Part 5. Comprehensive Projects: Practical CUDA Development

Real-world Project Implementation

CNN and Image Processing Pipeline

프로젝트 기반 학습의 중요성

실무에서 CUDA를 사용할 때는 단순한 커널 하나만 작성하는 것이 아닙니다. 복잡한 시스템을 설계하고, 다양한 최적화 기법을 조합하며, 유지보수 가능한 코드를 작성해야 합니다.

실무 CUDA 프로젝트의 특징

1. **복합적 문제 해결**
    - 여러 알고리즘의 조합
    - 메모리 관리와 성능 최적화
    - 에러 처리와 디버깅
2. **시스템 통합**
    - 호스트-디바이스 상호작용
    - 외부 라이브러리 활용
    - 실시간 처리 요구사항
3. **확장성과 유지보수성**
    - 모듈화된 설계
    - 코드 재사용성
    - 성능 모니터링
4. **실제 데이터와 제약조건**
    - 대용량 데이터 처리
    - 메모리 제한
    - 실시간 처리 요구

표준 프로젝트 구조

```
cuda-project/
├── include/
│   ├── kernels.cuh         # CUDA kernel declarations
│   ├── tensor.h            # Tensor class definition
│   └── utils.h             # Utility functions
├── src/
│   ├── kernels/
│   │   ├── conv2d.cu       # Convolution kernels
│   │   ├── pooling.cu      # Pooling operations
│   │   └── activation.cu   # Activation functions
│   ├── host/
│   │   ├── main.cpp        # Main program
│   │   └── model.cpp       # Model implementation
│   └── utils/
│       └── memory.cpp      # Memory management
├── tests/
│   ├── test_kernels.cpp    # Unit tests
│   └── benchmark.cpp       # Performance tests
├── build/                  # Build output
├── CMakeLists.txt
└── README.md
```

필수 개발 도구와 설정

CMake 설정 예시

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

# CUDA 설정
set(CMAKE_CUDA_STANDARD 17)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CUDA_ARCHITECTURES "75;80;86;89")

# 컴파일 옵션
set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -O3 -use_fast_math")
```

핵심 유틸리티 헤더

```
// src/utils/cuda_utils.h
#define CUDA_CHECK(call) do { /* ... */ } while(0)
#define CUDA_CHECK_KERNEL() do { /* ... */ } while(0)
void printGPUInfo() {  /* ... */  }
void checkMemoryUsage() {  /* ... */  }
int getOptimalBlockSize(const void* kernel_func, int dynamic_smem_size = 0);
```

데이터 관리 및 검증 시스템

테스트 데이터 생성기

```
class DataGenerator {
private:
    std::mt19937 rng;
public:
    DataGenerator(uint32_t seed = 42) : rng(seed) {}
    void generateRandomFloats(float* data, size_t count, float min_val = -1.0f, float max_val = 1.0f);
    void generateSequentialFloats(float* data, size_t count, float start = 0.0f);
    void generateImage(float* data, int width, int height, int channels);
    void generateSparseMatrix(float* data, int rows, int cols, float sparsity = 0.9f);
};
```

결과 검증 클래스

```
class ResultValidator {
public:
    static bool compareFloats(float a, float b, float epsilon = 1e-5f);
    static bool compareArrays(const float* gpu_result, const float* cpu_result,
                size_t count, float epsilon = 1e-5f);
    static void compareStatistics(const float* gpu_result, const float* cpu_result,
                  size_t count);
};
```

성능 벤치마킹 시스템

```
class PerformanceBenchmark {
private:
    struct BenchmarkResult {
        std::string name;
        double time_ms;
        size_t data_size;
        size_t operations;
        size_t memory_used;
    };
    std::vector<BenchmarkResult> results;

public:
    void addResult(const std::string& name, double time_ms,
                   size_t data_size, size_t operations, size_t memory_used);
    void printResults();
    void saveToFile(const std::string& filename);
    void compareResults(const std::string& baseline);
};
```

자동화된 벤치마킹 함수

```
template<typename Kernel>
void benchmarkKernel(const std::string& name, Kernel kernel_func,
        void** args, size_t data_size, size_t operations,
        dim3 grid, dim3 block, size_t shared_mem = 0) {
    const int num_runs = 100;
    const int warmup_runs = 10;
    std::vector<float> times;

    // Warm-up runs
    for (int i = 0; i < warmup_runs; i++) {
        kernel_func<<<grid, block, shared_mem>>>(args[0], args[1], args[2]);
    }
    CUDA_CHECK(cudaDeviceSynchronize());

