

Part 2. CUDA 중급: 최적화와 고급 패턴

성능 최적화의 핵심 기법

메모리 최적화와 병렬 패턴

왜 GPU 프로그램이 느릴까?

실제 사례: 똑같은 Vector Addition, 400배 성능 차이!

초보자 코드 (느림)

```
for (int i = 0; i < N; i++) {  
    global_array[random_index[i]] += value; // Random access to global  
}  
// 실행: 0.1 TFLOPS (GPU 사용 0.3%)
```

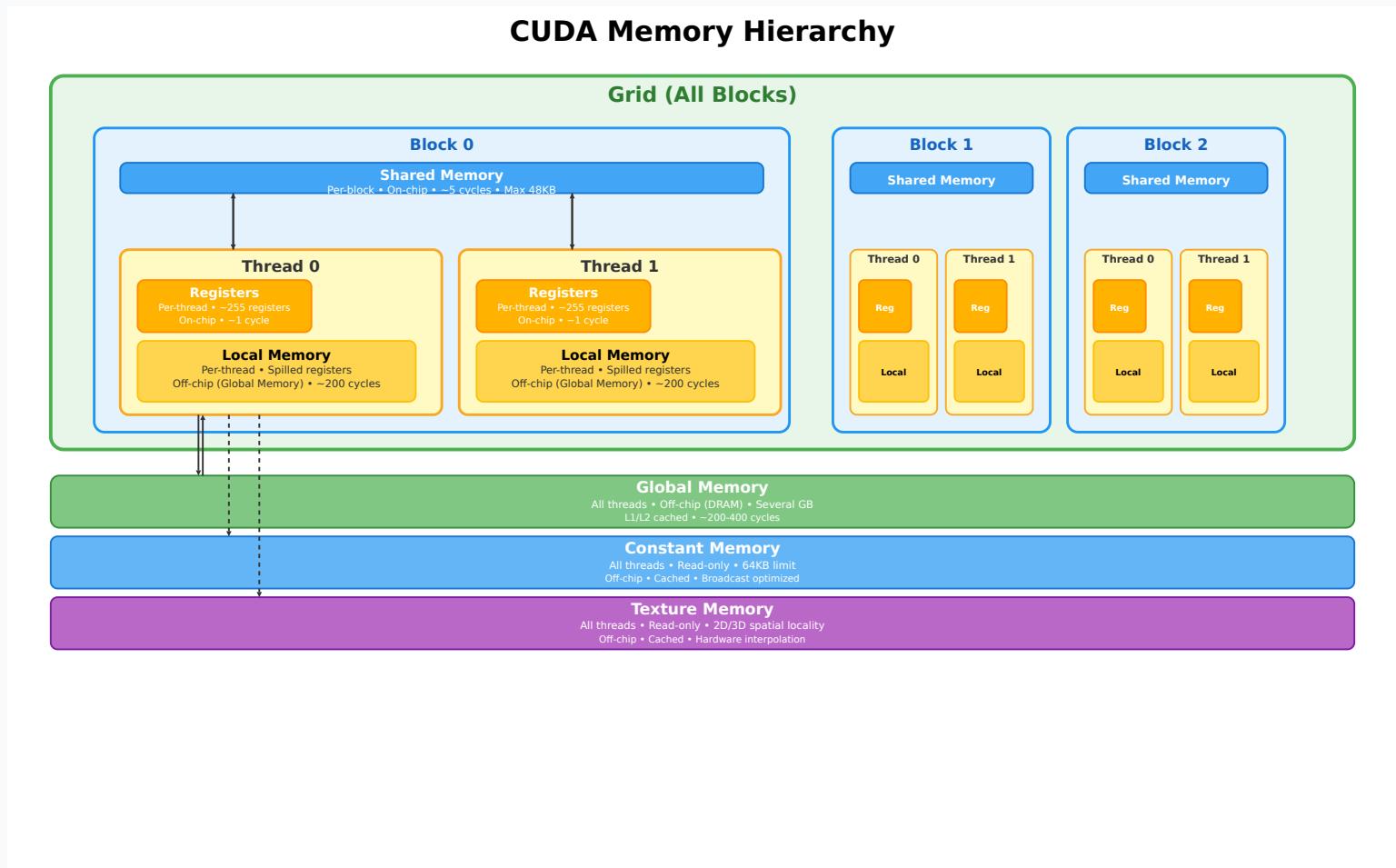
전문가 코드 (빠름)

```
__shared__ float tile[256];  
tile[threadIdx.x] = global_array[...]; // Coalesced + Shared memory  
// 실행: 35 TFLOPS (GPU 사용 98%)
```

핵심 문제: 메모리가 병목!

