
```

Новое описание констант =
Новый конвейер шейдера

```
void VkShader::rebuild_pipeline(const VkShaderConstants& spec_consts) {
  // ...
  // Build specialization entries: index → offset → size
  std::vector<vk::SpecializationMapEntry> entries;
  // for (...) {
     entries.emplace_back(i, offset, spec_consts.item_size(i));
  // }...

  vk::SpecializationInfo spec_info(entries.size(), entries.data(),
                                   spec_consts.total_size(), spec_consts.data());

  compute_pipeline_ = device_.createComputePipeline(pipeline_cache_,
                                             { ..., shader_module_, &spec_info, ... });
}
```

Новое описание констант = Новый конвейер шейдера

```
void VkShader::rebuild_pipeline(const VkShaderConstants& spec_consts) {
  // ...
  // Build specialization entries: index → offset → size
  std::vector<vk::SpecializationMapEntry> entries;
  // for (...) {
     entries.emplace_back(i, offset, spec_consts.item_size(i));
  // }...

  vk::SpecializationInfo spec_info(entries.size(), entries.data(),
                                   spec_consts.total_size(), spec_consts.data());

  compute_pipeline_ = device_.createComputePipeline(pipeline_cache_,
                                                    { ..., shader_module_, &spec_info, ... });
}
```

Новое описание констант = Новый конвейер шейдера

```
void VkShader::rebuild_pipeline(const VkShaderConstants& spec_consts) {
  // ...
  // Build specialization entries: index → offset → size
  std::vector<vk::SpecializationMapEntry> entries;
  // for (...) {
     entries.emplace_back(i, offset, spec_consts.item_size(i));
  // }...

  vk::SpecializationInfo spec_info(entries.size(), entries.data(),
                                   spec_consts.total_size(), spec_consts.data());

  compute_pipeline_ = device_.createComputePipeline(pipeline_cache_,
                                                     { ..., shader_module_, &spec_info, ... });
}
```

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;           // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;           // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;          // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;           // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;           // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Кэш шейдеров — повторно используем объекты

```
struct VkShaderCacheItem {
  index_t device_id;            // поддержка нескольких GPU
  ShaderSourcePtr source_ptr; // бинарник SPIR-V
  std::shared_ptr<const VkShaderConstants> spec_consts;
  std::weak_ptr<VkShader> shader;

  bool operator==(const VkShaderCacheItem& other) const;
};

template<> struct std::hash<VkShaderCacheItem> {
  size_t operator()(const VkShaderCacheItem& key) const;
};
```

Хэш = объединение всех трёх полей для уникальной идентификации

Макросы для запуска шейдеров

1. Создание или получение шейдера из кэша

```
#define RUN_SHADER(device_id, dtype, shader, buffers, consts, ...) \
  auto shader_wptr = VkManager::instance().make_shader(...); \
  EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...);
```

2. Отправка шейдера в очередь на запуск

```
#define EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...) \
  VkManager::instance().launch(device_id, shader_wptr, buffers, ...);
```

Макросы для запуска шейдеров

1. Создание или получение шейдера из кэша

```
#define RUN_SHADER(device_id, dtype, shader, buffers, consts, ...) \
  auto shader_wptr = VkManager::instance().make_shader(...); \
  EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...);
```

2. Отправка шейдера в очередь на запуск

```
#define EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...) \
  VkManager::instance().launch(device_id, shader_wptr, buffers, ...);
```

Макросы для запуска шейдеров

1. Создание или получение шейдера из кэша

```
#define RUN_SHADER(device_id, dtype, shader, buffers, consts, ...) \
  auto shader_wptr = VkManager::instance().make_shader(...); \
  EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...);
```

2. Отправка шейдера в очередь на запуск

```
#define EXEC_SHADER(device_id, shader_wptr, buffers, consts, ...) \
  VkManager::instance().launch(device_id, shader_wptr, buffers, ...);
```

Пример операции сложения тензоров

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Пример операции сложения тензоров

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

Пример операции сложения тензоров

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...
  auto buffers = make_transitions(
    self.buffer_for_write(),
    other.buffer_for_read()
  );
  auto push_constants = make_arithmetics_push_consts();
  auto spec_constants = make_shader_consts(num_invokes);
  // ...
  RUN_SHADER(device_id, dtype, shader,
             buffers,
             spec_constants, push_constants,
             ceil_div(count, num_invokes));   // workgroup size
}
```