    // Actual benchmark runs
    CUDATimer timer;
    for (int i = 0; i < num_runs; i++) {
        timer.start();
        kernel_func<<<grid, block, shared_mem>>>(args[0], args[1], args[2]);
```

핵심: 체계적인 벤치마킹 시스템은 성능 병목을 식별하고 최적화 효과를 정량적으로 측정하는 데 필수적입니다.

5.2 프로젝트 1: CNN 계산 가속기

프로젝트 개요

딥러닝의 핵심인 Convolutional Neural Network(CNN)를 처음부터 CUDA로 구현합니다.

학습 목표

- **텐서 연산 최적화**: 다차원 배열의 효율적인 처리
- **메모리 계층 활용**: Shared Memory를 통한 컨볼루션 가속
- **배치 처리**: 여러 이미지 동시 처리로 처리량 극대화
- **성능 비교**: cuDNN 대비 우리 구현의 성능 분석

구현할 레이어

- Convolution 2D (with im2col, Winograd)
- Batch Normalization
- Pooling (Max, Average)
- Activation Functions (ReLU, Sigmoid, Tanh)
- Fully Connected

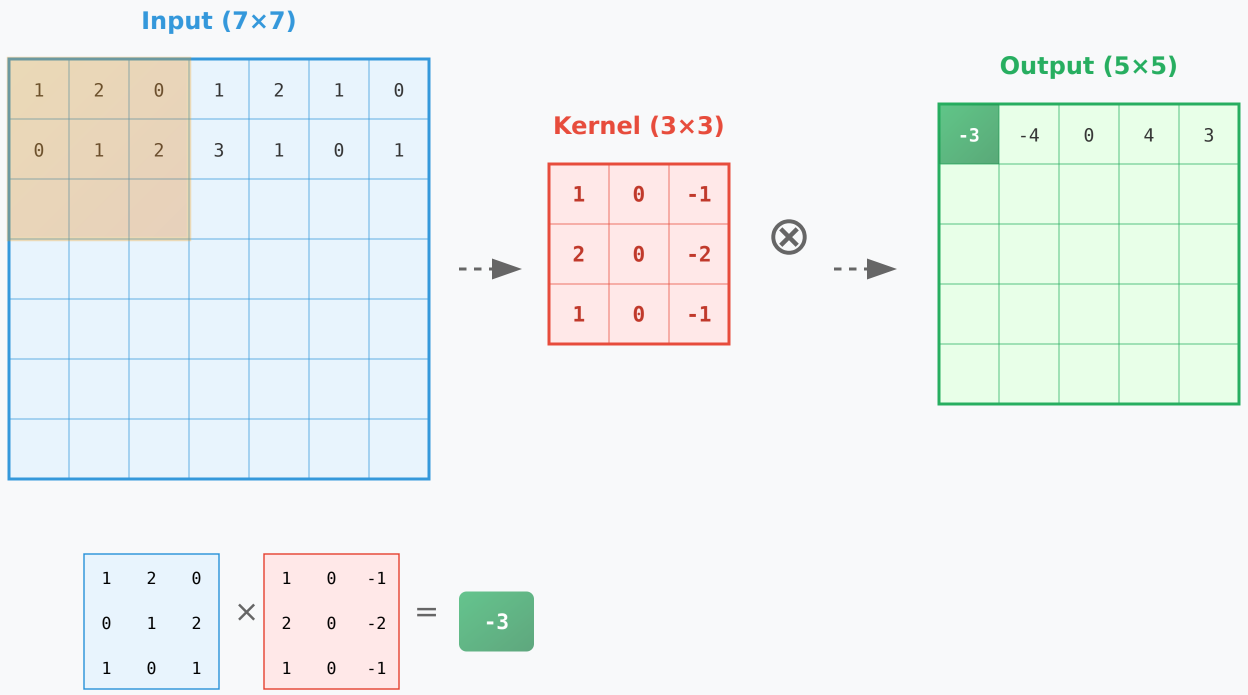
Project Implementation

Practice files: cnn_project/

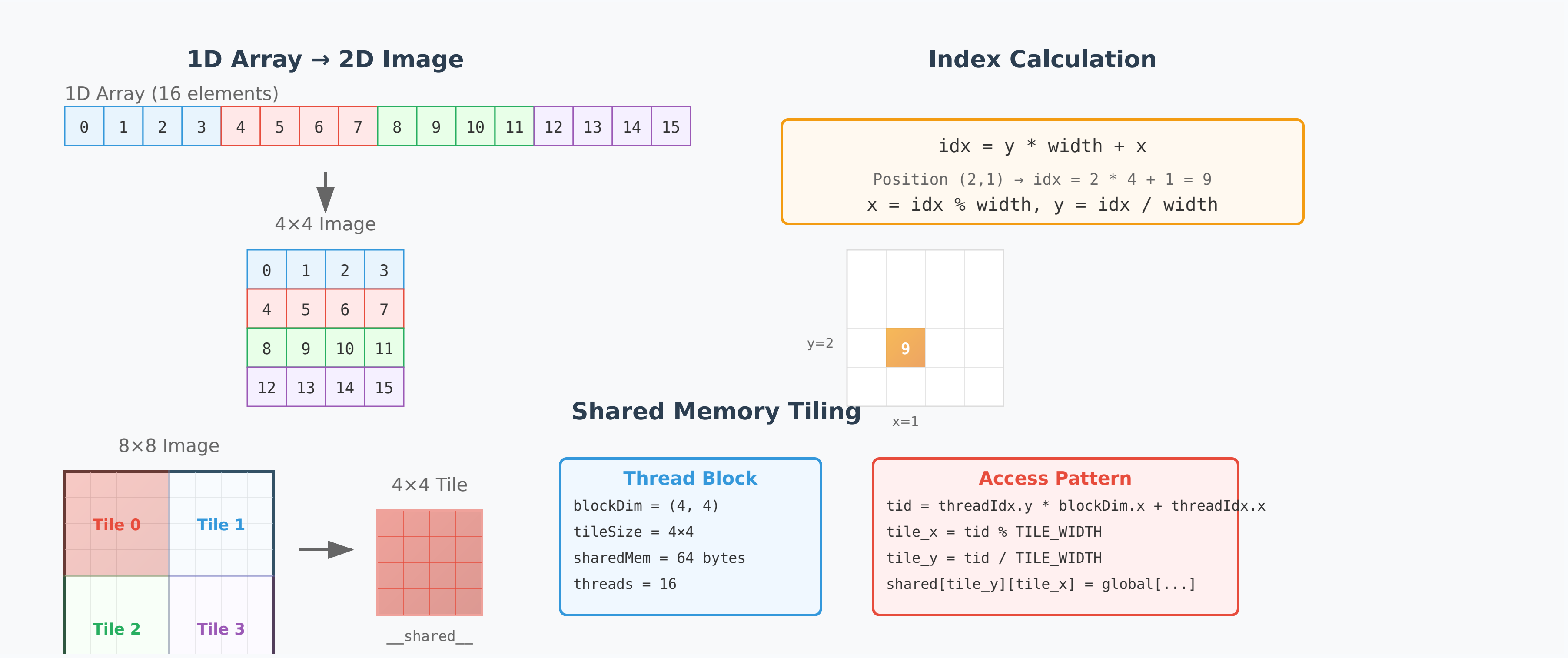
Project Structure

