Part 3. CUDA 고급: 전문가 기법

고급 최적화와 디버깅

프로덕션 레벨 CUDA 개발

CUDA 디버깅 - 수천 개 스레드의 버그 찾기

GPU 디버깅의 도전과제

도전과제	문제점	해결 방법	난이도
병렬 실행	수천 개 스레드 동시 실행	특정 스레드만 디버깅	
비동기성	에러 발생/감지 시점 차이	동기화 후 체크	
제한된 출력	printf 1KB 버퍼 한계	조건부 출력	
하드웨어 의존	GPU별 다른 동작	다양한 GPU 테스트	
메모리 에러	세그폴트 없이 잘못된 결과	cuda-memcheck	

필수 에러 처리 매크로

```
#define CUDA_CHECK(call) do { \
    cudaError_t error = call; \
    if (error != cudaSuccess) { \
        fprintf(stderr, "CUDA error at %s:%d: %s\n", \
                __FILE__, __LINE__, cudaGetErrorString(error)); \
        exit(EXIT_FAILURE); \
    } \
} while(0)

// □□ □□
CUDA_CHECK(cudaMalloc(&d_data, size));
kernel<<<blocks, threads>>>();
CUDA_CHECK(cudaGetLastError());  // □□ □□ □□
```

에러 처리 실습

즉시 실습: 에러 감지

```
// 어디서 에러가 발생할까?
kernel1<<<1000, 1024>>>();  // OK
kernel2<<<1000, 2048>>>();  // Block size 초과!
CUDA_CHECK(cudaGetLastError());  // 여기서 감지
```

핵심: 커널 실행 후 즉시 에러 체크 필수!

커널 내부 디버깅 - printf와 assert 활용

printf 디버깅 전략

Good: 스마트한 출력

```
__global__ void smartDebug(float *data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 첫 블록의 첫 스레드만
    if (blockIdx.x == 0 && threadIdx.x == 0) {
        printf("Grid: %d blocks, Block: %d threads\n",
               gridDim.x, blockDim.x);
    }

    // □□□ □□ □□ □□
```

Bad: 출력 폭탄

```
__global__ void badDebug(float *data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 모든 스레드가 출력 - 버퍼 오버플로우!
    printf("Thread %d: data[%d] = %f\n",
           idx, idx, data[idx]);
}
```

cuda-memcheck 도구 활용

도구	검사 내용	오버헤드	사용 시점
`memcheck`	메모리 접근 오류	10-50x	개발 중 항상
`racecheck`	Race condition	20-200x	동기화 버그 의심
`synccheck`	동기화 오류	2-5x	__syncthreads() 문제
`initcheck`	초기화되지 않은 메모리	10-100x	이상한 값 발생 시

```
# 실제 사용 예제
cuda-memcheck --leak-check full ./my_program
cuda-memcheck --tool racecheck --save racecheck.log ./my_program
```

Assert vs Printf 선택 가이드

즉시 실습: Assert vs Printf

```
// 언제 assert? 언제 printf?
assert(blockDim.x <= 1024);  // 설정 오류 - 즉시 중단
if (result < 0) printf("Warning: negative at %d\n", idx);  // 데이터 문제 - 계속 실행
```

상황	선택	이유
설정 오류	assert	즉시 중단 필요
데이터 문제	printf	디버깅 정보 수집
논리 오류	assert	개발 중 빠른 발견

Nsight 도구 - 프로급 디버깅과 프로파일링

Nsight 도구 선택 가이드

도구	용도	주요 메트릭	언제 사용?
Nsight Compute	커널 분석	Occupancy, Memory throughput	커널 최적화
Nsight Systems	전체 타임라인	CPU-GPU 동기화, 스트림	병목 찾기
cuda-gdb	라인별 디버깅	변수값, 콜스택	논리 에러

Nsight Compute 실전 활용

```
# 1. 기본 프로파일링
ncu ./my_program

# 2. 상세 메모리 분석
ncu --set full --target-processes all ./my_program
```

성능 병목 찾기 플로우

```
// Step 1: □□ □□ □□
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start);

kernel<<<blocks, threads>>>();

cudaEventRecord(stop);
```

성능 병목 진단 실습

Nsight Compute 결과 분석

```
Nsight Compute □□:
- Compute Throughput: 45%
- Memory Throughput: 92%  ← □□!
- L2 Cache Hit Rate: 23%

□□: Memory-bound kernel
□□: Shared memory □□, Coalescing □□
```

디버깅 체크리스트 - 운영 환경 준비

개발 단계별 디버깅 전략

단계	도구	체크 항목	자동화
개발	printf, assert	로직 검증	-
테스트	cuda-memcheck	메모리 오류	CI/CD 통합
최적화	Nsight Compute	성능 병목	프로파일링 스크립트
운영	Error handling	에러 로깅	모니터링 시스템

운영 환경 에러 처리 패턴

```
class CudaErrorHandler {
    std::ofstream log_file;
public:
    void check(cudaError_t err, const char* file, int line) {
        if (err != cudaSuccess) {
            log_file << "[" << timestamp() << "] CUDA Error: "
                     << cudaGetErrorString(err)
                     << " at " << file << ":" << line << std::endl;

            // □□ □□ □□ graceful shutdown
            attemptRecovery(err);
        }
    }
};
```

빌드 설정 가이드

디버그 vs 릴리즈 빌드

```
# Makefile
debug: NVCCFLAGS += -G -g -DDEBUG -lineinfo
debug: NVCCFLAGS += -Xcompiler -Wall -Xcompiler -Wextra
release: NVCCFLAGS += -O3 -DNDEBUG

# 사용
make debug    # 디버깅 빌드
make release  # 운영 빌드
```

빌드 타입	용도	성능	디버깅
Debug	개발/테스트	느림	가능
Release	운영	빠름	제한적

에러 처리 수준별 전략

CUDA 에러 처리 - 프로덕션 코드의 필수

수준	방법	장점	단점	사용 시점
Basic	return code 체크	간단함	반복 코드	프로토타입
Macro	CUDA_CHECK	일관성	매크로 의존	개발 단계
Exception	try-catch	복구 가능	오버헤드	프로덕션
Logging	파일/DB 기록	추적 가능	성능 영향	운영 환경

프로덕션 에러 처리 클래스

```
class CudaErrorHandler {
```

커널 에러 검사 패턴

Good: 즉시 검사

```
// □널 실행 후 즉시 검사
kernel<<<grid, block>>>();
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaDeviceSynchronize());

// 비동기 실행 검사
kernel<<<grid, block, 0, stream>>>();
CUDA_CHECK(cudaGetLastError());
// ... 다른 작업
CUDA_CHECK(cudaStreamSynchronize(stream));
```

Bad: 지연 검사

```
// 여러 □널 실행 후 검사
kernel1<<<grid, block>>>();
kernel2<<<grid, block>>>();
kernel3<<<grid, block>>>();
// 어떤 □널에서 오류?
CUDA_CHECK(cudaDeviceSynchronize());
```