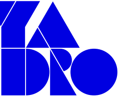
Последовательность запуска, спрятанная в макросе

User	Tensor	Manager	Device	Shader	GPU

- User → Tensor: tensor.add(other)
- Tensor → Manager: RUN_SHADER1 macro
- Manager → Manager: make-shader() cache lookup
- Manager → Device: launch()
- Device → Device: record barriers
- Device → Shader: bind buffers/pipeline
- Device → GPU: dispatch + submit

Async execution

Подготовка и отправка шейдера в очередь запуска



```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Подготовка и отправка шейдера в очередь запуска

```
void VkDevice::launch(VkShader& shader,
                      const BufferTransitions& buffers,
                      const VkShaderConstants& push_consts,
                      uint32_t group_count_x, ...) {
  start_record();

  record_buffer_memory_barriers(buffers);
  shader.bind_buffers(cmd_buf, desc_pool, ...);
  shader.bind_pipeline(cmd_buf);
  shader.push_constants(cmd_buf, push_consts);
  cmd_buf.dispatch(group_count_x, ...); // только записывает вызов

  end_record(/*sync=*/false); // тут может быть отправка в GPU
}
```

Накладные расходов при запуске шейдера

Объект	Стоимость создания	Частота
Буфер команд	Средняя	На каждую отправку
Набор дескрипторов	Средняя	На каждый запуск шейдера
Буфер памяти	Высокая	На каждый тензор
Конвейер	Очень высокая	На каждую комбинацию шейдер+спецификация

Решение: пулы объектов, пакетирование и кэширование

Накладные расходов при запуске шейдера

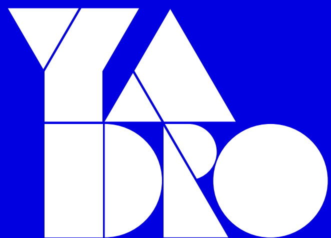
Объект	Стоимость создания	Частота
Буфер команд	**Средняя**	**На каждую отправку**
Набор дескрипторов	**Средняя**	**На каждый запуск шейдера**
Буфер памяти	Высокая	На каждый тензор
Конвейер	Очень высокая	На каждую комбинацию шейдер+спецификация

Решение: пулы объектов, пакетирование и кэширование

Пакетирование + Пулы объектов

- Буферы команд: выделение 1024 штук за раз
- Наборы дескрипторов: пул с обработкой переполнения

Амортизация стоимости выделения при множественном использовании



Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул буферов команд

Пакетное выделение пула команд амортизирует накладные расходы

```
class VkCommandPool {
private:
  std::vector<vk::CommandBuffer> buffers_;
  size_t current_buffer_index_{0};

  // ...
public:
  vk::CommandBuffer get_command_buffer() {
    // vkAllocateCommandBuffers
    allocate_new_batch(VULKAN_COMMAND_BUFFER_BATCH_SIZE);
    auto handle = buffers_[current_buffer_index_];
    ++current_buffer_index_;
    return handle;
  }
};
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                      vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());         // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                       vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());         // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                       vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());        // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                       vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());         // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                       vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());        // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Пул дескрипторов

```
vk::DescriptorSet VkDescriptorPool::allocate_descriptor_set(
                                       vk::DescriptorSetLayout layout) {
  auto descriptor_pool = pools_[current_pool_index_];
  vk::DescriptorSetAllocateInfo alloc_info(descriptor_pool, layout);
  try {
    auto sets = device_.allocateDescriptorSets(alloc_info);
    return sets.front();
  } catch (const vk::OutOfPoolMemoryError&) {
    ++current_pool_index_;
    if (current_pool_index_ < pools_.size()) {
      return allocate_descriptor_set(layout);  // Retry existing pools
    }
    pools_.emplace_back(create_pool());        // Overflow: new pool
    return allocate_descriptor_set(layout);    // Retry new pool
  }
}
```

Итоги раздела

01

Создание шейдер pipeline — дорогая операция

02

Вычислительные нагрузки требуют большого количества запусков в секунду

03

Кэширование амортизирует стоимость запуска

04

Большинство операций повторяются с различиями только в динамических параметрах

05

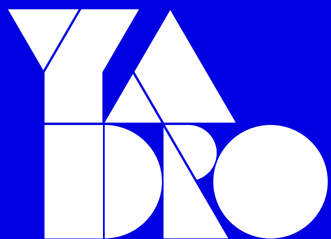
Идентификация шейдеров по device_id, SPIR-V source, specialization constants