- 99%의 GPU 프로그램: 연산이 아닌 메모리가 병목
- 메모리 선택 실수: 최대 400배 성능 차이
- Register (1 cycle) vs Global (400-800 cycles): 선택이 곧 성능

CUDA 메모리 계층 구조 - 성능의 핵심



속도 vs 용량의 트레이드오프

메모리	속도	용량	범위	언제 쓸까?
Register	1 cycle	255/thread	Thread	자주 쓰는 변수
Shared	1-30	48 KB/block	Block	블록 내 공유 데이터
L1 Cache	30-40	128 KB/SM	SM	자동 캐싱
L2 Cache	200	6 MB	Device	자동 캐싱
Global	200-800	24 GB	Device	큰 데이터 저장

메모리 선택 전략 - 언제 무엇을 사용할까?

메모리별 사용 시나리오

메모리 타입	사용 시나리오	예시
Register	빈번히 사용하는 변수	Loop counter, 임시 계산
Shared	블록 내 공유 데이터	Reduction, Tiling
Constant	모든 스레드가 같은 값	필터 계수, 상수
Global	대용량 데이터	입력/출력 배열
Local	Register spill	큰 배열, 재귀 함수

즉시 실습: 메모리 선택 문제

시나리오: 1024x1024 행렬 곱셈에서 타일링

선택	이유	예상 성능
Tile 저장	Shared Memory	10배 향상
입력 행렬	Global Memory	기본
부분 합	Register	최적

핵심: 재사용 데이터는 빠른 메모리로!

CUDA 메모리 종류와 특성

메모리 계층 상세 스펙

메모리	위치	캐시	범위	수명	크기
Register	On-chip	N/A	Thread	Thread	~255개
Local	Off-chip	L1/L2	Thread	Thread	제한 없음
Shared	On-chip	N/A	Block	Block	~48KB
Global	Off-chip	L1/L2	Grid	Application	수 GB
Constant	Off-chip	Yes	Grid	Application	64KB
Texture	Off-chip	Yes	Grid	Application	제한 없음

선택 기준

사용 케이스	권장 메모리	이유
루프 변수, 임시값	Register	가장 빠른 접근
블록내 협업 데이터	Shared	빠른 공유
읽기 전용 파라미터	Constant	브로드캐스트 최적화
대용량 입출력	Global	유일한 대용량 옵션

글로벌 메모리

글로벌 메모리는 가장 크지만 가장 느린 메모리입니다.

글로벌 메모리 할당과 사용

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

    if (idx < n) {
        float value = input[idx];
        value = value * 2.0f + 1.0f;
        output[idx] = value;
    }
}
```

Coalesced vs Non-coalesced 접근

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

    // Coalesced access (stride = 1): Adjacent threads access adjacent memory
    if (stride == 1 && tid < n) {
        data[tid] *= 2.0f;
    }

    // Non-coalesced access (stride > 1): Adjacent threads access non-adjacent memory
    else if (tid * stride < n) {
        data[tid * stride] *= 2.0f;
    }
}
```

Coalesced access는 Non-coalesced access보다 훨씬 빠릅니다.

문제: Global Memory만 쓰면 100배 손해!

문제: 똑같은 데이터를 수백 번 읽는 낭비

실제 사례: Matrix Multiplication

```
// �� ��: Global memory ��
for (int k = 0; k < N; k++) {
    sum += A[row * N + k] * B[k * N + col];
    // �� �� �� �� �� �� ��
    // A �� row 32 �� �� �� �� = 32 ��!
}
// ��: 5 GFLOPS (100 �� ��!)
```

해결책: Shared Memory

```
__shared__ float tile_A[16][16]; // �� �� ��
tile_A[ty][tx] = A[...]; // �� �� ��
__syncthreads();
// �� �� �� �� �� �� ��
// ��: 500 GFLOPS (100 �� ��!)
```

