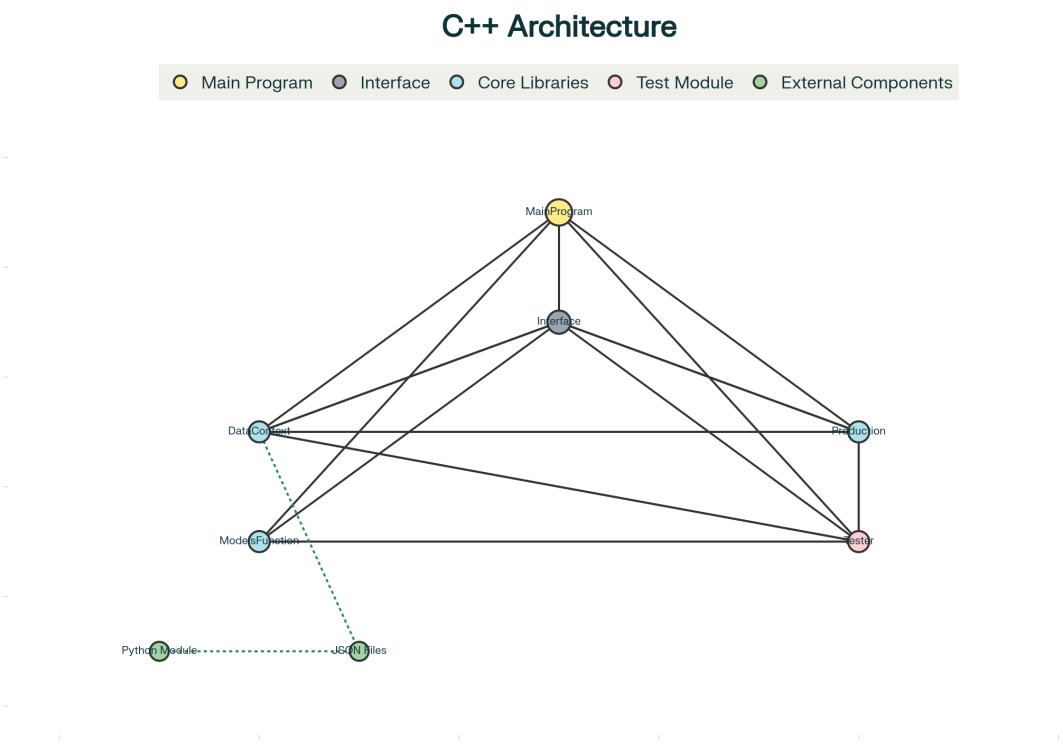
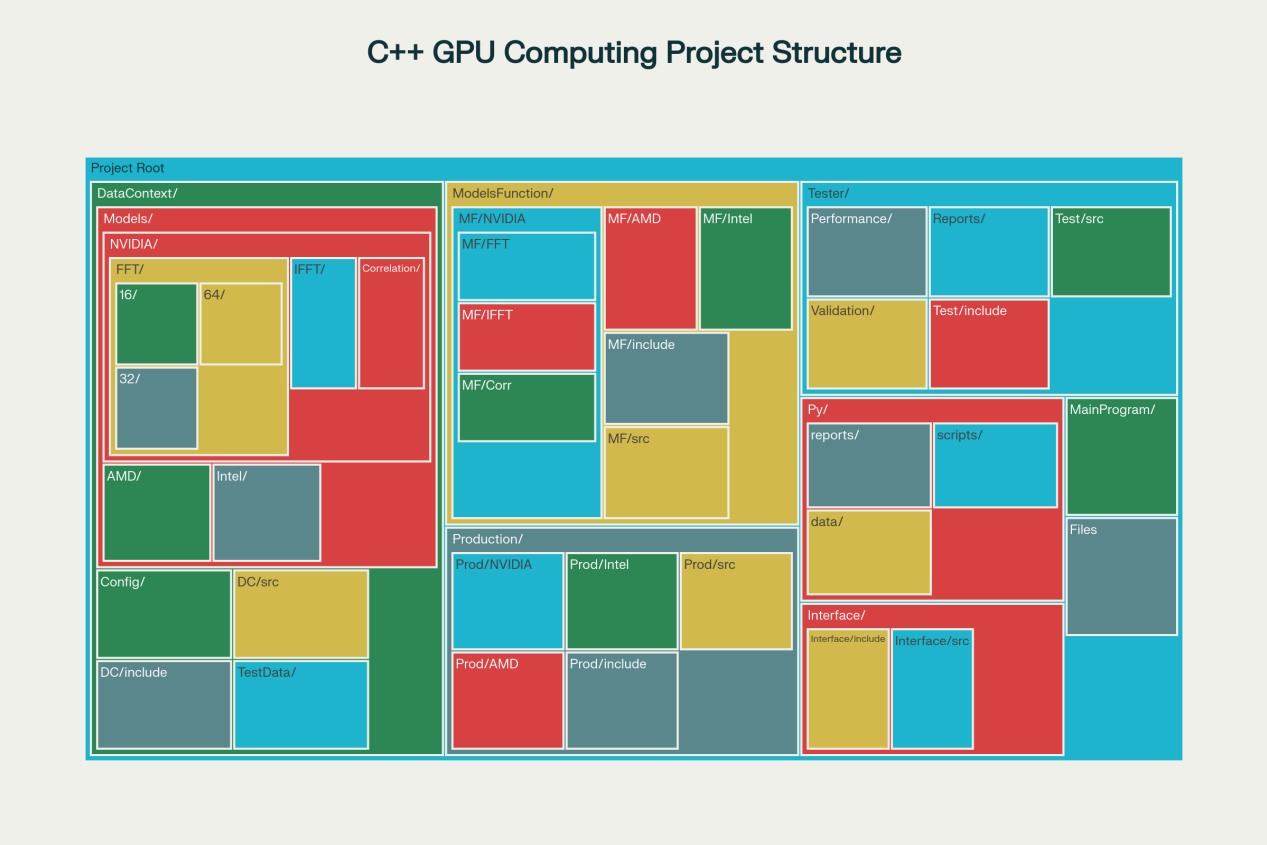
Архитектура проекта GPU-вычислений на C++