06

Шейдеры компилируются заранее

Синхронизация доступа к данным

- Отслеживание доступов
- Автоматическая генерация барьеров
- Уменьшение влияния ручного контроля



Проблема видимости изменений в памяти

CUDA: автоматически внутри потока

Vulkan: требуются явные барьеры

Execution

Visibility

Kernel Run Completes → Memory Flush Required → Sync Point Makes Data Visible

Сложность ручных барьеров

```cpp
void manual_barrier(vk::CommandBuffer& cmd_buf,
                    vk::AccessFlags src_access,
                    vk::AccessFlags dst_access,
                    vk::PipelineStageFlags src_stage,
                    vk::PipelineStageFlags dst_stage,
                    vk::QueueFamilyIndices src_queue,
                    vk::QueueFamilyIndices dst_queue) {

  vk::BufferMemoryBarrier barrier;
  barrier.srcAccessMask = src_access;         // Must know previous access
  barrier.dstAccessMask = dst_access;         // Must know next access
  barrier.srcQueueFamilyIndex = src_queue;  // Must track queue ownership
  barrier.dstQueueFamilyIndex = dst_queue;  // Must transfer ownership
  barrier.buffer = buffer;

  cmd_buf.pipelineBarrier(src_stage, dst_stage, ..., barrier);
}
```

Сложность ручных барьеров

```
void manual_barrier(vk::CommandBuffer& cmd_buf,
                    vk::AccessFlags src_access,
                    vk::AccessFlags dst_access,
                    vk::PipelineStageFlags src_stage,
                    vk::PipelineStageFlags dst_stage,
                    vk::QueueFamilyIndices src_queue,
                    vk::QueueFamilyIndices dst_queue) {

  vk::BufferMemoryBarrier barrier;
  barrier.srcAccessMask = src_access;       // Must know previous access
  barrier.dstAccessMask = dst_access;       // Must know next access
  barrier.srcQueueFamilyIndex = src_queue;  // Must track queue ownership
  barrier.dstQueueFamilyIndex = dst_queue;  // Must transfer ownership
  barrier.buffer = buffer;

  cmd_buf.pipelineBarrier(src_stage, dst_stage, ..., barrier);
}
```

Сложность ручных барьеров

```
void manual_barrier(vk::CommandBuffer& cmd_buf,
                    vk::AccessFlags src_access,
                    vk::AccessFlags dst_access,
                    vk::PipelineStageFlags src_stage,
                    vk::PipelineStageFlags dst_stage,
                    vk::QueueFamilyIndices src_queue,
                    vk::QueueFamilyIndices dst_queue) {

  vk::BufferMemoryBarrier barrier;
  barrier.srcAccessMask = src_access;           // Must know previous access
  barrier.dstAccessMask = dst_access;           // Must know next access
  barrier.srcQueueFamilyIndex = src_queue;      // Must track queue ownership
  barrier.dstQueueFamilyIndex = dst_queue;      // Must transfer ownership
  barrier.buffer = buffer;

  cmd_buf.pipelineBarrier(src_stage, dst_stage, ..., barrier);
}
```

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- **Сравнение** с предыдущей информацией о доступе
- **Генерация барьера** при необходимости
- **Обновление состояния** для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- **Сравнение** с предыдущей информацией о доступе
- **Генерация барьера** при необходимости
- **Обновление состояния** для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- **Сравнение** с предыдущей информацией о доступе
- **Генерация барьера** при необходимости
- **Обновление состояния** для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- **Сравнение с предыдущей информацией о доступе**
- **Генерация барьера** при необходимости
- **Обновление состояния** для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- Сравнение с предыдущей информацией о доступе
- Генерация барьера при необходимости
- **Обновление состояния** для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Решение: автоматическое отслеживание доступа

История последней операции сохраняется в буфере

```
last_access_type_
last_pipeline_stage_
```

При каждом доступе к данным происходит

- Сравнение с предыдущей информацией о доступе
- Генерация барьера при необходимости
- Обновление состояния для отслеживания

