# Многопоточность в OpenCL: Работа с контекстами, очередями и данными

## Краткий ответ

**Нельзя напрямую использовать объекты памяти из разных контекстов** — это фундаментальное ограничение OpenCL. Данные нужно явно копировать между контекстами через хост. Внутри одного контекста **можно безопасно делить все объекты между потоками**, но с важными оговорками по поводу ядер и синхронизации.

## Правила thread safety в OpenCL

### Что безопасно делать из разных потоков

OpenCL API спроектирован с учётом многопоточности:[1] [2]

| Операция | Thread-safe | Ограничения |
|---|---|---|
| `clCreateBuffer`, `clCreateImage` | ✅ Да | Нет |
| `clEnqueueRead/WriteBuffer` | ✅ Да | Нет |
| `clEnqueueNDRangeKernel` | ✅ Да | Нет |
| `clSetKernelArg` | ⚠️ Частично | **Только на разных cl_kernel объектах**[3] |
| `clFinish`, `clFlush` | ✅ Да | Нет |
| `clRelease*/clRetain*` | ✅ Да | Нет |

**Критическое ограничение**: Несколько потоков **не могут безопасно** вызывать `clSetKernelArg` на одном и том же объекте `cl_kernel`. Поведение неопределено и приведёт к гонкам данных.[3]

### Что нельзя делать между контекстами

**Объекты памяти (cl_mem) привязаны к контексту и не могут быть использованы в другом контексте**. Это архитектурное ограничение OpenCL:[4]

```
// ❌ НЕВОЗМОЖНО: Использование буфера из контекста A в очереди контекста B
cl_context ctxA = clCreateContext(...);
cl_context ctxB = clCreateContext(...);
cl_mem bufferA = clCreateBuffer(ctxA, ...);
```

```
cl_command_queue queueB = clCreateCommandQueue(ctxB, ...);
clEnqueueNDRangeKernel(queueB, kernel, ..., &bufferA); // ОШИБКА!
```

### Работа с одним контекстом из множества потоков

### Стратегия 1: Один контекст, множество очередей, по ядру на поток

**Рекомендуемый подход** для высокопроизводительных приложений:[5]

```
class ThreadSafeOpenCLManager {
private:
    cl::Context context;
    cl::Device device;
    std::vector<cl::CommandQueue> computeQueues;
    std::vector<cl::CommandQueue> transferQueues;

public:
    ThreadSafeOpenCLManager() {
        // Инициализация в главном потоке
        auto platforms = cl::Platform::get();
        platforms[^0].getDevices(CL_DEVICE_TYPE_GPU, &device);
        context = cl::Context(device);

        // Создаём очереди для каждого потока
        size_t numThreads = std::thread::hardware_concurrency();
        for (size_t i = 0; i < numThreads; ++i) {
            computeQueues.emplace_back(context, device,
                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
            transferQueues.emplace_back(context, device);
        }
    }

    // Получить очереди для текущего потока (по ID)
    cl::CommandQueue& getComputeQueue(size_t threadId) {
        return computeQueues[threadId % computeQueues.size()];
    }

    cl::CommandQueue& getTransferQueue(size_t threadId) {
        return transferQueues[threadId % transferQueues.size()];
    }

    cl::Context& getContext() { return context; }
    cl::Device& getDevice() { return device; }
};

// Использование в worker потоках
void workerThread(ThreadSafeOpenCLManager& manager,
                  size_t threadId, cl::Kernel kernelTemplate) {
    // Создаём СВОЙ экземпляр ядра для потокобезопасности
    cl::Kernel kernel = kernelTemplate; // Копия ядра

    auto& computeQueue = manager.getComputeQueue(threadId);
    auto& transferQueue = manager.getTransferQueue(threadId);
    auto& context = manager.getContext();
```

```
    // Создаём буферы (привязаны к контексту, не к потоку)
    cl::Buffer buffer(context, CL_MEM_READ_WRITE, size);

    // Потокобезопасно: каждый поток работает со своим ядром
    kernel.setArg(0, buffer);
    kernel.setArg(1, param);

    // Перекрытие передачи и вычислений
    transferQueue.enqueueWriteBuffer(buffer, CL_FALSE, ...);
    computeQueue.enqueueNDRangeKernel(kernel, ...);
    transferQueue.enqueueReadBuffer(buffer, CL_FALSE, ...);

    computeQueue.finish();
}
```

**Преимущества**:

- ✅ Полная потокобезопасность (разные ядра)
- ✅ Перекрытие передачи данных и вычислений
- ✅ Оптимальное использование GPU[6]

### Стратегия 2: Пул ядер с thread-local аргументами

Если создание ядер дорого, можно использовать пул:

```
class KernelPool {
private:
    std::mutex mutex;
    std::vector<cl::Kernel> availableKernels;
    cl::Program program;
    const char* kernelName;

public:
    cl::Kernel acquireKernel() {
        std::lock_guard<std::mutex> lock(mutex);
        if (!availableKernels.empty()) {
            cl::Kernel kernel = availableKernels.back();
            availableKernels.pop_back();
            return kernel;
        }
        // Создаём новое ядро, если пул пуст
        return cl::Kernel(program, kernelName);
    }

    void releaseKernel(cl::Kernel kernel) {
        std::lock_guard<std::mutex> lock(mutex);
        availableKernels.push_back(kernel);
    }
};

// Использование
void workerThread(KernelPool& pool, ThreadSafeOpenCLManager& manager, size_t threadId) {
```

```
    cl::Kernel kernel = pool.acquireKernel();

    // Установка аргументов безопасна, т.к. ядро не делится
    kernel.setArg(0, buffer);
    kernel.setArg(1, value);

    manager.getComputeQueue(threadId).enqueueNDRangeKernel(kernel, ...);

    pool.releaseKernel(kernel); // Возвращаем в пул
}
```