즉시 실습: 에러 복구

```
// Out of Memory □□ □□
```

커널 내부 디버깅 기법

Advanced Printf 디버깅

```
__device__ int g_debug_counter = 0;

__global__ void advancedDebugKernel(float* data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // 조건부 디버깅 출력
    #ifdef DEBUG
    if (tid == 0) {
        printf("=== Kernel Launch Info ===\n");
        printf("Grid: %d blocks, Block: %d threads\n", gridDim.x, blockDim.x);
        printf("Total threads: %d, Data size: %d\n", gridDim.x * blockDim.x, N);
    }
    #endif

    // 특정 스레드만 추적
    const int DEBUG_THREAD = 1023;
    if (tid == DEBUG_THREAD) {
        printf("[Thread %d] Before: data[%d] = %f\n", tid, tid, data[tid]);
    }
```

Assert와 Trap 활용

커널 내 Assert

```
__global__ void kernelWithAssert(float* data, int* indices, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N) {
        // Debug 빌드에서만 활성화
        assert(indices[tid] >= 0 && indices[tid] < N);
        assert(data != nullptr);
```

Trap을 사용한 중단점

```
__global__ void kernelWithTrap(float* data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N) {
        if (data[tid] < 0) {
            // 디버거에서 중단
            printf("Trap at tid=%d, value=%f\n", tid, data[tid]);
            __trap();  // cuda-gdb에서 breakpoint처럼 동작
        }
```

Tip: Release 빌드에서는 assert 자동 제거됨 (성능 영향 없음)

Race Condition 찾기 실무

Race Condition 증상

흔한 징후들

- 실행할 때마다 다른 결과
- Debug 모드에서는 정상, Release에서 오류
- 작은 데이터는 정상, 큰 데이터에서 오류
- GPU 모델 바꾸면 결과 달라짐

주요 원인

1. **Missing __syncthreads()**
2. **Shared memory race**
3. **Global memory 동시 쓰기**
4. **Warp divergence에서 동기화 오류**

검출 도구

```
# compute-sanitizer로 race 검출
compute-sanitizer --tool racecheck ./program

# 상세 정보 포함
compute-sanitizer --tool racecheck --racecheck-report all ./program
```

Race Condition 패턴과 해결

Race Condition 있는 코드

```
__global__ void raceKernel(int* counter) {
    // 든 스레드가 동시에 쓰기
    (*counter)++;  // Race!
}

__global__ void sharedRace(float* out) {
    __shared__ float sum;

    if (threadIdx.x == 0)
        sum = 0;  // Race! 초기화 동기화 없음

    sum += threadIdx.x;  // Race!

    if (threadIdx.x == 0)
        *out = sum;
}
```

해결된 코드

```
__global__ void fixedKernel(int* counter) {
    // Atomic 연산 사용
    atomicAdd(counter, 1);
}

__global__ void fixedShared(float* out) {
    __shared__ float sum;

    if (threadIdx.x == 0)
        sum = 0;
    __syncthreads();  // 초기화 후 동기화

    // Atomic 더하기
    atomicAdd(&sum, (float)threadIdx.x);
    __syncthreads();  // 든 더하기 완료 대기

    if (threadIdx.x == 0)
```

Golden Rule: Shared memory 쓰기 전후엔 항상 __syncthreads()

Nsight Compute 실전 활용

프로파일링 워크플로우

1. 기본 프로파일링

```
# 전체 커널 프로파일링
ncu -o profile ./program

# 특정 커널만 프로파일링
ncu --kernel-name myKernel -o profile ./program

# 상세 메트릭 수집
ncu --set full -o profile ./program
```

2. 주요 메트릭 확인

- **Occupancy**: 실제 활성 warp 비율
- **SM Efficiency**: SM 활용도
- **Memory Throughput**: 메모리 대역폭 활용
- **Cache Hit Rate**: L1/L2 캐시 히트율

3. 병목 지점 식별

```
# Roofline 분석
ncu --section SpeedOfLight --section MemoryWorkloadAnalysis ./program
```

Nsight Compute 메트릭 해석

성능 문제 진단

Occupancy 낮음 (< 50%)

```
// □제: Register 과다 사용
__global__ void heavyKernel() {
    float regs[64];  // 너무 많은 레지스터
}

// 해결: Register 사용 줄이기
__launch_bounds__(256, 2)  // 제한 설정
__global__ void optimizedKernel() {
    float regs[16];  // 최소화
}
```

Memory 비효율

```
L1 Cache Hit: 20% → Coalescing □제
Global Load Efficiency: 25% → Strided access
```

CLI 자동화

```
# 배치 프로파일링 스크립트
for size in 1024 2048 4096; do
    ncu --csv --log-file metrics_$size.csv ./program $size
```

Dynamic Parallelism - GPU에서 GPU 커널 호출하기

Dynamic Parallelism vs Static Parallelism

비교	Static (CPU에서 호출)	Dynamic (GPU에서 호출)
레이턴시	Host-Device 통신	GPU 내부 통신
오버헤드	높음 (10-100μs)	낮음 (1-10μs)
유연성	제한적	높음
데이터 위치	메모리 이동 필요	GPU 메모리 상주
사용 사례	일관 처리	적응형/재귀

재귀적 커널 호출 패턴

```
__global__ void quickSort(int *data, int left, int right) {
    if (left >= right) return;

    int pivot = partition(data, left, right);

    if (right - left > THRESHOLD) {
        // □□□ □□ □□
        quickSort<<<1, 1>>>(data, left, pivot - 1);
        quickSort<<<1, 1>>>(data, pivot + 1, right);
    } else {
        sequentialSort(data, left, right);
    }
    cudaDeviceSynchronize();
}
```

핵심: GPU에서 직접 재귀 호출로 Host-Device 통신 제거

적응형 알고리즘 패턴

```
__global__ void adaptiveMeshRefinement(float *data, int level) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    float gradient = computeGradient(data, idx);

    if (gradient > THRESHOLD && level < MAX_LEVEL) {
        int new_blocks = computeNewBlocks(gradient);
        refinedComputation<<<new_blocks, 256>>>
            (data, idx, level + 1);
    }
    cudaDeviceSynchronize();
}
```

효과: 런타임에 필요한 부분만 세분화

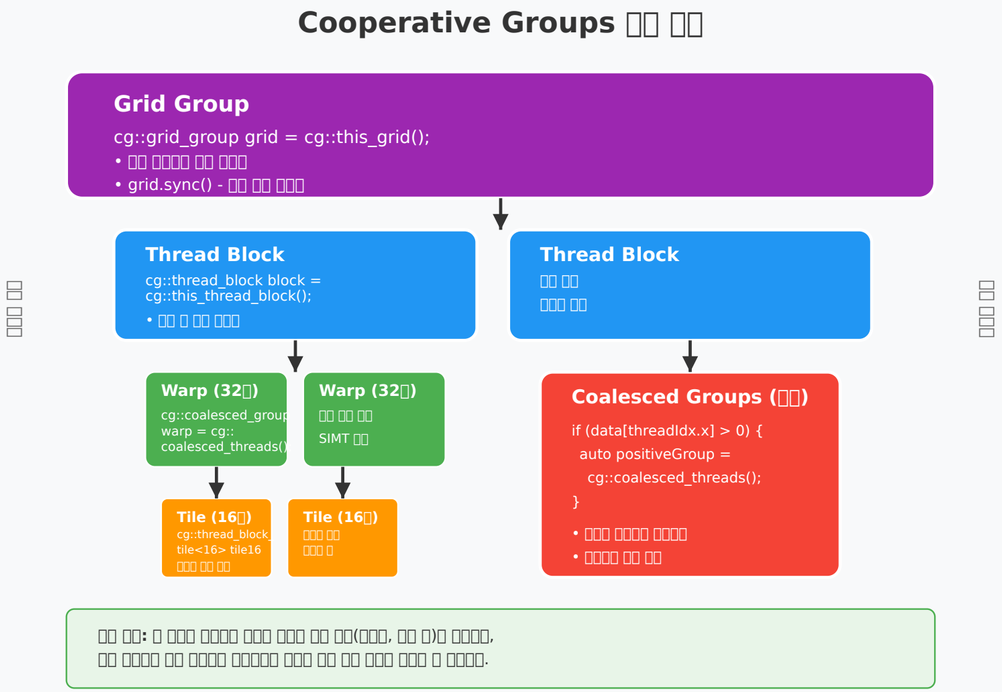
즉시 실습: 동적 vs 정적 비교

```
// 시나리오: N-body simulation
// Static: 1000번 CPU-GPU 통신 = 100ms
// Dynamic: GPU 내부 처리 = 10ms
// 전체 속도향상: 10x
```