Преимущество в том, что разработчику не нужно вручную вставлять барьеры

Типы доступа к буферу данных

Тип доступа	Флаг Vulkan	Стадия конвейера
Compute Read	eShaderRead	eComputeShader
Compute Write	eShaderWrite	eComputeShader
Transfer Read	eTransferRead	eTransfer
Transfer Write	eTransferWrite	eTransfer
Host Read	eHostRead	eHost
Host Write	eHostWrite	eHost

Варианты переходов при доступе к буферу

Compute Write

Host Read

Computer Read

Transfer Read

Host Write

Transfer Write

Структура для описания истории доступа

```
struct VkBufferTransition {
  storage_ptr_t parent_storage_;          // Keep buffer alive
  const vk::DescriptorBufferInfo* info;
  vk::AccessFlags src_access_type;        // source access type
  vk::AccessFlags dst_access_type;        // destination access type
  vk::PipelineStageFlags src_stage;       // source pipeline
  vk::PipelineStageFlags dst_stage;       // destination pipeline

  bool is_readonly() const;
};
```

Структура для описания истории доступа

```
struct VkBufferTransition {
  storage_ptr_t parent_storage_;          // Keep buffer alive
  const vk::DescriptorBufferInfo* info;
  vk::AccessFlags src_access_type;        // source access type
  vk::AccessFlags dst_access_type;        // destination access type
  vk::PipelineStageFlags src_stage;       // source pipeline
  vk::PipelineStageFlags dst_stage;       // destination pipeline

  bool is_readonly() const;
};
```

Структура для описания истории доступа

```
struct VkBufferTransition {
  storage_ptr_t parent_storage_;         // Keep buffer alive
  const vk::DescriptorBufferInfo* info;
  vk::AccessFlags src_access_type;       // source access type
  vk::AccessFlags dst_access_type;       // destination access type
  vk::PipelineStageFlags src_stage;      // source pipeline
  vk::PipelineStageFlags dst_stage;      // destination pipeline

  bool is_readonly() const;
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                               vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;           // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;               // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                // src → dst access
        prev_stage, dst_stage                      // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;              // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;                  // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                   // src → dst access
        prev_stage, dst_stage                         // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;           // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;               // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                // src → dst access
        prev_stage, dst_stage                      // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;           // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;               // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                // src → dst access
        prev_stage, dst_stage                      // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;            // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;                // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                 // src → dst access
        prev_stage, dst_stage                       // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;           // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;               // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                // src → dst access
        prev_stage, dst_stage                      // src → dst stage
    };
  }
};
```

Отслеживание доступа к буферу

```
class VkBufferBase {
  mutable vk::AccessFlags last_access_type_{eShaderRead};
  mutable vk::PipelineStageFlags last_pipline_stage_{eComputeShader};

  VkBufferTransition buffer_for(vk::AccessFlags dst_access_type,
                                vk::PipelineStageFlags dst_stage) const {
    auto prev_last = last_access_type_;
    last_access_type_ = dst_access_type;              // Update tracking state

    auto prev_stage = last_pipline_stage_;
    last_pipline_stage_ = dst_stage;                  // Update tracking state

    return VkBufferTransition{
        nullptr, descriptor_buffer_info(),
        prev_last, dst_access_type,                   // src → dst access
        prev_stage, dst_stage                         // src → dst stage
    };
  }
};
```

Методы для описания доступа к буферу

```
VkBufferTransition buffer_for_compute_read() const {
  return buffer_for(eShaderRead, eComputeShader);
}

VkBufferTransition buffer_for_compute_write() const {
  return buffer_for(eShaderWrite, eComputeShader);
}

VkBufferTransition buffer_for_transfer_read() const {
  return buffer_for(eTransferRead, eTransfer);
}

VkBufferTransition buffer_for_host_read() const {
  return buffer_for(eHostRead, eHost);
}
```

Методы для описания доступа к буферу

```
VkBufferTransition buffer_for_compute_read() const {
  return buffer_for(eShaderRead, eComputeShader);
}

VkBufferTransition buffer_for_compute_write() const {
  return buffer_for(eShaderWrite, eComputeShader);
}

VkBufferTransition buffer_for_transfer_read() const {
  return buffer_for(eTransferRead, eTransfer);
}

VkBufferTransition buffer_for_host_read() const {
  return buffer_for(eHostRead, eHost);
}
```