```
cnn_project/
├── include/
│   ├── tensor.h
│   ├── layers.h
│   └── network.h
├── src/
│   ├── kernels/
│   │   ├── conv2d.cu
│   │   ├── pooling.cu
│   │   └── activation.cu
│   └── main.cu
└── Makefile
```

Convolution Operation Visualization



Tiling Visualization



Key Components

1. **Tensor Class**: Multi-dimensional array management
2. **Layer Interface**: Forward/Backward propagation
3. **Optimized Kernels**:
   - 2D Convolution
   - Max Pooling
   - ReLU Activation

```
// Optimized convolution using shared memory
__global__ void conv2d_shared_memory(
  const float* input, const float* filter,
  float* output, int H, int W, int C
) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE];
    // Collaborative loading and computation
    // ...
}
```

Optimized 1x1 Convolution

```
// 1x1 convolution optimized as matrix multiplication
__global__ void conv2d_1x1(
   const float* input, const float* filter,
   float* output, int N, int C_in, int C_out, int HW
) {
    int out_ch = blockIdx.y;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N * HW) {
        float sum = 0.0f;
        for(int c = 0; c < C_in; c++) {
            sum += filter[out_ch * C_in + c] *
                   input[idx * C_in + c];
        }
        output[idx * C_out + out_ch] = sum;
    }
}
```

Detailed Tiling Concept Explanation

1D to 2D Conversion

- **Linear index → 2D coordinates**: `idx = y * width + x`
- **2D coordinates → Linear index**: `x = idx % width, y = idx / width`

Shared Memory Tiling

- **Purpose**: Minimize global memory access
- **Method**: Divide data into small tiles and load into shared memory
- **Effect**: Maximize memory bandwidth efficiency

Thread Block Mapping

```
// Each thread handles one element of the tile
int tid = threadIdx.y * blockDim.x + threadIdx.x;
int tile_x = tid % TILE_WIDTH;
int tile_y = tid / TILE_WIDTH;

// Collaborative loading
__shared__ float tile[TILE_WIDTH][TILE_WIDTH];
tile[tile_y][tile_x] = global_mem[global_idx];
__syncthreads();
```

Performance Benefits of Tiling

Memory Access Pattern

1. **Coalesced Access**: Consecutive threads access consecutive memory
2. **Data Reuse**: Reuse data within tile multiple times
3. **Latency Hiding**: Hide memory latency

Practical Example: Matrix Multiplication

```
// Matrix multiplication with tiling
for (int tile = 0; tile < numTiles; tile++) {
    // Load tile to shared memory
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // Compute partial result
    for (int k = 0; k < TILE_SIZE; k++) {
        sum += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}
```

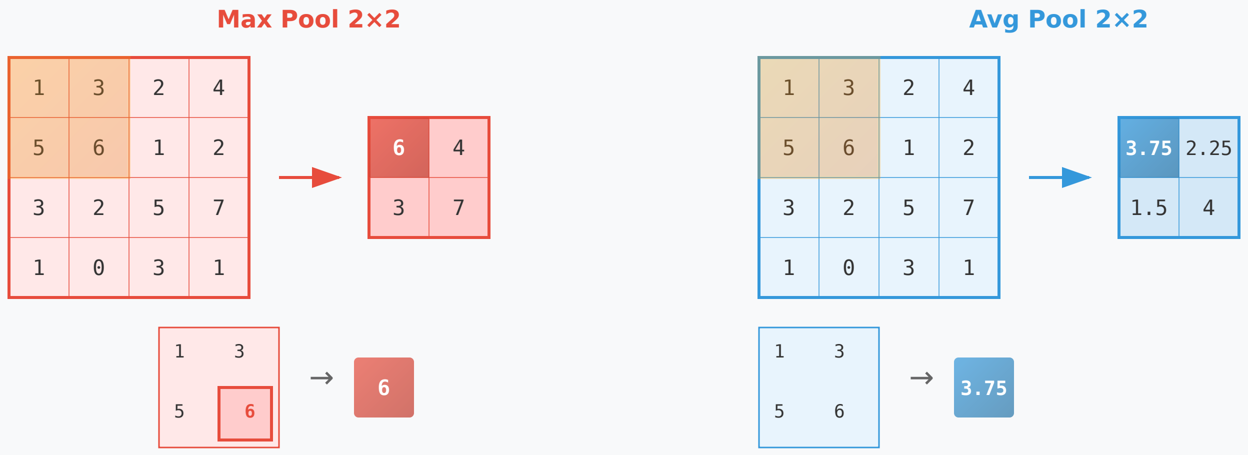
Performance improvement: Global memory access $N^3 \to N^3/TILE\_SIZE$

Batch Normalization Implementation