한계: 커널 호출 오버헤드, 커널 스택 깊이 제한 (24레벨)

실습코드: `dynamic_parallelism.cu`

Cooperative Groups - 병렬성의 새로운 패러다임



기존 __syncthreads()의 한계

문제	기존 방식	Cooperative Groups
고정 스코프	Block 전체만	임의 그룹
정적 그룹	컴파일 시점 결정	런타임 결정
단순 동기화	sync만 가능	sync + reduce + shuffle
Warp Divergence	강제 대기	Active thread만 처리

그룹 계층 구조

```
namespace cg = cooperative_groups;

// 1. 전체 그리드 (Multi-GPU 지원)
auto grid = cg::this_grid();

// 2. Thread Block (기존 __syncthreads() 대체)
auto block = cg::this_thread_block();

// 3. Warp 분할 (32, 16, 8, 4 threads)
auto warp = cg::tiled_partition<32>(block);
auto tile16 = cg::tiled_partition<16>(block);

// 4. 동적 그룹 (active thread만)
auto active = cg::coalesced_threads();
```

Warp-level Reduction 비교

Cooperative Groups 방식

```
__global__ void warpReduce(int *data, int *result) {
    auto tile = cg::tiled_partition<32>(cg::this_thread_block());
    int val = data[threadIdx.x];

    // Warp shuffle reduction (no sync needed!)
    for (int offset = 16; offset > 0; offset /= 2)
        val += tile.shfl_down(val, offset);

    if (tile.thread_rank() == 0)
        atomicAdd(result, val);
```

기존 Shared Memory 방식

```
__shared__ int sdata[256];
sdata[threadIdx.x] = data[threadIdx.x];
__syncthreads();  // 8□□ □□□ □□!
for (int s = 128; s > 0; s >>= 1) {
    if (threadIdx.x < s)
        sdata[threadIdx.x] += sdata[threadIdx.x + s];
    __syncthreads();
```

성능 차이: Shared memory 8 sync vs CG 0 sync = 2-3x 빠름

동적 그룹 및 고급 패턴

Coalesced Group 활용

```
__global__ void adaptiveProcess(float *data, float threshold) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    bool needs_work = data[idx] > threshold;

    auto active = cg::coalesced_threads();  // □□ □□□□
    if (active.size() > 0) {
        float result = expensiveComputation(data[idx]);
        float neighbor = active.shfl_xor(result, 1);
        data[idx] = (result + neighbor) * 0.5f;
    }
}
```

효과: Warp divergence에서도 효율적 실행

Multi-GPU 및 Cluster 기능

Multi-GPU Cooperative Kernel

```
__global__ void multiGpuReduce(int *data, int *result) {
    auto grid = cg::this_grid();
    auto block = cg::this_thread_block();

    extern __shared__ int sdata[];
    blockReduce(data, sdata);
    grid.sync();  // Multi-GPU sync!

    if (block.thread_rank() == 0 && grid.block_rank() == 0)
```

Thread Block Cluster (CUDA 11.0+)

```
__global__ void __cluster_dims__(2, 2, 1) clusterKernel(float *data) {
    auto cluster = cg::this_cluster();  // 4 blocks = 1 cluster
    __shared__ float sdata[256];
    cluster.sync();  // Cluster-wide sync
```

주요 장점:

- Warp divergence 최소화
- Shared memory 사용량 감소
- 복잡한 동기화 패턴 단순화

Persistent Threads 패턴

Work Queue 방식 구현

```
__global__ void persistentKernel(int* workQueue, int* results, int totalWork) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int gridSize = gridDim.x * blockDim.x;

    // □ 스레드가 여러 작업 처리
    while (true) {
        // Atomic으로 다음 작업 가져오기
        int workIdx = atomicAdd(workQueue, 1);

        if (workIdx >= totalWork) break;  // 모든 작업 완료

        // 작업 처리
        int result = processWork(workIdx);
        results[workIdx] = result;

        // Load balancing: 무거운 작업도 균등 분배
    }
}
```

Producer-Consumer 패턴

Ring Buffer 구현

```
__device__ volatile int head = 0;
__device__ volatile int tail = 0;
__device__ int buffer[BUFFER_SIZE];

__global__ void producer(int* input, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = tid; i < N; i += gridDim.x * blockDim.x) {
        int value = processInput(input[i]);

        // Ring buffer□ □□
        int slot;
        do {
            slot = atomicAdd((int*)&head, 1) % BUFFER_SIZE;
            // Buffer full □□
            while ((slot - tail + BUFFER_SIZE) % BUFFER_SIZE >= BUFFER_SIZE - 1) {
                __threadfence();  // □□ □□□ □□ □□
            }
        } while (false);
```

장점: 커널 재실행 오버헤드 제거, 동적 로드 밸런싱

적응형 병렬화 (Adaptive Parallelism)

작업 부하에 따라 동적으로 병렬화 수준을 조정하는 기법입니다.

```
__global__ void adaptiveKernel(float *input, float *output,
                               int *workload, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
        int load = workload[idx];

        if (load < LIGHT_THRESHOLD) {
            output[idx] = lightProcess(input[idx]);
        } else if (load < HEAVY_THRESHOLD) {
            cg::coalesced_group g = cg::coalesced_threads();
            output[idx] = mediumProcess(input[idx], g);
        } else {
            heavyProcessKernel<<<(load+255)/256, 256>>>(
                input + idx, output + idx, load);
        }
    }
}
```

계층적 병렬화

```
__global__ void hierarchicalKernel(TreeNode *nodes, int numNodes) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < numNodes) {
        TreeNode &node = nodes[idx];
        if (node.isLeaf()) {
            processLeaf(node);
        } else {
            // □식 노드 수에 따라 다른 전략 사용
            if (node.numChildren < WARP_SIZE) {
                // 워프 내에서 처리
            } else {
                // 새로운 커널 실행
            }
        }
    }
}
```

핵심: 작업량에 따라 최적의 병렬화 전략 선택

실습 파일: `adaptive_parallelism.cu`

커널 융합 (Kernel Fusion)

Kernel fusion은 여러 개의 연속적인 커널 호출을 하나의 커널로 결합하는 최적화 기법입니다.