Архитектурная схема проекта



Архитектура проекта GPU-вычислений на C++ с модульной структурой

Структура каталогов проекта



Структура каталогов проекта GPU-вычислений на C++

Описание модульной архитектуры на C++

Interface - Центральный интерфейсный слой

Ключевой модуль, обеспечивающий унификацию взаимодействия между всеми компонентами системы через заголовочные файлы и абстрактные классы.

Основные структуры данных:

```
// include/signal_data.h
struct InputSignalData {
    std::complex<float>* vector_ptr;    // Указатель на входной вектор
    int data_count;                     // Количество элементов вектора
    int signal_count;                   // Количество сигналов (многолучевой осциллограф)
    int window_fft;                     // Размер окна FFT
};

// include/spectral_data.h
struct OutputSpectralData {
    std::vector<std::vector<std::complex<float>>> spectral_vectors;
};

// include/igpu_processor.h
class IGPUProcessor {
public:
    virtual ~IGPUProcessor() = default;
    virtual OutputSpectralData process(const InputSignalData& input) = 0;
```

```
    virtual bool initialize() = 0;
    virtual void cleanup() = 0;
};
```

DataContext - Управление данными

Центральный узел для обработки, хранения и передачи данных между модулями системы с использованием современных C++ возможностей.

Структура модуля:

```
DataContext/
├── include/
│   ├── data_manager.h
│   ├── config_parser.h
│   ├── signal_generator.h
│   └── version_control.h
├── src/
│   ├── data_manager.cpp
│   ├── config_parser.cpp
│   ├── signal_generator.cpp
│   └── version_control.cpp
├── Config/
│   ├── gpu_configs.json
│   ├── test_scenarios.json
│   └── validation_settings.json
└── Models/                    // История и архив моделей
    ├── NVIDIA/
    │   ├── FFT/
    │   │   ├── 16/
    │   │   │   └── model_2025_10_09_v1/
    │   │   │       ├── fft16_optimized.cu
    │   │   │       ├── fft16_optimized.cpp
    │   │   │       ├── test_description.txt
    │   │   │       ├── performance_results.json
    │   │   │       └── validation_log.txt
    │   │   ├── 32/
    │   │   ├── 64/
    │   │   └── 128/
    │   ├── IFFT/
    │   ├── Correlation/
    │   └── Convolution/
    ├── AMD/
    │   ├── FFT/
    │   ├── IFFT/
    │   └── Correlation/
    └── Intel/
        ├── FFT/
        └── IFFT/
```

Ключевые возможности:

```
// include/data_manager.h
class DataManager {
private:
    std::unique_ptr<ConfigParser> config_parser_;
    std::unique_ptr<VersionControl> version_control_;

public:
    bool load_config(const std::string& config_path);
    std::vector<InputSignalData> generate_test_signals();
    void archive_model(const std::string& gpu_type,
                       const std::string& algorithm,
                       const ModelInfo& model_info);
    void log_operation(const std::string& operation,
                       const std::string& details);
};
```

Production - Продакшен-готовые решения

Библиотека протестированных и оптимизированных математических примитивов с использованием современных C++ стандартов.