```
__global__ void batch_norm_forward(
    const float* input, const float* gamma, const float* beta,
    float* output, float* mean, float* variance,
    int N, int C, int H, int W, float epsilon
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= N * C * H * W) return;

    int c = (idx / (H * W)) % C;
    float normalized = (input[idx] - mean[c]) /
                       sqrtf(variance[c] + epsilon);
    output[idx] = gamma[c] * normalized + beta[c];
}
```

Pooling Operation Visualization



Pooling Layer Implementation

```
// Max Pooling kernel
__global__ void max_pooling_2d(
    const float* input, float* output,
    int H, int W, int pool_size, int stride
) {
    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;

    if (out_x < W/stride && out_y < H/stride) {
        float maxval = -FLT_MAX;
        for(int i = 0; i < pool_size; i++) {
            for(int j = 0; j < pool_size; j++) {
                int in_y = out_y * stride + i;
                int in_x = out_x * stride + j;
                maxval = fmaxf(maxval, input[in_y * W + in_x]);
            }
        }
```

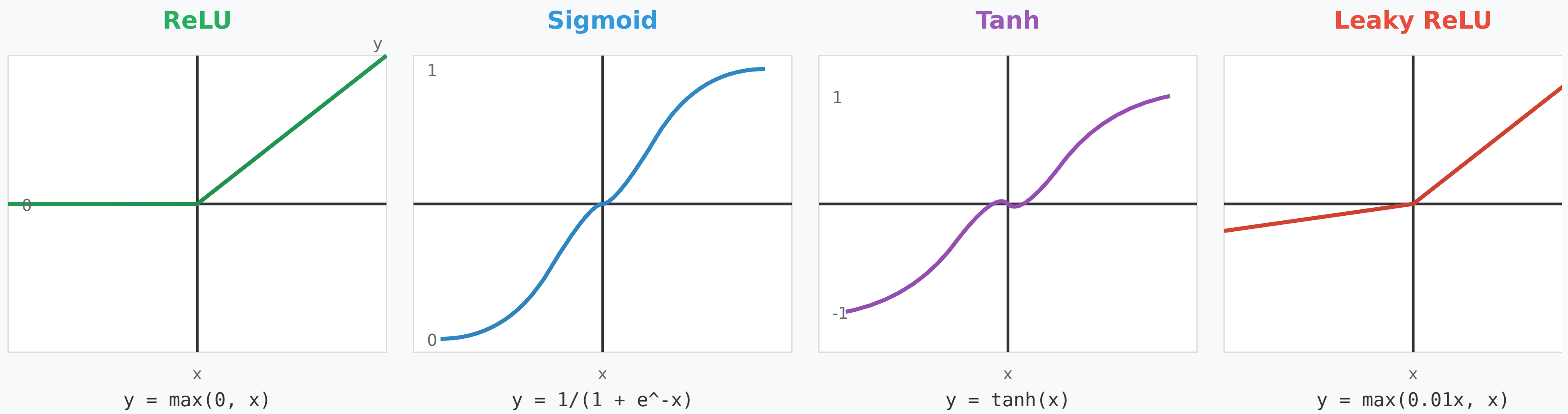
Average Pooling 구현

```
// Average Pooling kernel
__global__ void avg_pooling_2d(
    const float* input, float* output,
    int H, int W, int pool_size, int stride
) {
    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;

    if (out_x < W/stride && out_y < H/stride) {
        float sum = 0.0f;
        int count = 0;

        for(int i = 0; i < pool_size; i++) {
            for(int j = 0; j < pool_size; j++) {
                int in_y = out_y * stride + i;
                int in_x = out_x * stride + j;
                if(in_y < H && in_x < W) {
```

Activation Functions Visualization



CUDA Implementations

ReLU

```
__device__ float relu(float x) {
return fmaxf(0.0f, x);
}
```

Sigmoid

```
__device__ float sigmoid(float x) {
return 1.0f / (1.0f + expf(-x));
}
```

Tanh

```
__device__ float tanh_act(float x) {
return tanhf(x);
}
```

Leaky ReLU

```
__device__ float leaky_relu(float x) {
return fmaxf(0.01f * x, x);
}
```

Vectorized Kernel Example

```
__global__ void relu_activation(float* input, float* output, int n) {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < n) output[idx] = fmaxf(0.0f, input[idx]);
}
```

Activation Functions Implementation

```
// Fused activation kernel
__global__ void conv_relu_fused(
    const float* input, const float* kernel,
    float* output, int H, int W, int C
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= H * W * C) return;