Методы для описания доступа к буферу

```
VkBufferTransition buffer_for_compute_read() const {
  return buffer_for(eShaderRead, eComputeShader);
}

VkBufferTransition buffer_for_compute_write() const {
  return buffer_for(eShaderWrite, eComputeShader);
}

VkBufferTransition buffer_for_transfer_read() const {
  return buffer_for(eTransferRead, eTransfer);
}

VkBufferTransition buffer_for_host_read() const {
  return buffer_for(eHostRead, eHost);
}
```

Методы для описания доступа к буферу

```
VkBufferTransition buffer_for_compute_read() const {
  return buffer_for(eShaderRead, eComputeShader);
}

VkBufferTransition buffer_for_compute_write() const {
  return buffer_for(eShaderWrite, eComputeShader);
}

VkBufferTransition buffer_for_transfer_read() const {
  return buffer_for(eTransferRead, eTransfer);
}

VkBufferTransition buffer_for_host_read() const {
  return buffer_for(eHostRead, eHost);
}
```

Использование методов отслеживания доступа

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...

  auto buffers = make_transitions(
    self.buffer_for_write(), // compute pipeline
    other.buffer_for_read()
  );

  // RUN_SHADER( ...
    record_buffer_memory_barriers(buffers);
    shader.bind_buffers(cmd_buf, desc_pool, buffers, ...);
  // ... );
}
```

Использование методов отслеживания доступа

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...

  auto buffers = make_transitions(
    self.buffer_for_write(), // compute pipeline
    other.buffer_for_read()
  );

  // RUN_SHADER( ...
    record_buffer_memory_barriers(buffers);
    shader.bind_buffers(cmd_buf, desc_pool, buffers, ...);
  // ... );
}
```

Использование методов отслеживания доступа

```
void TensorImpl::add(const DeviceTensor& other) {
  // ...

  auto buffers = make_transitions(
    self.buffer_for_write(), // compute pipeline
    other.buffer_for_read()
  );

  // RUN_SHADER( ...
    record_buffer_memory_barriers(buffers);
    shader.bind_buffers(cmd_buf, desc_pool, buffers, ...);
  // ... );
}
```

Создание общего барьера для операции

```
void VkDevice::record_buffer_memory_barriers(
                                    const BufferTransitions& buffer_ars) {
  std::vector<vk::BufferMemoryBarrier> barriers;
  vk::PipelineStageFlags src_stages = vk::PipelineStageFlagBits::eNone;
  vk::PipelineStageFlags dst_stages = vk::PipelineStageFlagBits::eNone;

  // Aggregate barriers
  // ...

  if (!barriers.empty()) {
    current_submit_->command_buffer().pipelineBarrier(
                     src_stages, dst_stages,
                     vk::DependencyFlags(), nullptr, barriers, nullptr);
  }
}
```

Создание общего барьера для операции

```
void VkDevice::record_buffer_memory_barriers(
                                     const BufferTransitions& buffer_ars) {
  std::vector<vk::BufferMemoryBarrier> barriers;
  vk::PipelineStageFlags src_stages = vk::PipelineStageFlagBits::eNone;
  vk::PipelineStageFlags dst_stages = vk::PipelineStageFlagBits::eNone;

  // Aggregate barriers
  // ...

  if (!barriers.empty()) {
    current_submit_->command_buffer().pipelineBarrier(
                     src_stages, dst_stages,
                     vk::DependencyFlags(), nullptr, barriers, nullptr);
  }
}
```

Создание общего барьера для операции

```
void VkDevice::record_buffer_memory_barriers(
                                    const BufferTransitions& buffer_ars) {
  std::vector<vk::BufferMemoryBarrier> barriers;
  vk::PipelineStageFlags src_stages = vk::PipelineStageFlagBits::eNone;
  vk::PipelineStageFlags dst_stages = vk::PipelineStageFlagBits::eNone;

  // Aggregate barriers
  // ...

  if (!barriers.empty()) {
    current_submit_->command_buffer().pipelineBarrier(
                       src_stages, dst_stages,
                       vk::DependencyFlags(), nullptr, barriers, nullptr);
  }
}
```