융합 전 (개별 커널)

```
data → Scale → Add → ReLU → output
         ↓       ↓       ↓
    Global   Global Global
    Memory   Memory Memory
```

커널 융합 구현

융합 후 (통합 커널)

```
__global__ void fusedScaleAddRelu(
    float *a, float *b, float *output, float scale_factor) {

    float scaled_a = a[idx] * scale_factor;
    float sum = scaled_a + b[idx];
    output[idx] = fmaxf(0.0f, sum);
}
```

실전 Fusion 사례 - CNN 레이어

Convolution + Batch Norm + ReLU Fusion

```
__global__ void fusedConvBNReLU(
    float *input, float *weight, float *bias,
    float *bn_scale, float *bn_shift,
    float *output, int channels) {
```

성능 비교 결과

방식	Memory BW	Kernel Overhead	총 시간	속도향상
개별 커널	5.2 GB/s	150 μs	1.2 ms	1.0x
Fused 커널	8.1 GB/s	50 μs	0.4 ms	3.0x

메모리 사용량: 60% 감소 (intermediate buffer 제거) **전력 효율**: 40% 향상 (memory traffic 감소)

즉시 실습: Auto-Fusion 감지

```
// □□ fusion□□ □□□?
float fusion_benefit =
    (original_memory_access - fused_memory_access) /
```

주의사항: Register 압박, Occupancy 감소 가능성 **반대 사례**: 복잡한 브랜칭, 다른 메모리 패턴

실습코드: `advanced_fusion.cu`

커널 분할 전략

때로는 하나의 큰 커널을 여러 개로 나누는 것이 유리합니다.

주요 분할 이유:

- 메모리 제약 (Shared Memory/레지스터 부족)
- 점유율 최적화
- 성능 최적화
- 디버깅 및 유지보수성

1. 메모리 제약에 의한 분할

Shared Memory나 레지스터 부족으로 인해 커널을 여러 패스로 분할

```
// Problem: Shared memory □□
__global__ void largeSharedMemKernel(float *input, float *output, int n) {
    // Error: 48KB needed but only 48KB available per SM
    __shared__ float buffer[12288];  // 48KB
    // ...
}
```

실습 파일: `kernel_splitting_memory.cu`

메모리 제약 해결 방법

타일 크기를 줄이고 여러 패스로 분할

```
// Solution: □□ □□□ □□□ □□ □□□ □□
__global__ void tiledKernelPass1(float *input, float *temp, int n) {
    __shared__ float buffer[4096];  // 16KB only
    // Process first part
}

__global__ void tiledKernelPass2(float *temp, float *output, int n) {
    __shared__ float buffer[4096];  // 16KB only
    // Process second part
}
```

2. 점유율 최적화를 위한 분할

레지스터 사용량이 너무 많아 점유율이 저하될 때 계산을 여러 단계로 분할

```
// Problem: 레지스터 사용량이 너무 많아 점유율 저하
__global__ void complexKernel(float *data, int n) {
    float registers[64];  // 많은 레지스터 사용
    // Complex computation...
}
```

실습 파일: `kernel_splitting_occupancy.cu`

점유율 최적화 해결 방법

계산을 단계별로 분할하여 각 단계의 레지스터 사용량 감소

```
// Solution: 계산을 단계별로 분할
__global__ void stage1Kernel(float *input, float *intermediate, int n) {
    // 레지스터 사용량: 32개
    // ...
}

__global__ void stage2Kernel(float *intermediate, float *output, int n) {
    // 레지스터 사용량: 32개
    // ...
}
```

CUDA 템플릿 프로그래밍

템플릿을 사용하면 다양한 데이터 타입과 파라미터에 대해 재사용 가능한 커널을 작성할 수 있습니다.

타입 템플릿

```
template <typename T>
__global__ void vectorAdd(T *a, T *b, T *c, int n) {
    c[idx] = a[idx] + b[idx];
}
```

템플릿 커널 활용

값 템플릿

```
template <int BLOCK_SIZE>
__global__ void reduction(float *input, float *output) {
    __shared__ float sdata[BLOCK_SIZE];
}
```

사용 예시

```
vectorAdd<float><<<blocks, threads>>>(f_a, f_b, f_c, n);
vectorAdd<double><<<blocks, threads>>>(d_a, d_b, d_c, n);
reduction<256><<<blocks, 256>>>(input, output);
```

템플릿 프로그래밍의 장점

- **코드 재사용**: 하나의 구현으로 다양한 타입 지원
- **컴파일 타임 최적화**: 타입별로 최적화된 코드 생성
- **타입 안정성**: 컴파일 타임 타입 검사

핵심: 코드 중복을 줄이고, 컴파일 타임에 최적화된 코드를 생성

실습 파일: `template_kernels.cu`

Tensor Cores - AI 연산 가속의 혁신

Tensor Core 하드웨어 지원 및 성능

GPU 세대	Tensor Core 버전	지원 정밀도	성능 (TOPS)
Volta (V100)	1세대	FP16	125 TOPS
Turing (RTX 20xx)	2세대	INT8, FP16	260 TOPS
Ampere (A100)	3세대	BF16, TF32, FP16	624 TOPS
Hopper (H100)	4세대	FP8, BF16, TF32	1979 TOPS

기본 WMMA API 사용법

```
#include <mma.h>
using namespace nvcuda;

__global__ void tensorCoreGEMM_optimized(half* A, half* B, float* C, int M, int N, int K) {
    // 최적화된 16x16x16 Tensor Core 연산
    constexpr int WMMA_M = 16, WMMA_N = 16, WMMA_K = 16;
```

즉시 실습: GEMM 성능 비교

```
// 4096x4096 행렬 곱셈 비교
// Standard CUDA Core: 8.5 TFLOPS
// Tensor Core (FP16): 45.2 TFLOPS
// 속도 향상: 5.3x
```

Mixed Precision 및 고급 최적화

marp: true theme: custom-theme-modular class: code-focus

정밀도별 성능 비교

정밀도	비트	메모리 효율	성능	정확도	사용 사례
FP32	32	1x	1x	최고	과학 계산
TF32	19	1x	8x	높음	일반 딥러닝
BF16	16	2x	16x	중간	훈련
FP16	16	2x	16x	낮음	추론
INT8	8	4x	32x	낮음	배포
FP8	8	4x	64x	매우 낮음	최신 모델

실전 Mixed Precision - Good Practice

전략적 정밀도 사용

```
__global__ void smartMixedPrecision(
    half* weights, float* gradients, float* master_weights) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 1. Forward: FP16로 빠른 연산
    half2 w_h2 = *reinterpret_cast<half2*>(&weights[idx * 2]);
    half2 result = __hmul2(w_h2, input_h2);

    // 2. Loss: FP32로 정밀도 보장
    float loss = computeLoss(result);

    // 3. Gradient: FP32 master weight 업데이트
    float grad = gradients[idx];
    master_weights[idx] -= 0.001f * grad;

    // 4. Weight sync: FP32 → FP16
    weights[idx] = __float2half(master_weights[idx]);
}
```

핵심: 연산은 FP16, 누적은 FP32로 분리

Mixed Precision - Bad Practice

무분별 정밀도 사용

```
__global__ void badMixedPrecision(half* data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

문제: 언더플로우/오버플로우 발생

TF32 자동 가속 (Ampere+)

```
// TF32 □□ □□
cudaSetTensorMathMode(cudaDataType_t::CUDA_R_32F);
```

주의: TF32는 기본 활성화, 비활성화 하려면:

```
cudaSetTensorMathMode(cudaDataType_t::CUDA_R_32F, false);
```

즉시 실습: 정밀도 선택 가이드

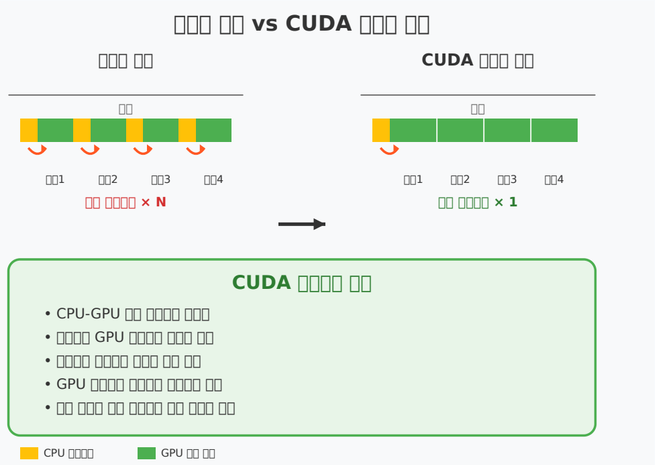
```
□□ □□□ □□:
```

성능 도득: Tensor Core 활용시 5-50x 속도향상 **주의사항**: Numerical stability, Gradient scaling 필요

실습코드: `tensor_cores_mixed_precision.cu`

CUDA Graphs - CPU 오버헤드 제거의 게임 체인저

전통적 실행 vs 그래프 실행



주요 장점

- CPU 오버헤드 최소화
- 반복 실행 최적화
- 작업 의존성 자동 관리

그래프 생성 방법

Stream Capture (권장)

```
cudaStreamBeginCapture(stream,
    cudaStreamCaptureModeGlobal);

kernel<<<blocks, threads, 0, stream>>>();

cudaStreamEndCapture(stream, &graph);
```

언제 사용?

- 반복적인 워크로드
- 고정된 실행 패턴
- 낮은 레이턴시 필요

Stream Capture 구현

```
class CudaGraphManager {
private:
    cudaGraph_t graph;
```

명시적 API 구현

```
void createGraphExplicitly() {
    cudaGraph_t graph;
    cudaGraphCreate(&graph, 0);

    // □□ □□
    std::vector<cudaGraphNode_t> nodes;

    // □□ □□ □□
    cudaKernelNodeParams kernelParams = {};
    kernelParams.func = (void*)myKernel;
    kernelParams.gridDim = grid;
    kernelParams.blockDim = block;

    cudaGraphNode_t kernelNode;
    cudaGraphAddKernelNode(&kernelNode, graph,
                           nullptr, 0, &kernelParams);
```

특징: 노드 간 의존성을 명시적으로 정의

그래프 업데이트 기법

파라미터 업데이트 (재생성 없이)

```
void updateGraphParameters() {
    // 그래프 □드 찾기
    cudaGraphNode_t node;
    size_t numNodes = 1;
    cudaGraphGetNodes(graph, &node, &numNodes);

    // 커널 파라미터 업데이트
    cudaKernelNodeParams newParams;
    cudaGraphKernelNodeGetParams(node, &newParams);
    newParams.kernelParams[0] = newData;
    cudaGraphKernelNodeSetParams(node, &newParams);

    // 실행 그래프 업데이트
    cudaGraphExecUpdate(graphExec, graph, nullptr, nullptr);
}
```

장점: 1000x 빠른 파라미터 업데이트 (vs 재생성)

조건부 그래프

```
void createConditionalGraph() {
    cudaGraph_t mainGraph;
    cudaGraphCreate(&mainGraph, 0);

    // 조건부 □드 생성
    cudaGraphNode_t conditionalNode;
    cudaConditionalNodeParams condParams = {};
    condParams.type = cudaGraphCondTypeIf;

    cudaGraphAddConditionalNode(&conditionalNode, mainGraph,
                                nullptr, 0, &condParams);

    // 조건에 따른 서브그래프 정의
    if (condition) {
        cudaGraphSetConditionalNodeParams(conditionalNode, &condParams);
    }
}
```

활용: 런타임 조건 분기

그래프 성능 최적화 가이드

최적화 기법

1. 그래프 크기 최적화

- 너무 작은 그래프는 오버헤드 증가
- 적절한 크기의 작업 단위로 그룹화

2. 메모리 접근 최적화

- 그래프 내에서 메모리 재사용
- 불필요한 메모리 복사 제거

3. 의존성 최소화

- 불필요한 동기화 지점 제거
- 병렬 실행 가능한 작업 분리

성능 측정:

```
float measureGraphPerformance() {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    cudaGraphLaunch(graphExec, stream);
```

그래프 디버깅

1. 그래프 구조 분석

```
void analyzeGraph() {
    size_t numNodes;
    cudaGraphGetNodes(graph, nullptr, &numNodes);

    std::vector<cudaGraphNode_t> nodes(numNodes);
    cudaGraphGetNodes(graph, nodes.data(), &numNodes);

    for (auto node : nodes) {
        cudaGraphNodeType type;
        cudaGraphNodeGetType(node, &type);

        switch (type) {
            case cudaGraphNodeTypeKernel:
                printf("Kernel node found\n");
                break;
            case cudaGraphNodeTypeMemcpy:
                printf("Memcpy node found\n");
```

2. 실행 오류 처리

그래프 사용 가이드라인

언제 사용할까?

상황	그래프 효과	성능 향상
반복 실행 10회+	매우 효과적	5-10x
작은 커널 연속 실행	필수	10-100x
CPU-GPU 빈번한 동기화	추천	2-5x
메모리 할당/해제 포함	유용	3-10x

조건부 실행과 동적 그래프

CUDA 그래프에서 조건부 실행과 동적 업데이트를 구현하는 방법

주요 기능:

- 조건부 그래프 노드 (CUDA 11+)
- 동적 그래프 파라미터 업데이트
- 런타임 그래프 수정

실습 파일: `conditional_graphs.cu`

조건부 그래프 노드

CUDA 11부터 `cudaGraphConditionalNode`를 사용하여 그래프 내에서 조건부 실행을 구현

```
class ConditionalGraph {
private:
    cudaGraph_t mainGraph;
    cudaGraph_t branchA;
    cudaGraph_t branchB;
    cudaGraphExec_t mainExec;
    cudaGraphConditionalHandle handle;

public:
    void createConditionalGraph() {
        cudaGraphCreate(&mainGraph, 0);
        createBranchA(&branchA);
        createBranchB(&branchB);
        // ...
    }
};
```

조건부 노드 생성 및 실행

```
void createConditionalGraph() {
    cudaGraphConditionalParams conditionalParams;
    conditionalParams.handle = handle;
    conditionalParams.type = cudaGraphCondTypeIf;
    conditionalParams.size = 1;

    cudaGraphAddConditionalNode(&conditionalNode, mainGraph,
                                nullptr, 0, &conditionalParams);
}

void executeWithCondition(bool condition) {
    unsigned int value = condition ? 1 : 0;
    cudaGraphSetConditional(handle, &value, cudaGraphCondAssignDefault);
    cudaGraphLaunch(mainExec, 0);
}
```

동적 그래프 업데이트

`cudaGraphExecKernelNodeSetParams` 등을 사용하여 그래프 실행 인스턴스의 파라미터를 동적으로 변경

```
void updateGraphDynamically(cudaGraphExec_t graphExec,
                            void *newKernelArgs,
                            size_t newSharedMem) {
    cudaGraphExecKernelNodeSetParams(graphExec, kernelNode, &newParams);
    cudaGraphExecMemcpyNodeSetParams(graphExec, memcpyNode, &newMemcpyParams);
    cudaGraphLaunch(graphExec, stream);
}
```

실습 파일: `dynamic_graph_update.cu`

그래프 최적화 기법

1. 그래프 템플릿

템플릿을 사용하여 다양한 데이터 타입이나 파라미터에 대해 재사용 가능한 그래프를 생성합니다.