    // Convolution computation
    float sum = compute_conv(input, kernel, idx);

    // Fused ReLU activation
    output[idx] = fmaxf(0.0f, sum);
}
```

Activation Gradient Implementation

```
// Backward pass for activation functions
__global__ void activation_backward(
    const float* grad_out, const float* input,
    float* grad_in, int n, ActivationType type
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= n) return;

    switch(type) {
        case RELU:
            grad_in[idx] = input[idx] > 0 ? grad_out[idx] : 0;
            break;
        case SIGMOID:
            float s = 1.0f / (1.0f + expf(-input[idx]));
            grad_in[idx] = grad_out[idx] * s * (1.0f - s);
            break;
    }
```

Activation Functions

Basic Activation Functions

```
__device__ float relu(float x) {
    return fmaxf(0.0f, x);
}

__device__ float sigmoid(float x) {
    return 1.0f / (1.0f + expf(-x));
}

__device__ float tanh_act(float x) {
    return tanhf(x);
}

__device__ float leaky_relu(float x, float alpha = 0.01f) {
    return x > 0 ? x : alpha * x;
}
```

Advanced Activation Functions

```
// GELU (Gaussian Error Linear Unit)
__device__ float gelu(float x) {
    const float sqrt_2_over_pi = 0.7978845608f;
    const float a = 0.044715f;
    float x3 = x * x * x;
    return 0.5f * x * (1.0f + tanhf(sqrt_2_over_pi * (x + a * x3)));
}

// Swish activation
__device__ float swish(float x, float beta = 1.0f) {
    return x * sigmoid(beta * x);
}
```

Vectorized Activation Kernels

```
// Process multiple elements per thread using float4
__global__ void relu_vectorized(const float4* input, float4* output, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        float4 val = input[idx];
        output[idx] = make_float4(
            fmaxf(0.0f, val.x), fmaxf(0.0f, val.y),
            fmaxf(0.0f, val.z), fmaxf(0.0f, val.w)
        );
    }
}
```

CNN 레이어 클래스

Conv2D Layer

```
class Conv2DLayer {
private:
    TensorFloat weights_, bias_, output_;
    int in_channels_, out_channels_;
    int kernel_h_, kernel_w_;
    int stride_, pad_;

public:
    Conv2DLayer(int in_ch, int out_ch, int kernel_size) {
        // 가중치 초기화
        cudaMalloc(&weights_, out_ch * in_ch * kernel_size * kernel_size);
        cudaMalloc(&bias_, out_ch);
    }

    TensorFloat& forward(const TensorFloat& input) {
        // Convolution 연산
        conv2d_kernel<<<grid, block>>>(
            input.data, weights_, bias_, output_.data
        );
```

BatchNorm & Pooling Layer

```
class BatchNormLayer {
private:
    float *gamma_, *beta_;
    float *running_mean_, *running_var_;
    float momentum_ = 0.1f;

public:
    TensorFloat& forward(const TensorFloat& input, bool training) {
        if (training) {
            // 배치 통계 계산 및 정규화
            compute_batch_stats<<<grid, block>>>(input);
        }
        batch_norm_forward<<<grid, block>>>(
            input, gamma_, beta_, output_, running_mean_, running_var_
        );
        return output_;
    }
};
```

CNN Network Configuration

```
class SimpleCNN {
private:
    std::vector<std::unique_ptr<Layer>> layers_;

public:
    SimpleCNN() {
        // Conv1: 3 -> 32
        layers_.push_back(std::make_unique<Conv2D>(3, 32, 3));
        layers_.push_back(std::make_unique<BatchNorm>(32));
        layers_.push_back(std::make_unique<ReLU>());
        layers_.push_back(std::make_unique<MaxPool2D>(2, 2));

        // Conv2: 32 -> 64
        layers_.push_back(std::make_unique<Conv2D>(32, 64, 3));
        layers_.push_back(std::make_unique<BatchNorm>(64));
        layers_.push_back(std::make_unique<ReLU>());
        layers_.push_back(std::make_unique<MaxPool2D>(2, 2));
    }
};
```

Forward Propagation

```
TensorFloat SimpleCNN::forward(const TensorFloat& input) {
    TensorFloat output = input;

    // Forward through each layer
    for (auto& layer : layers_) {
        output = layer->forward(output);
    }

    return output;
}
```

CNN 성능 벤치마킹

Custom CNN vs cuDNN

```
class CNNBenchmark {
    SimpleCNN custom_cnn;
    CuDNNCNN cudnn_cnn;
    CUDATimer timer;

public:
    void benchmark(int batch, int size) {
        TensorFloat input(batch, 3, size, size);

        // Warmup
        for(int i = 0; i < 10; i++) {
            custom_cnn.forward(input);
            cudnn_cnn.forward(input);
        }