### Работа с разными контекстами (редкий случай)

#### Когда нужны разные контексты

- Разные платформы (AMD GPU + Intel CPU)
- Изолированные вычислительные домены
- Безопасность и изоляция ресурсов

#### Копирование данных между контекстами

**Единственный способ** — через хост-память:[4]

```
// Контекст A
cl_context ctxA = ...;
cl_mem bufferA = clCreateBuffer(ctxA, CL_MEM_READ_WRITE, size, NULL, NULL);

// Контекст B
cl_context ctxB = ...;
cl_mem bufferB = clCreateBuffer(ctxB, CL_MEM_READ_WRITE, size, NULL, NULL);

// Временный хост-буфер
void* hostBuffer = malloc(size);

// Читаем из A
clEnqueueReadBuffer(queueA, bufferA, CL_TRUE, 0, size, hostBuffer, 0, NULL, NULL);

// Пишем в B
clEnqueueWriteBuffer(queueB, bufferB, CL_TRUE, 0, size, hostBuffer, 0, NULL, NULL);

free(hostBuffer);
```

**Производительность**: Эта операция **в 2 раза медленнее**, чем обычный transfer, т.к. требует:

1. GPU → Host (чтение)
2. Host → GPU (запись)

### Альтернатива: OpenGL sharing

Если оба контекста созданы из одного OpenGL контекста с `CL_GL_CONTEXT_KHR`, можно делить GL-буферы:[4]

```
cl_context_properties props[] = {
    CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),
    CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
    0
};

cl_context ctxA = clCreateContext(props, deviceA, NULL, NULL, NULL);
cl_context ctxB = clCreateContext(props, deviceB, NULL, NULL, NULL);

// Оба контекста могут делить GL-буферы
cl_mem glBufferA = clCreateFromGLBuffer(ctxA, CL_MEM_READ_WRITE, glBuffer, NULL);
cl_mem glBufferB = clCreateFromGLBuffer(ctxB, CL_MEM_READ_WRITE, glBuffer, NULL);
```

### Синхронизация между потоками

### События OpenCL vs примитивы ОС

OpenCL events **не синхронизируют потоки ОС**, только очереди команд:[7]

```
// ✗ НЕПРАВИЛЬНО: Ожидание в одном потоке, запуск в другом
cl_event event;
queueA.enqueueNDRangeKernel(kernelA, ..., &event);

// В другом потоке
queueB.enqueueNDRangeKernel(kernelB, 1, &event); // Ждёт событие OpenCL

// НО: потоки ОС всё ещё работают параллельно!
```

### Правильная синхронизация потоков

```
std::mutex mtx;
std::condition_variable cv;
bool kernelA_done = false;

// Поток A
void threadA(cl::CommandQueue& queue, cl::Kernel& kernel) {
    queue.enqueueNDRangeKernel(kernel);
    queue.finish(); // Гарантируем выполнение

    {
        std::lock_guard<std::mutex> lock(mtx);
        kernelA_done = true;
    }
    cv.notify_all(); // Уведомляем поток B
}
```

```
// Поток B
void threadB(cl::CommandQueue& queue, cl::Kernel& kernel) {
    {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []{ return kernelA_done; });
    }

    // Теперь можно использовать результаты kernelA
    queue.enqueueNDRangeKernel(kernel);
}
```

## Производительность и лучшие практики

### Накладные расходы на операции

| Операция | Примерное время | Потокобезопасность |
|---|---|---|
| `clEnqueueNDRangeKernel` | 5-50 мкс | ✅ Да |
| `clSetKernelArg` | 1-5 мкс | ⚠ Только разные ядра |
| `clEnqueueRead/WriteBuffer` | 10-1000 мкс | ✅ Да |
| **Копирование между контекстами** | **2x медленнее** | ✅ Да |

### Рекомендации по количеству

⁂


1. https://community.khronos.org/t/share-context-between-different-host-threads/6962
2. https://stackoverflow.com/questions/65462444/how-to-pass-the-opencl-context-between-different-cpu-pthreads
3. https://stackoverflow.com/questions/63212484/opencl-race-condition-with-clsetkernelarg
4. https://stackoverflow.com/questions/12782134/share-opencl-memory-objects-between-different-contexts
5. https://www.reddit.com/r/OpenCL/comments/vsfhwu/threadsafety_of_operations_with_kernel/
6. https://stackoverflow.com/questions/63768538/opencl-multiple-threads-on-a-gpu
7. https://cgmb-rocm-docs.readthedocs.io/en/latest/Programming_Guides/Opencl-programming-guide.html
8. https://clehaxze.tw/gemlog/2022/06-23-understanding-explicit-opencl-memory-migration-between-devices.gmi
9. https://community.khronos.org/t/multiple-host-threads-with-single-command-queue-and-device/2209
10. https://www.intel.com/content/www/us/en/docs/opencl-sdk/developer-reference-cpu-runtime/2018/intel-immediate-command-execution.html
11. https://community.khronos.org/t/shared-opencl-buffers-between-processes/2281
12. https://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf

13. https://stackoverflow.com/questions/43925804/does-opencl-have-shared-context-feature-between-opencl-contexts
14. https://www.aronaldg.org/webfiles/compecon/src/opencl/doc/OpenCL_tutorial.pdf
15. https://github.com/opencv/opencv/issues/20577