```
template<typename T, int BLOCK_SIZE>
class GraphTemplate {
private:
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
public:
    void createTemplate(int numElements);
    void execute(cudaStream_t stream) {
        cudaGraphLaunch(graphExec, stream);
    }
};
```

2. 멀티 GPU 그래프

각 GPU에 대한 그래프를 생성하고, GPU 간 통신을 그래프 노드로 포함합니다.

```
class MultiGPUGraph {
private:
    std::vector<cudaGraph_t> deviceGraphs;
    std::vector<cudaGraphExec_t> deviceExecs;
    int numDevices;

public:
    void createMultiGPUGraph();
    void executeAllDevices();
};
```

실습 파일:

- `graph_templates.cu`
- `multi_gpu_graphs.cu`

그래프 디버깅과 프로파일링

그래프 시각화

```
void visualizeGraph(cudaGraph_t graph) {
    size_t numNodes;
    cudaGraphGetNodes(graph, nullptr, &numNodes);

    std::vector<cudaGraphNode_t> nodes(numNodes);
    cudaGraphGetNodes(graph, nodes.data(), &numNodes);

    printf("Graph contains %zu nodes:\n", numNodes);
}
```

실습 파일: `graph_visualizer.cu`

그래프 실행 시간 측정

```
class GraphProfiler {
private:
    std::map<std::string, float> timings;
    cudaEvent_t start, stop;

public:
    void profileGraph(const std::string &name,
                      cudaGraphExec_t graphExec,
                      cudaStream_t stream,
                      int iterations = 100);
};
```

도구: Nsight Systems로 그래프 실행 흐름과 성능 병목 분석

실습 파일: `graph_profiler.cu`

CUDA Runtime vs Driver API

Overview and Key Differences

Two Levels of CUDA API

CUDA provides two API levels for different needs:

- **Runtime API**: High-level, user-friendly
- **Driver API**: Low-level, full control

Quick Comparison

Side-by-Side Code Example

Runtime API

```
#include <cuda_runtime.h>

// Automatic initialization
float *d_data;
cudaMalloc(&d_data, size);

// Simple kernel launch
kernel<<<grid, block>>>(d_data);

// Error checking
cudaError_t err = cudaGetLastError();
```

Driver API

```
#include <cuda.h>

// Manual initialization
cuInit(0);
CUcontext ctx;
cuCtxCreate(&ctx, 0, device);
```

Decision Matrix

Which API Should You Use?

Criteria	Runtime API	Driver API
Learning Curve	Easy	Steep
Development Speed	Fast	Slow
Code Complexity	Simple	Complex
Performance	98-99%	100%
JIT Compilation	No	Yes
Multi-Context	Limited	Full
PTX Loading	No	Yes
Graphics Interop	Basic	Advanced

Recommendation

- **Start with Runtime API** (99% of use cases)
- **Use Driver API only when** you need:
  - Runtime compilation
  - Multiple contexts
  - Custom PTX/CUBIN loading
  - Advanced memory management

Advanced Features Comparison

Unique Capabilities

Runtime API Only

- Unified Memory (`cudaMallocManaged`)
- Cooperative Groups (high-level)
- Graph API (simplified)
- Built-in libraries (cuBLAS, cuDNN)

Driver API Only

- JIT compilation from PTX
- Module management
- Virtual memory mapping
- Function attributes query
- Context priority control

Example: JIT Compilation (Driver API)

```
// Only possible with Driver API
const char* ptx_code = generate_ptx_at_runtime();
CUmodule module;
cuModuleLoadData(&module, ptx_code);
```

Migration Path

Converting Between APIs

Runtime to Driver

```
// Runtime API
cudaMemcpy(d_dst, d_src, size, cudaMemcpyDeviceToDevice);

// Equivalent Driver API
cuMemcpyDtoD(d_dst, d_src, size);
```

Getting Driver Context from Runtime

```
// Initialize with Runtime API
cudaSetDevice(0);

// Get Driver API context
CUcontext driverContext;
cuCtxGetCurrent(&driverContext);

// Now use Driver API functions
CUmodule module;
```

Best Practice

- Use Runtime API as primary
- Access Driver API features when needed
- Maintain single context when mixing

CUDA API Level Comparison

Runtime vs Driver API

Two API Levels

CUDA provides two levels of APIs for different use cases:

Aspect	Runtime API	Driver API
Header	`cuda_runtime.h`	`cuda.h`
Initialization	Automatic	Explicit (`cuInit`)
Context	Auto-managed	Manual
Prefix	`cuda`	`cu`
Ease of Use	High	Low
Control Level	Limited	Complete

API Feature Comparison

Runtime API Features

```
// Automatic initialization
cudaSetDevice(0);  // Simple device selection

// Unified memory support
float *data;
cudaMallocManaged(&data, size);

// Built-in error handling
cudaError_t err = cudaGetLastError();
if(err != cudaSuccess) {
    printf("Error: %s\n", cudaGetErrorString(err));
```

Driver API Features

```
// Manual everything
CUresult res;
CUdevice device;
CUcontext context;

// Explicit initialization
res = cuInit(0);
res = cuDeviceGet(&device, 0);
res = cuCtxCreate(&context, 0, device);
```

Memory Management Comparison

Runtime API Memory

```
// Simple allocation
float *d_ptr;
cudaMalloc(&d_ptr, size);
cudaMemcpy(d_ptr, h_ptr, size, cudaMemcpyHostToDevice);

// Unified memory
float *unified;
cudaMallocManaged(&unified, size);
// Direct access from both CPU and GPU
```

Driver API Memory

```
// More control over memory
CUdeviceptr d_ptr;
cuMemAlloc(&d_ptr, size);

// Memory attributes
CUmemAllocationProp prop = {};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
```

Kernel Launch Comparison

Runtime API Launch

```
// Simple kernel launch
__global__ void kernel(float *data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) data[idx] *= 2.0f;
}

// Launch with <<<>>> syntax
kernel<<<gridSize, blockSize, sharedMem, stream>>>(data, N);
```

Driver API Launch

```
// Complex but flexible
void* args[] = {&d_data, &N};

// Launch with parameter buffer
cuLaunchKernel(function,
    gridX, gridY, gridZ,
    blockX, blockY, blockZ,
    sharedMem, stream, args, NULL);

// Cooperative launch
```

API Selection Guide

When to Use Each API

Runtime API Use Cases

Suitable for:

- Rapid prototyping
- Standard GPU computing tasks
- Educational purposes

Advantages:

- Fast development
- Less code
- Automatic optimization

Example Applications:

```
// Deep Learning with cuDNN
cudnnHandle_t cudnn;
cudnnCreate(&cudnn);
cudnnConvolutionForward(cudnn, ...);