        // Benchmark
        timer.start();
        for(int i = 0; i < 100; i++)
            custom_cnn.forward(input);
```

5.3 Project 2: Large-scale Image Processing Pipeline

Project Overview

Build a high-performance pipeline for real-time image/video processing

Practical Scenarios

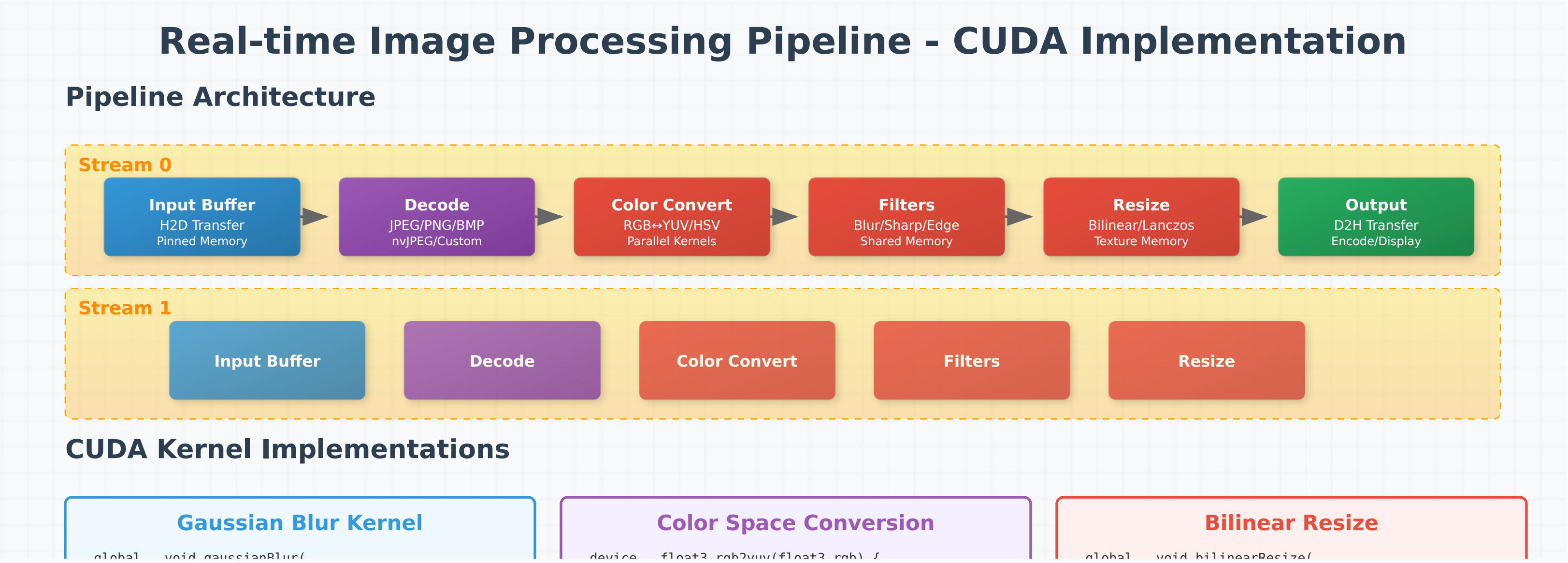
- **Real-time Streaming**: 4K/8K video real-time filtering
- **Batch Processing**: Process thousands of images simultaneously
- **Format Conversion**: JPEG, PNG, RAW format support

Implementation Features

- **Color Space Conversion**: RGB
  ↔️
  YUV, HSV
- **Filtering**: Gaussian, Sobel, Bilateral

Performance Goals

- 4K@60fps real-time processing
- 100x acceleration compared to CPU



Basic Image Processing Kernels

Gaussian Blur

```
__global__ void gaussian_blur(
    const uchar3* input, uchar3* output,
    int width, int height, float* kernel, int k_size
) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // Apply convolution with Gaussian kernel
}
```

Separable Filter Optimization

```
// Horizontal pass
__global__ void separable_filter_h(
    const uchar3* input, float3* temp,
    int w, int h, float* kernel, int k_size
) {
    // Process horizontal direction
}

// Vertical pass
__global__ void separable_filter_v(
    const float3* temp, uchar3* output,
    int w, int h, float* kernel, int k_size
) {
    // Process vertical direction
}
```

Edge Detection and Histogram

```
// Sobel edge detection
__global__ void sobel_edge(
    const uchar3* input, uchar3* output,
    int width, int height
) {
    // Apply Sobel operator
}

// Histogram equalization
__global__ void hist_equalize(
    const uchar* input, uchar* output,
    int* hist, int* cdf, int size
) {
    // Equalize histogram
}
```

Color Space Conversion

RGB → YUV Conversion

```
__global__ void rgb_to_yuv(
    const uchar3* rgb, uchar3* yuv, int size
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= size) return;

    // ITU-R BT.709 conversion matrix
    float3 color = make_float3(rgb[idx].x, rgb[idx].y, rgb[idx].z);

    yuv[idx].x = 0.2126f * color.x + 0.7152f * color.y + 0.0722f * color.z;
    yuv[idx].y = -0.0999f * color.x - 0.3360f * color.y + 0.4360f * color.z;
    yuv[idx].z = 0.6150f * color.x - 0.5586f * color.y - 0.0563f * color.z;
}
```

YUV → RGB Conversion

```
__global__ void yuv_to_rgb(
    const uchar3* yuv, uchar3* rgb, int size
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= size) return;