Создание общего барьера для операции

```
void VkDevice::record_buffer_memory_barriers(
                                    const BufferTransitions& buffer_ars) {
  std::vector<vk::BufferMemoryBarrier> barriers;
  vk::PipelineStageFlags src_stages = vk::PipelineStageFlagBits::eNone;
  vk::PipelineStageFlags dst_stages = vk::PipelineStageFlagBits::eNone;

  // Aggregate barriers
  // ...

  if (!barriers.empty()) {
    current_submit_->command_buffer().pipelineBarrier(
                       src_stages, dst_stages,
                       vk::DependencyFlags(), nullptr, barriers, nullptr);
  }
}
```

Создание общего барьера для операции

```
void VkDevice::record_buffer_memory_barriers(
                                    const BufferTransitions& buffer_ars) {
  std::vector<vk::BufferMemoryBarrier> barriers;
  vk::PipelineStageFlags src_stages = vk::PipelineStageFlagBits::eNone;
  vk::PipelineStageFlags dst_stages = vk::PipelineStageFlagBits::eNone;

  // Aggregate barriers
  // ...

  if (!barriers.empty()) {
    current_submit_->command_buffer().pipelineBarrier(
                       src_stages, dst_stages,
                       vk::DependencyFlags(), nullptr, barriers, nullptr);
  }
}
```

Агрегация барьеров

```
// ...
  for (auto& arg : buffer_ars) {
    current_submit_->add_storage(arg.parent_storage_);

    if (!arg.is_readonly()) {
      barriers.emplace_back(arg.src_access_type, arg.dst_access_type,
                            VK_QUEUE_FAMILY_IGNORED,
                            VK_QUEUE_FAMILY_IGNORED,
                            arg.info->buffer, 0, arg.info->range,
                            nullptr);
      src_stages |= arg.src_stage;
      dst_stages |= arg.dst_stage;
    }
  }
// ...
```

Агрегация барьеров

```
// ...
  for (auto& arg : buffer_ars) {
    current_submit_->add_storage(arg.parent_storage_);

    if (!arg.is_readonly()) {
      barriers.emplace_back(arg.src_access_type, arg.dst_access_type,
                            VK_QUEUE_FAMILY_IGNORED,
                            VK_QUEUE_FAMILY_IGNORED,
                            arg.info->buffer, 0, arg.info->range,
                            nullptr);
      src_stages |= arg.src_stage;
      dst_stages |= arg.dst_stage;
    }
  }
// ...
```

Агрегация барьеров

```
// ...
  for (auto& arg : buffer_ars) {
    current_submit_->add_storage(arg.parent_storage_);

    if (!arg.is_readonly()) {
      barriers.emplace_back(arg.src_access_type, arg.dst_access_type,
                            VK_QUEUE_FAMILY_IGNORED,
                            VK_QUEUE_FAMILY_IGNORED,
                            arg.info->buffer, 0, arg.info->range,
                            nullptr);
      src_stages |= arg.src_stage;
      dst_stages |= arg.dst_stage;
    }
  }
// ...
```

Итоги раздела

01

Отслеживание доступа — решает проблему корректного создания барьеров

02

Автоматическая генерация барьеров устраняет целый класс ошибок синхронизации

03

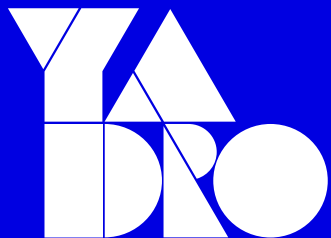
Оптимизации (read→read skip, агрегация) дают измеримый прирост производительности

04

Разработчик работает с простым API, не думая о низкоуровневой синхронизации

Жизненный цикл ресурсов

- Блокировка ресурсов во время вычислений
- Освобождение и возврат ресурсов
- Балансировка нагрузки на конвейер



Проблема жизненного цикла ресурсов

GPU Side

Shader 1 – Executing...

Barrier: Data dependency

Shader 2 – Executing...

Resource Pool

Data Buffers

Command Buffers

- Во время выполнения ресурсы, связанные с вычислениями, нельзя возвращать в пул до окончания работы.
- Отслеживать каждый ресурс — дорого.

VkSubmit — пакет команд для отправки в GPU

Инкапсулирует всё необходимое для одной отправки на GPU

VkSubmit

Descriptor Pool → binds → Command Buffer → records → GPU Work

Timeline Semaphore → signals → signal_value_ → tracks → Completion Status

Концепция отслеживания submits

stopped_submits_ – Пул готовых

Submit готов

1

Запись – current_submit_

start_record

Запись команд
Барьеры, Bind, Dispatch

2

active_submits_ – Выполнение

end_record

GPU выполняет

Timeline Semaphore

false

3

Completion – завершение

is_completed?