// Scientific Computing with cuBLAS
cublasHandle_t cublas;
cublasCreate(&cublas);
```

Driver API Use Cases

When Driver API is Necessary

Required for:

- Runtime kernel compilation (JIT)
- Multiple context management
- Low-level memory control

Example: Dynamic Kernel Generation

```
// Generate PTX at runtime
char* generatePTX(int config) {
    char* ptx = malloc(10000);
    sprintf(ptx, ".version 7.0\n"
                 ".target sm_80\n"
                 ".entry dynamicKernel(...) {\n"
                 "   // Generated code based on config\n"
                 "}\n");
    return ptx;
}

// Load and execute
CUmodule module;
char* ptx = generatePTX(userConfig);
```

Performance Considerations

Runtime API Performance

```
// Slight overhead but usually negligible
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(data);  // ~1-2μs overhead
cudaEventRecord(stop);
```

Driver API Performance

```
// Minimal overhead, maximum control
CUevent start, stop;
cuEventCreate(&start, CU_EVENT_DEFAULT);
cuEventCreate(&stop, CU_EVENT_DEFAULT);

cuEventRecord(start, stream);
cuLaunchKernel(kernel, ...);  // <1μs overhead
cuEventRecord(stop, stream);
```

Performance Impact

- Runtime API: 1-2% overhead typical
- Driver API: <1% overhead

Mixing APIs

Using Both APIs Together

```
// Start with Runtime API
cudaSetDevice(0);
float* d_data;
cudaMalloc(&d_data, size);

// Get Driver API context
CUcontext cuContext;
cuCtxGetCurrent(&cuContext);

// Load custom module with Driver API
CUmodule module;
cuModuleLoad(&module, "custom.ptx");

CUfunction specialKernel;
cuModuleGetFunction(&specialKernel, module, "special");

// Launch Driver API kernel on Runtime API memory
void* args[] = {&d_data};
cuLaunchKernel(specialKernel, grid, 1, 1,
```

Best Practice

Use Runtime API by default. Driver API only when needed

고급 Driver API 기능

1. 동적 커널 생성 (JIT 컴파일)

```
// JIT 컴파일 옵션 설정
void jitCompileKernel(const char *source) {
    CUmodule module;
    CUjit_option options[] = { /* ... */ };
    void *optionValues[8];

    // JIT 컴파일
    CUresult result = cuModuleLoadDataEx(&module, source,
                                         8, options, optionValues);

    if (result != CUDA_SUCCESS) {
        printf("JIT compilation failed\n");
    }
}
```

런타임 코드 생성

```
class DynamicKernelGenerator {
private:
    std::stringstream ptxCode;
public:
    void generateKernel(int unrollFactor, bool useFMA);
    std::string getPTX() const { return ptxCode.str(); }
};
```

실습 파일: `jit_compilation.cu`

2. 멀티 컨텍스트 관리

```
class MultiContextManager {
private:
    std::vector<CUcontext> contexts;
    std::vector<CUstream> streams;
    int numDevices;

public:
    void initialize() {
        cuInit(0);
        cuDeviceGetCount(&numDevices);
        // 각 디바이스에 컨텍스트 생성
    }
};
```

실습 파일: `multi_context_manager.cu`

Production CUDA Project Structure

Best Practices for Large-Scale Development

Standard Directory Layout

```
cuda-project/
├── CMakeLists.txt           # Build configuration
├── README.md                # Project documentation
├── include/                 # Header files
│   ├── kernels.cuh         # CUDA kernel headers
│   ├── utils.h             # Utility functions
│   └── error_check.h       # Error checking macros
├── src/                     # Source files
│   ├── main.cpp            # Main entry point
│   ├── kernels.cu          # CUDA kernel implementations
│   └── utils.cpp           # Utility implementations
├── tests/                   # Test code
│   ├── test_kernels.cu     # Kernel unit tests
│   └── benchmark.cpp       # Performance benchmarks
├── data/                    # Test data
├── docs/                    # Documentation
└── build/                   # Build output
```

File Organization Patterns

Header Files (.cuh, .h)

```
// include/kernels.cuh
#ifndef KERNELS_CUH
#define KERNELS_CUH

#include <cuda_runtime.h>

// Kernel declarations
__global__ void vectorAdd(float* a, float* b, float* c, int n);
__global__ void matrixMul(float* a, float* b, float* c, int m, int n, int k);

// Host wrapper functions
```

Implementation Files (.cu)

```
// src/kernels.cu
#include "kernels.cuh"
#include "error_check.h"

__global__ void vectorAdd(float* a, float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) c[idx] = a[idx] + b[idx];
}
```

Error Handling Infrastructure

Error Check Macros

```
// include/error_check.h
#ifndef ERROR_CHECK_H
#define ERROR_CHECK_H

#include <cstdio>
#include <cuda_runtime.h>

#define CHECK_CUDA(call) do { \
    cudaError_t error = call; \
    if (error != cudaSuccess) { \
        fprintf(stderr, "CUDA error at %s:%d - %s\n", \
                __FILE__, __LINE__, cudaGetErrorString(error)); \
        exit(1); \
    } \
} while(0)

#define CHECK_LAST_CUDA_ERROR() CHECK_CUDA(cudaGetLastError())

// For kernel launches
#define KERNEL_LAUNCH(kernel, grid, block, ...) do { \
    kernel<<<grid, block>>>(__VA_ARGS__); \
    CHECK_LAST_CUDA_ERROR(); \
```

CMake Configuration

Professional CMakeLists.txt

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

# CUDA settings
set(CMAKE_CUDA_STANDARD 17)
set(CMAKE_CUDA_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 17)

# Find CUDA
find_package(CUDAToolkit REQUIRED)

# Detect GPU architecture
include(FindCUDA/select_compute_arch)
CUDA_SELECT_NVCC_ARCH_FLAGS(ARCH_FLAGS Auto)
set(CMAKE_CUDA_ARCHITECTURES ${ARCH_FLAGS})

# Include directories
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)

# Source files
file(GLOB CUDA_SOURCES src/*.cu)
```

Testing Framework

Unit Test Structure

```
// tests/test_kernels.cu
#include <gtest/gtest.h>
#include "kernels.cuh"

class KernelTest : public ::testing::Test {
protected:
    void SetUp() override {
        cudaSetDevice(0);
        size = 1024;
        bytes = size * sizeof(float);

        // Allocate memory
        cudaMalloc(&d_a, bytes);
        cudaMalloc(&d_b, bytes);
        cudaMalloc(&d_c, bytes);

        h_a = new float[size];
        h_b = new float[size];
        h_c = new float[size];
    }

    void TearDown() override {
```

Build and Deployment

Build Script

```
#!/bin/bash
# build.sh

# Create build directory
mkdir -p build
cd build

# Configure with CMake
cmake .. \
    -DCMAKE_BUILD_TYPE=Release \
```

Docker Deployment

```
FROM nvidia/cuda:11.8.0-devel-ubuntu22.04

# Install dependencies
RUN apt-get update && apt-get install -y \
  cmake \
  build-essential \
  git

# Copy project
```

CMake 설정

CUDA 프로젝트를 위한 CMake 설정입니다.