    // Inverse transformation matrix
    float3 color = make_float3(yuv[idx].x, yuv[idx].y, yuv[idx].z);

    rgb[idx].x = saturate(color.x + 1.28033f * color.z);
    rgb[idx].y = saturate(color.x - 0.21482f * color.y - 0.38059f * color.z);
    rgb[idx].z = saturate(color.x + 2.12798f * color.y);
}
```

Geometric Transforms

Image Resizing (Bilinear Interpolation)

```
__global__ void resize_bilinear(
    const unsigned char* input, unsigned char* output,
    int in_w, int in_h, int out_w, int out_h
) {
    int x_out = blockIdx.x * blockDim.x + threadIdx.x;
    int y_out = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate input image coordinates
    float x_in = x_out * (in_w - 1.0f) / (out_w - 1.0f);
    float y_in = y_out * (in_h - 1.0f) / (out_h - 1.0f);

    // Bilinear interpolation
    // Weighted average from 4 neighbor pixels
    output[idx] = bilinear_sample(input, x_in, y_in);
}
```

Image Rotation

```
__global__ void rotate_image(
    const unsigned char* input, unsigned char* output,
    int width, int height, float angle_rad
) {
    int x_out = blockIdx.x * blockDim.x + threadIdx.x;
    int y_out = blockIdx.y * blockDim.y + threadIdx.y;

    // Apply rotation transformation matrix
    float cos_a = cosf(angle_rad);
    float sin_a = sinf(angle_rad);

    // Calculate input coordinates using inverse transform
    float x_in = x_centered * cos_a - y_centered * sin_a;
    float y_in = x_centered * sin_a + y_centered * cos_a;

    output[idx] = bilinear_sample(input, x_in, y_in);
}
```

이미지 파이프라인 클래스

```
class ImagePipeline {
private:
    std::vector<std::function<void(Image&)>> stages_;
    cudaStream_t stream_;

public:
    struct Image {
        unsigned char* data;
        int width, height, channels;

        Image(int w, int h, int c) {
            cudaMalloc(&data, w * h * c);
        }
    };

    void addGaussianBlur(float sigma);
    void addSobelEdgeDetection();
    void addColorConversion(ColorSpace from, ColorSpace to);
    void addResize(int new_width, int new_height);
```

파이프라인 스테이지 추가

```
void ImagePipeline::addStage(std::function<void(Image&)> stage) {
    stages_.push_back(stage);
}

void ImagePipeline::addSobel() {
    stages_.push_back([](Image& img) {
        dim3 block(16, 16);
        dim3 grid((img.width + 15) / 16, (img.height + 15) / 16);
        sobel<<<grid, block>>>(img.data, img.width, img.height);
    });
}

void ImagePipeline::addResize(int w, int h) {
    stages_.push_back([w, h](Image& img) {
        Image resized(w, h, img.channels);
        resize<<<grid, block>>>(img.data, resized.data,
            img.width, img.height, w, h);
        img = std::move(resized);
    });
```

파이프라인 실행

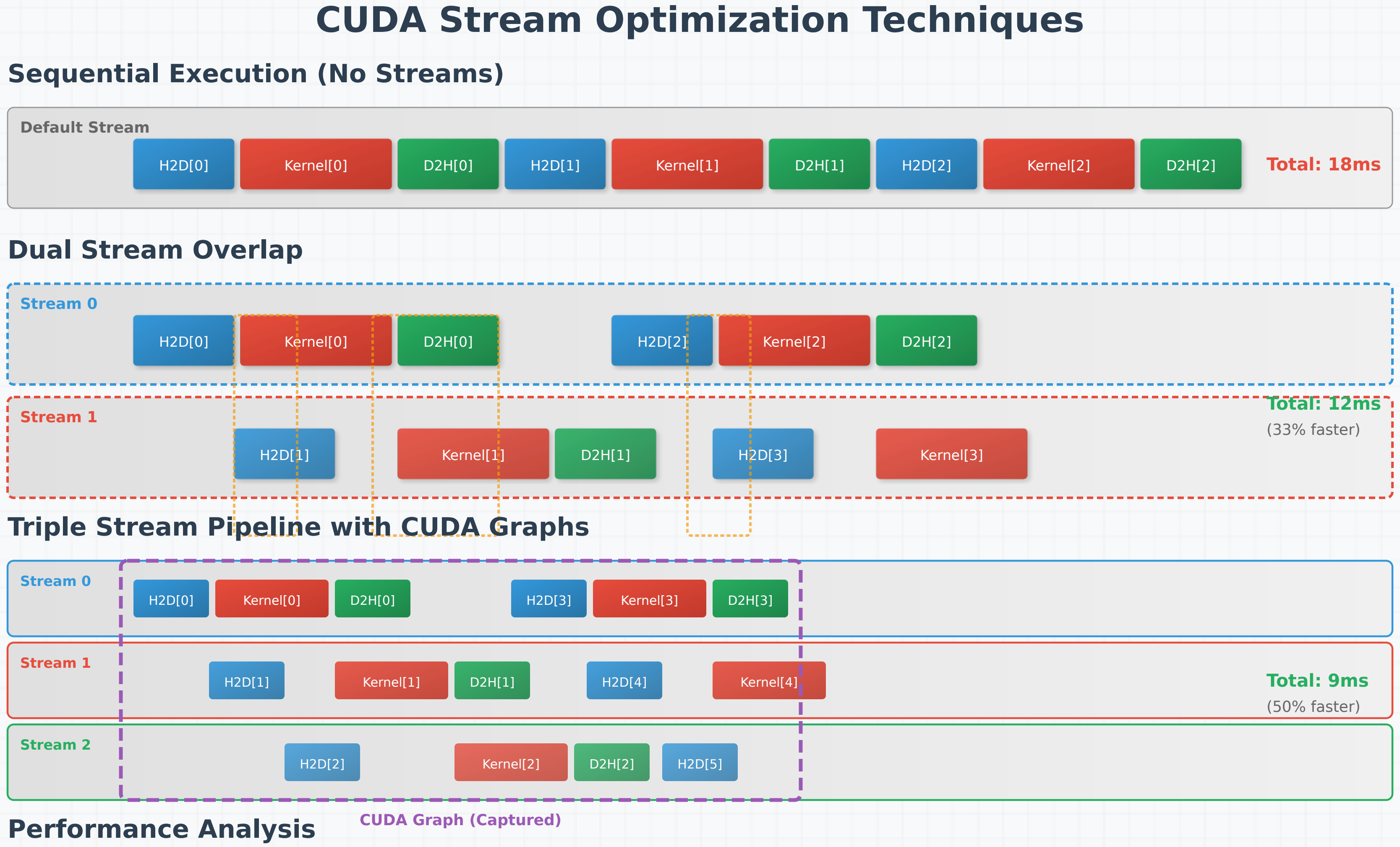
```
void ImagePipeline::process(const Image& input, Image& output) {
    Image current = input;