Структура модуля:

```
Production/
├── include/
│   ├── nvidia/
│   │   ├── fft_production.h
│   │   ├── ifft_production.h
│   │   └── correlation_production.h
│   ├── amd/
│   │   ├── fft_rocm.h
│   │   └── correlation_opencl.h
│   └── intel/
│       ├── fft_oneapi.h
│       └── ifft_sycl.h
├── src/
│   ├── nvidia/
│   │   ├── fft_production.cpp
│   │   ├── fft_kernels.cu
│   │   ├── ifft_production.cpp
│   │   └── correlation_production.cpp
│   ├── amd/
│   └── intel/
└── CMakeLists.txt
```

Пример реализации:

```
// include/nvidia/fft_production.h
namespace Production::NVIDIA {
    class FFT16Production : public IGPUProcessor {
    private:
        cufftHandle plan_;
```

```
        float* d_input_;
        cufftComplex* d_output_;

    public:
        FFT16Production();
        ~FFT16Production();

        bool initialize() override;
        OutputSpectralData process(const InputSignalData& input) override;
        void cleanup() override;

        // Производительные методы
        void process_batch(const std::vector<InputSignalData>& batch,
                           std::vector<OutputSpectralData>& results);
    };
}
```

ModelsFunction - Экспериментальные модели

Модуль для разработки, тестирования и отладки новых моделей с использованием шаблонов C++ и наследования.

Структура организации:

```
ModelsFunction/
├── include/
│   ├── nvidia/
│   │   ├── fft/
│   │   │   ├── fft16_basic.h
│   │   │   ├── fft16_hamming.h
│   │   │   ├── fft32_basic.h
│   │   │   └── fft32_windowed.h
│   │   ├── ifft/
│   │   └── correlation/
│   ├── amd/
│   └── intel/
├── src/
│   ├── nvidia/
│   │   ├── fft/
│   │   │   ├── FFT16_Basic/
│   │   │   │   ├── fft16_basic.cpp
│   │   │   │   ├── fft16_kernel.cu
│   │   │   │   └── CMakeLists.txt
│   │   │   ├── FFT16_Hamming/
│   │   │   │   ├── fft16_hamming.cpp      // Наследует от FFT16Basic
│   │   │   │   ├── fft16_hamming.cu
│   │   │   │   ├── hamming_window.cpp
│   │   │   │   └── CMakeLists.txt
│   │   │   ├── FFT32_Basic/
│   │   │   └── FFT32_Windowed/
│   │   ├── ifft/
│   │   └── correlation/
│   ├── amd/
```

```
│   └── intel/
└── CMakeLists.txt
```

Принцип версионирования с наследованием:

```
// include/nvidia/fft/fft16_basic.h
namespace ModelsFunction::NVIDIA {
    class FFT16Basic : public IGPUProcessor {
    protected:
        cufftHandle plan_;
        virtual void prepare_input(const InputSignalData& input);

    public:
        FFT16Basic();
        virtual ~FFT16Basic();

        bool initialize() override;
        virtual OutputSpectralData process(const InputSignalData& input) override;
        void cleanup() override;
    };
}

// include/nvidia/fft/fft16_hamming.h
namespace ModelsFunction::NVIDIA {
    class FFT16Hamming : public FFT16Basic {
    private:
        std::vector<float> hamming_window_;
        bool is_hamming_enabled_;

        void apply_hamming_window(InputSignalData& input);

    public:
        FFT16Hamming();
        ~FFT16Hamming() = default;

        OutputSpectralData process(const InputSignalData& input) override;
        void enable_hamming(bool enable) { is_hamming_enabled_ = enable; }
        bool is_hamming() const { return is_hamming_enabled_; }
    };
}
```

Tester - Система тестирования и валидации

Комплексный модуль профилирования и валидации с использованием современных C++ библиотек.