CMakeLists.txt 예시

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

# CUDA 설정
set(CMAKE_CUDA_STANDARD 17)
set(CMAKE_CUDA_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 17)

# CUDA 아키텍처 설정 (GPU 모델에 맞게)
set(CMAKE_CUDA_ARCHITECTURES 70 75 80 86)

# 소스 파일
set(CUDA_SOURCES
    src/kernels.cu
    src/utils.cu
)

set(CPP_SOURCES
    src/main.cpp
```

Docker/Kubernetes GPU 배포

Docker GPU 지원

NVIDIA Container Toolkit 설정

```
# docker-compose.yml
services:
  cuda-app:
    image: nvidia/cuda:12.0-runtime-ubuntu22.04
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1    # GPU 개수
              capabilities: [gpu]
    environment:
```

실행 명령

```
# GPU 포함 컨테이너 실행
docker run --gpus all nvidia/cuda:12.0-base nvidia-smi

# 특정 GPU만 사용
docker run --gpus '"device=0,1"' myapp
```

Kubernetes GPU 스케줄링

GPU 리소스 관리

Pod 정의 (GPU 요청)

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
  - name: cuda-container
    image: myapp:cuda
```

Node Feature Discovery

```
# GPU 노드 라벨링
nodeSelector:
  accelerator: nvidia-tesla-v100

# GPU 성능별 스케줄링
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
```

프로덕션 GPU 모니터링

필수 모니터링 메트릭

DCGM (Data Center GPU Manager)

```
# Prometheus 연동
- job_name: 'dcgm'
  static_configs:
```

주요 메트릭

메트릭	의미	임계값
GPU Utilization	SM 사용률	> 90% 경고
Memory Usage	VRAM 사용량	> 95% 위험
Temperature	GPU 온도	> 83°C 스로틀링
Power Draw	전력 소비	TDP 대비 %
PCIe Throughput	데이터 전송	< 50% 비효율

Grafana 대시보드

```
{
  "dashboard": {
    "panels": [
      {"target": "dcgm_gpu_utilization"},
      {"target": "dcgm_memory_used"},
      {"target": "dcgm_sm_clock"}
    ]
```

Production Deployment Checklist

Comprehensive Pre-Release Validation

Code Quality Assurance

- [ ] All CUDA API calls have error checking
- [ ] Memory leak testing with cuda-memcheck
- [ ] Boundary checks and array index validation
- [ ] Race condition detection with racecheck
- [ ] Thread synchronization correctness
- [ ] Proper resource cleanup in all paths

Performance Validation

- [ ] Occupancy analysis and optimization (>50%)
- [ ] Memory coalescing verification (>90%)
- [ ] Bank conflict elimination
- [ ] Unnecessary synchronization removal
- [ ] Kernel launch configuration optimization
- [ ] Stream utilization for overlap

Memory Safety Checklist

Memory Validation Tools

```
# Complete memory check suite
cuda-memcheck --leak-check full ./application
cuda-memcheck --racecheck ./application
cuda-memcheck --initcheck ./application
cuda-memcheck --synccheck ./application

# Sanitizer for newer GPUs (Ampere+)
compute-sanitizer --tool memcheck ./application
```

Common Memory Issues

```
// BAD: No boundary check
__global__ void unsafe_kernel(float* data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    data[idx] = idx;  // Can overflow!
}

// GOOD: With boundary check
__global__ void safe_kernel(float* data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {  // Boundary check
        data[idx] = idx;
```

Performance Benchmarking

Performance Metrics to Track

```
// Comprehensive timing framework
class CUDATimer {
    cudaEvent_t start, stop;
public:
    CUDATimer() {
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
    }

    void startTimer() {
        cudaEventRecord(start);
    }

    float stopTimer() {
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        float ms = 0;
        cudaEventElapsedTime(&ms, start, stop);
        return ms;
    }

    ~CUDATimer() {
```

Compatibility Matrix

Multi-GPU Architecture Support

```
// Build for multiple architectures
// CMakeLists.txt
set(CMAKE_CUDA_ARCHITECTURES "60;70;75;80;86;89;90")

// Or in code with runtime compilation
void compileForCurrentGPU() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);

    char flag[256];
```

Version Requirements

```
// Check CUDA version at runtime
int runtimeVersion, driverVersion;
cudaRuntimeGetVersion(&runtimeVersion);
cudaDriverGetVersion(&driverVersion);

if(runtimeVersion < 11080) {  // CUDA 11.8
    fprintf(stderr, "Requires CUDA 11.8 or newer\n");
    exit(1);
}
```

Testing Strategy

Comprehensive Test Suite

```
// Test different data sizes
void testScalability() {
    int sizes[] = {1024, 1048576, 16777216, 268435456};

    for(int size : sizes) {
        printf("Testing size: %d\n", size);

        // Allocate and test
        float* data;
        cudaMalloc(&data, size * sizeof(float));

        // Run kernel
        kernel<<<getGridSize(size), 256>>>(data, size);

        // Verify results
        verifyResults(data, size);

        cudaFree(data);
    }
}

// Edge case testing
```

Documentation Requirements

Essential Documentation

```
# CUDA Application Documentation

## System Requirements
- CUDA Toolkit: 11.8 or newer
- GPU: Compute Capability 7.0+ (Volta or newer)
- Driver: 520.61.05 or newer
- Memory: Minimum 4GB GPU memory

## Build Instructions
\`\`\`bash
mkdir build && cd build
cmake .. -DCMAKE_CUDA_ARCHITECTURES="70;75;80;86"
make -j$(nproc)
\`\`\`

## Performance Tuning
- Block size: 256 (optimal for most kernels)
- Grid size: 2 * multiprocessor count
- Shared memory: 48KB per block maximum

## Benchmarks
```

Final Validation Steps

Pre-Release Checklist

```
#!/bin/bash
# release-validation.sh

echo "=== CUDA Production Validation ==="

# 1. Memory checks
echo "Running memory checks..."
cuda-memcheck --leak-check full ./app || exit 1
cuda-memcheck --racecheck ./app || exit 1

# 2. Performance benchmarks
echo "Running benchmarks..."
./benchmark --full || exit 1

# 3. Unit tests
echo "Running unit tests..."
ctest --output-on-failure || exit 1
```

최종 승인 기준

- 메모리 누수 없음
- 레이스 컨디션 없음
- 목표 성능의 10% 이내

Part 3. 요약

이 장에서 우리는 다음을 배웠습니다:

1. **디버깅과 커널 설계 전략**
    - CUDA 에러 처리, `printf` 디버깅, `cuda-memcheck`
    - 점진적 개발, 커널 설계 패턴, 일반적인 버그 해결
2. **다양한 병렬화 기법**
    - Task, Data, Pipeline, Dynamic Parallelism
    - Cooperative Groups, Persistent Threads, Adaptive Parallelism
3. **커널 퓨전과 커널 분할**
    - 수직/수평 퓨전, 메모리/점유율 최적화를 위한 분할
4. **CUDA 그래프와 실행 최적화**
    - Stream Capture, 명시적 API를 통한 그래프 생성
    - 조건부 실행, 동적 그래프 업데이트
5. **런타임 API vs 드라이버 API**
    - 두 API 레벨의 비교와 Driver API 기초

> **다음 장 예고:** Part 4에서는 CUDA의 최신 기능과 고급 주제들을 다룹니다. 멀티 GPU 프로그래밍, CUDA 라이브러리 활용, 그리고 최신 GPU 아키텍처에 대한 깊이 있는 내용을 배우게 됩니다.