    for (auto& stage : stages_) {
        stage(current);
        cudaStreamSynchronize(stream_);
    }

    output = std::move(current);
}

// 사용 예제
ImagePipeline pipeline;
pipeline.addGaussianBlur(1.5f);
pipeline.addSobelEdgeDetection();
pipeline.addResize(640, 480);
pipeline.process(input_image, output_image);
```

스트리밍 최적화



Performance Analysis

Throughput Comparison

Configuration	FPS	Speedup
No Streams	55 FPS	1.0x
2 Streams	83 FPS	1.5x
3 Streams + Graph	111 FPS	2.0x

Optimization Guidelines

- Use streams = min(SM count, workload chunks)
- Ensure kernel execution time > transfer time
- Use CUDA Graphs for repetitive workloads
- Pin host memory for async transfers
- Profile with Nsight Systems for bottlenecks

Resource Utilization

Advanced streaming techniques for maximizing GPU throughput and minimizing latency

Multi-Stream Pipeline 구현

```
// Create and configure streams
cudaStream_t streams[NUM_STREAMS];
for(int i = 0; i < NUM_STREAMS; i++) {
    cudaStreamCreate(&streams[i]);
}

// Pipeline processing
for(int i = 0; i < num_batches; i++) {
    int sid = i % NUM_STREAMS;

    // Async H2D transfer
    cudaMemcpyAsync(d_in[sid], h_in[i], size,
                    cudaMemcpyHostToDevice, streams[sid]);

    // Process kernel
    process<<<grid, block, 0, streams[sid]>>>(
        d_in[sid], d_out[sid], w, h);

    // Async D2H transfer
    cudaMemcpyAsync(h_out[i], d_out[sid], size,
                    cudaMemcpyDeviceToHost, streams[sid]);
}
```

Stream Synchronization Strategy

```
// Event-based sync
cudaEvent_t events[NUM_STREAMS];
for(int i = 0; i < NUM_STREAMS; i++)
    cudaEventCreate(&events[i]);

// Process stages with events
for(int stage = 0; stage < num_stages; stage++) {
    for(int s = 0; s < NUM_STREAMS; s++) {
        kernel<<<grid, block, 0, streams[s]>>>();
        cudaEventRecord(events[s], streams[s]);

        if(s > 0)
            cudaStreamWaitEvent(streams[s], events[s-1], 0);
    }
}

// Sync all streams
for(int i = 0; i < NUM_STREAMS; i++)
    cudaStreamSynchronize(streams[i]);
```

Part 5. 요약

이 장에서 우리는 다음을 배웠습니다:

1. **프로젝트 설정과 구조**
    - 실무 CUDA 프로젝트의 특징과 표준 디렉토리 구조
    - CMake 설정, 핵심 유틸리티 헤더, 데이터 관리 및 벤치마킹 시스템
2. **프로젝트 1: CNN 계산 가속기**
    - CNN 기본 연산 분석 및 CUDA 구현 (컨볼루션, 배치 정규화, 풀링, 활성화 함수)
    - Tensor 클래스 및 CNN 네트워크 클래스 설계
    - 성능 벤치마킹 및 cuDNN과의 비교
3. **프로젝트 2: 대용량 이미지 처리 파이프라인**
    - 실시간 이미지 처리 파이프라인 요구사항
    - 이미지 처리 기본 커널들 (필터링, 색상 공간 변환, 기하학적 변환)
    - 이미지 파이프라인 클래스 설계 및 스트리밍 최적화

> **축하합니다!** 이제 여러분은 CUDA를 활용하여 복잡한 실제 애플리케이션을 설계하고 구현할 수 있는 전문가 수준의 개발자가 되었습니다. 이 지식을 바탕으로 다양한 분야에서 GPU 컴퓨팅의 힘을 발휘하시길 바랍니다.