Структура модуля:

```
Tester/
├── include/
│   ├── performance/
│   │   ├── gpu_profiler.h
│   │   ├── memory_profiler.h
```

```
│   │   └── benchmark_runner.h
│   ├── validation/
│   │   ├── base_validator.h
│   │   ├── fft_validator.h
│   │   ├── correlation_validator.h
│   │   └── comparison_engine.h
│   └── test_suites/
│       ├── fft_test_suite.h
│       └── integration_tests.h
├── src/
│   ├── performance/
│   ├── validation/
│   └── test_suites/
├── Performance/
├── Validation/
├── Reports/
└── CMakeLists.txt
```

Реализация тестирования:

```
// include/validation/base_validator.h
class BaseValidator {
protected:
    double tolerance_;
    std::ofstream log_file_;

public:
    BaseValidator(double tolerance = 1e-6);
    virtual ~BaseValidator() = default;

    virtual bool validate(const OutputSpectralData& result,
                          const OutputSpectralData& reference) = 0;
    virtual void log_result(const std::string& test_name, bool passed);
};

// include/validation/fft_validator.h
class FFTValidator : public BaseValidator {
public:
    FFTValidator();

    bool validate(const OutputSpectralData& result,
                  const OutputSpectralData& reference) override;

    // Специализированные методы для FFT
    bool validate_parseval_theorem(const InputSignalData& input,
                                   const OutputSpectralData& output);
    bool validate_linearity(const InputSignalData& input1,
                            const InputSignalData& input2);
};
```

MainProgram - Основное приложение

Центральная точка входа с современной архитектурой C++.

Структура:

```
MainProgram/
├── include/
│   ├── program.h
│   ├── workflow_manager.h
│   ├── configuration_manager.h
│   └── command_processor.h
├── src/
│   ├── main.cpp
│   ├── workflow_manager.cpp
│   ├── configuration_manager.cpp
│   └── command_processor.cpp
└── CMakeLists.txt
```

Основной workflow:

```
// src/main.cpp
#include <memory>
#include <iostream>
#include "workflow_manager.h"

int main(int argc, char* argv[]) {
    try {
        auto workflow = std::make_unique<WorkflowManager>();

        if (!workflow->initialize(argc, argv)) {
            std::cerr << "Failed to initialize workflow" << std::endl;
            return -1;
        }

        return workflow->run();

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return -1;
    }
}
```

CMake конфигурация

Корневой CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.18)
project(GPUComputingProject LANGUAGES CXX CUDA)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
# CUDA support
find_package(CUDA REQUIRED)
enable_language(CUDA)

# OpenCL support
find_package(OpenCL REQUIRED)

# cuFFT
find_package(CUDAToolkit REQUIRED)

# JSON support
find_package(nlohmann_json REQUIRED)

# Добавляем подпроекты
add_subdirectory(Interface)
add_subdirectory(DataContext)
add_subdirectory(Production)
add_subdirectory(ModelsFunction)
add_subdirectory(Tester)
add_subdirectory(MainProgram)

# Глобальные настройки компилятора
set(CMAKE_CUDA_SEPARABLE_COMPILATION ON)
set(CMAKE_CUDA_ARCHITECTURES "60;70;75;80;86")
```

Пример CMakeLists.txt для ModelsFunction:

```
# ModelsFunction/CMakeLists.txt
add_library(ModelsFunction STATIC)

# Исходные файлы
target_sources(ModelsFunction
    PRIVATE
        src/nvidia/fft/FFT16_Basic/fft16_basic.cpp
        src/nvidia/fft/FFT16_Basic/fft16_kernel.cu
        src/nvidia/fft/FFT16_Hamming/fft16_hamming.cpp
        src/nvidia/fft/FFT16_Hamming/fft16_hamming.cu
        src/nvidia/fft/FFT16_Hamming/hamming_window.cpp
)

# Заголовочные файлы
target_include_directories(ModelsFunction
    PUBLIC
        include/
    PRIVATE
        src/
)

# Зависимости
target_link_libraries(ModelsFunction
    PUBLIC
        Interface
        CUDA::cufft
        CUDA::cudart
```

```
    PRIVATE
        DataContext
)

# CUDA настройки
set_target_properties(ModelsFunction PROPERTIES
    CUDA_SEPARABLE_COMPILATION ON
    CUDA_ARCHITECTURES "60;70;75;80;86"
)
```

Особенности C++ реализации

Управление памятью

- Использование smart pointers (`std::unique_ptr`, `std::shared_ptr`)
- RAII для управления GPU ресурсами
- Автоматическая очистка CUDA контекстов

Современный C++

- Шаблоны для универсальности алгоритмов
- Move семантика для оптимизации производительности
- Constexpr для compile-time вычислений
- Structured bindings для удобства работы с данными

Интеграция с Python

```
// include/python_interface.h
class PythonInterface {
private:
    std::string data_directory_;
    std::string reports_directory_;

public:
    void export_results_json(const std::string& filename,
                             const TestResults& results);
    void trigger_report_generation(const std::string& report_type);
    std::vector<std::string> get_available_reports();
};
```

Данная архитектура на C++ обеспечивает высокую производительность, типобезопасность и эффективное управление ресурсами GPU при сохранении гибкости и расширяемости проекта.