4

Логика семафора в VkSubmit

```
class VkSubmit {
  uint64_t signal_value_;
  vk::UniqueSemaphore complete_semaphore_;
  // ...

  void reset() {
    if (complete_semaphore_) {
      ++signal_value_;  // Increment counter for next submission
    } else {
      complete_semaphore_ = create_sumbit_semaphore();
    }
  }

  bool is_completed() const {
    return device_.getSemaphoreCounterValue(...) == signal_value_;
  }
};
```

Логика семафора в VkSubmit

```
class VkSubmit {
  uint64_t signal_value_;
  vk::UniqueSemaphore complete_semaphore_;
  // ...

  void reset() {
    if (complete_semaphore_) {
      ++signal_value_;  // Increment counter for next submission
    } else {
      complete_semaphore_ = create_sumbit_semaphore();
    }
  }

  bool is_completed() const {
    return device_.getSemaphoreCounterValue(...) == signal_value_;
  }
};
```

Логика семафора в VkSubmit

```
class VkSubmit {
  uint64_t signal_value_;
  vk::UniqueSemaphore complete_semaphore_;
  // ...

  void reset() {
    if (complete_semaphore_) {
      ++signal_value_;  // Increment counter for next submission
    } else {
      complete_semaphore_ = create_sumbit_semaphore();
    }
  }

  bool is_completed() const {
    return device_.getSemaphoreCounterValue(...) == signal_value_;
  }
};
```

Логика семафора в VkSubmit

```
class VkSubmit {
  uint64_t signal_value_;
  vk::UniqueSemaphore complete_semaphore_;
  // ...

  void reset() {
    if (complete_semaphore_) {
      ++signal_value_;  // Increment counter for next submission
    } else {
      complete_semaphore_ = create_sumbit_semaphore();
    }
  }

  bool is_completed() const {
    return device_.getSemaphoreCounterValue(...) == signal_value_;
  }
};
```

Логика семафора в VkSubmit

```
class VkSubmit {
  uint64_t signal_value_;
  vk::UniqueSemaphore complete_semaphore_;
  // ...

  void reset() {
    if (complete_semaphore_) {
      ++signal_value_;  // Increment counter for next submission
    } else {
      complete_semaphore_ = create_sumbit_semaphore();
    }
  }

  bool is_completed() const {
    return device_.getSemaphoreCounterValue(...) == signal_value_;
  }
};
```

Семафоры как метки для отслеживания

Host | GPU | Timeline Semaphore

Counter = 0

Host → GPU: Submit Work A

GPU → GPU: Execute Work A

GPU → Timeline Semaphore: Signal (value 1)

Counter = 1

Host → Host: Poll: getSemaphoreCounterValue()

Returns 1 = Complete

Время жизни хранилища/буфера данных

```
struct VkBufferTransition {
  storage_ptr_t parent_storage_;  // Holds ref until GPU done
  // ...
};
```

```
// In record_buffer_memory_barriers(): Increments refcount
current_submit_->add_storage(arg.parent_storage_);
```

Поиск доступного пакета — VkSubmit

```
void VkDevice::start_record() {
  if (!current_submit_) {
    if (!stopped_submits_.empty()) {
      current_submit_ = std::move(stopped_submits_.front());
      stopped_submits_.pop_front();
    } else {
      current_submit_ = std::make_unique<VkSubmit>(...);
    }
    current_submit_->start();
  }
}
```

Поиск доступного пакета — VkSubmit

```
void VkDevice::start_record() {
  if (!current_submit_) {
    if (!stopped_submits_.empty()) {
      current_submit_ = std::move(stopped_submits_.front());
      stopped_submits_.pop_front();
    } else {
      current_submit_ = std::make_unique<VkSubmit>(...);
    }
    current_submit_->start();
  }
}
```

Поиск доступного пакета — VkSubmit

```
void VkDevice::start_record() {
  if (!current_submit_) {
    if (!stopped_submits_.empty()) {
      current_submit_ = std::move(stopped_submits_.front());
      stopped_submits_.pop_front();
    } else {
      current_submit_ = std::make_unique<VkSubmit>(...);
    }
    current_submit_->start();
  }
}
```

Поиск доступного пакета — VkSubmit

```
void VkDevice::start_record() {
  if (!current_submit_) {
    if (!stopped_submits_.empty()) {
      current_submit_ = std::move(stopped_submits_.front());
      stopped_submits_.pop_front();
    } else {
      current_submit_ = std::make_unique<VkSubmit>(...);
    }
    current_submit_->start();
  }
}
```

Пакетная запись команд

VULKAN_MAX_RECORD_COUNT = 16

Запись 16 команд → единая отправка

Вместо: 16 отдельных отправок

Преимущество: меньше вызовов vkQueueSubmit, эффективнее на CPU

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Отправка пакета на выполнение

```
void VkDevice::end_record(bool sync) {
  records_count_++;
  if (records_count_ >= VULKAN_MAX_RECORD_COUNT || sync) {
    current_submit_->end();

    if (sync) {
      // Sync: fence + CPU wait
      reset_fence();
      queue_.submit({submit_info}, global_fence_);
      wait_fence();
    } else {
      // Async: timeline semaphore for completion tracking
      submit_info.pSignalSemaphores = &current_submit_->semaphore();
      queue_.submit({submit_info}, VK_NULL_HANDLE);
      active_submits_.emplace_back(std::move(current_submit_));
    }
     // ... in-flight limiting and recycling ...
    records_count_ = 0;
  }
}
```

Синхронная vs. Асинхронная отправка

Путь	Синхронизация	Вариант использования
Sync (sync=true)	Fence + ожидание CPU	Пересылки хост-устройство
Async (sync=false)	НЕТ (Семафор для ресурсов)	Запуски вычислительных ядер

Ограничение находящихся в процессе отправок

Баланс: утилизация GPU vs. давление на память

```
if (active_submits_.size() > VULKAN_MAX_SUBMITS_COUNT) {
  active_submits_.back()->wait();
}
```

Полинг завершенных отправок

```
for (auto it = active_submits_.begin(); it != active_submits_.end();) {
  if ((*it)->is_completed()) {
    stopped_submits_.splice(..., active_submits_, it);
    break;
  }
}
```

Конвейер асинхронных отправок

Итоги раздела

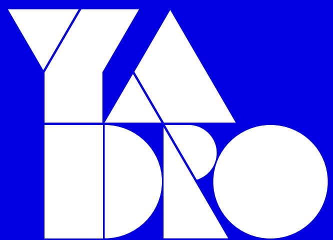
Использование семафора позволило реализовать:

- Отслеживание завершения вычислений
- Корректное повторное использование ресурсов

Ограничение количества текущих отправок позволило:

- Балансировать утилизацию GPU и памяти

Сравнение производительности



Описание тестовой системы

Intel Core i9-13900HX

- 8 P-ядер
- 16 E-ядер
- 800 МГц - 5.4 ГГц

GeForce RTX 4070 8ГБ:

- 46 SMs
- 184 Tensor Cores
- L1 Cache 128 KB (per SM)
- L2 Cache 36 MB

RAM 32 ГБ

Обучение многослойного перцептрона (MatMul)

MLP:784x1014x512x256x10

Пакетов в секунду

600
500
400
300
200
100
0

32 64 128 256 512 1024 2048 4096

Размер пакета

- Adept
- PyTroch cuBLAS(:16:8)
- PyTroch cuBLAS
- PyTroch cuDNN

Обучение свёрточной сети LeNet5(Conv2d – im2col)

Пакетов в секунду

250
200
150
100
50
0

32
64
128
256
512
1024

Размер пакета

■ Adept
■ PyTroch cuBLAS(:16:8)
■ PyTroch cuBLAS

Обучение свёрточной сети LeNet5 с cuDNN

Пакетов в секунду

600
500
400
300
200
100
0

32
64
128
256
512
1024

Размер пакета

Adept
PyTroch cuBLAS(:16:8)
PyTroch cuBLAS
PyTroch cuDNN

Общие итоги

Конвейер вычислений про производительности сравним с профессиональными решениями

Что повлияло на производительность?

Пакетирование команд для отправки	Создание барьеров для доступа данных для каждого тензора
Асинхронная отправка вычислений	Использование пулов объектов(буферы команды, дескрипторы, буферы данных)
Явная синхронизация только для чтения данных на хоста	Кеширование шейдеров

Спасибо за внимание!

Остаемся на связи