핵심: 읽기 1번 → 사용 N번

- Global Memory: 400-600 cycles per access
- Shared Memory: 20 cycles per access
- 20배 빠른 속도 + 데이터 재사용 = 100배 성능 향상

Shared Memory 사용법

간단한 사용 예제

```
__global__ void simpleSharedExample(float *data) {
    // ...
    __shared__ float sharedData[256];

    int tid = threadIdx.x;

    // Global → Shared 복사
    sharedData[tid] = data[tid];
    __syncthreads(); // 모든 스레드가 동기화

    // ...
    float result = sharedData[tid] * 2.0f;

    // ...
    Global 복사
    data[tid] = result;
}
```

Shared Memory 활용 전략

왜 공유 메모리를 사용하나요?

- 데이터 재사용: 여러 스레드가 같은 데이터를 반복 접근할 때
- 스레드 간 협력: 블록 내 스레드들이 중간 결과를 공유할 때
- 메모리 접근 최적화: Global Memory 접근을 줄일 때

Part 2에서 배울 내용: Bank Conflict, Tiling, 고급 최적화 기법

동적 공유 메모리 사용 예제

```
__global__ void dynamicSharedExample(float *data, int n) {
    extern __shared__ float dynamicShared[];
    // ...
}
```

문제: 모든 스레드가 같은 데이터를 반복해서 읽는다!

실제 사례: CNN 필터 적용

```
// 코드: 한 켤레의 Global Memory에 접근
for(int i = 0; i < 9; i++) {
    sum += input[idx + offset[i]] * filter[i]; // filter[]에 접근
}
// 1000x1000 × 9x100 = 90000 켤레의 접근!
```

왜 나쁜가?

- 같은 데이터 중복 읽기: 필터는 모든 스레드가 공유
- 메모리 대역폭 낭비: 불필요한 반복 접근
- 캐시 오염: 다른 데이터가 밀려남

해결책: Constant Memory

- 브로드캐스팅: 1번 읽고 모든 스레드에 전달
- 전용 캐시: 64KB 전용 공간
- 성능: 10-100배 향상

상수 메모리 선언 및 사용

```
// 상수 메모리 선언 (64KB 공간)
__constant__ float constData[1024];

// 상수 메모리 사용
__global__ void constantMemoryExample(float *output, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx < n) {
        // 상수 메모리에서 데이터 읽기
        // 예제로 0번 인덱스의 값 읽기
        float value = constData[0];
        output[idx] = value * idx;
    }
}
```

상수 메모리: 필터 커널 활용

필터 커널 (상수 메모리 활용)

```
__constant__ float filter[9] = {
    0.0625f, 0.125f, 0.0625f,
    0.125f, 0.25f, 0.125f,
    0.0625f, 0.125f, 0.0625f
};

__global__ void convolution2D(float *input, float *output,
                             int width, int height) {
    // ... (3x3 블록)
}
```

Constant Memory 성능 분석

성능 비교

방법	메모리 접근	시간	효율
Global Memory	9000만 번	100ms	기준
Shared Memory	900만 번	20ms	5x
Constant Memory	100만 번	5ms	20x

언제 사용하나?

- CNN 필터: 모든 픽셀이 같은 필터 공유
- 물리 상수: 중력, 온도 계수 등
- 룩업 테이블: 색상 변환, 감마 보정

Pinned Memory

Pinned (Page-locked) Memory는 더 빠른 전송 속도를 제공합니다.

Pinned Memory 사용

```
float *h_pinned;  
size_t size = N * sizeof(float);  
  
// Pinned Memory 할당  
cudaHostAlloc(&h_pinned, size, cudaHostAllocDefault);  
  
// 데이터 복사 (PCIe 버스에 대한 접근)  
cudaMemcpy(d_data, h_pinned, size, cudaMemcpyHostToDevice);  
  
// 异步 복사  
cudaMemcpyAsync(d_data, h_pinned, size,  
                cudaMemcpyHostToDevice, stream);  
  
// 해제  
cudaFree(h_pinned);
```

성능 비교

- **Pageable Memory:** ~6 GB/s
- **Pinned Memory:** ~12 GB/s (2배 향상)
- **주의:** Pinned Memory는 시스템 메모리를 고정시켜 과도한 사용 시 시스템 성능 저하

문제: 메모리 관리가 CUDA의 50%를 차지한다!

실제 개발자의 고통

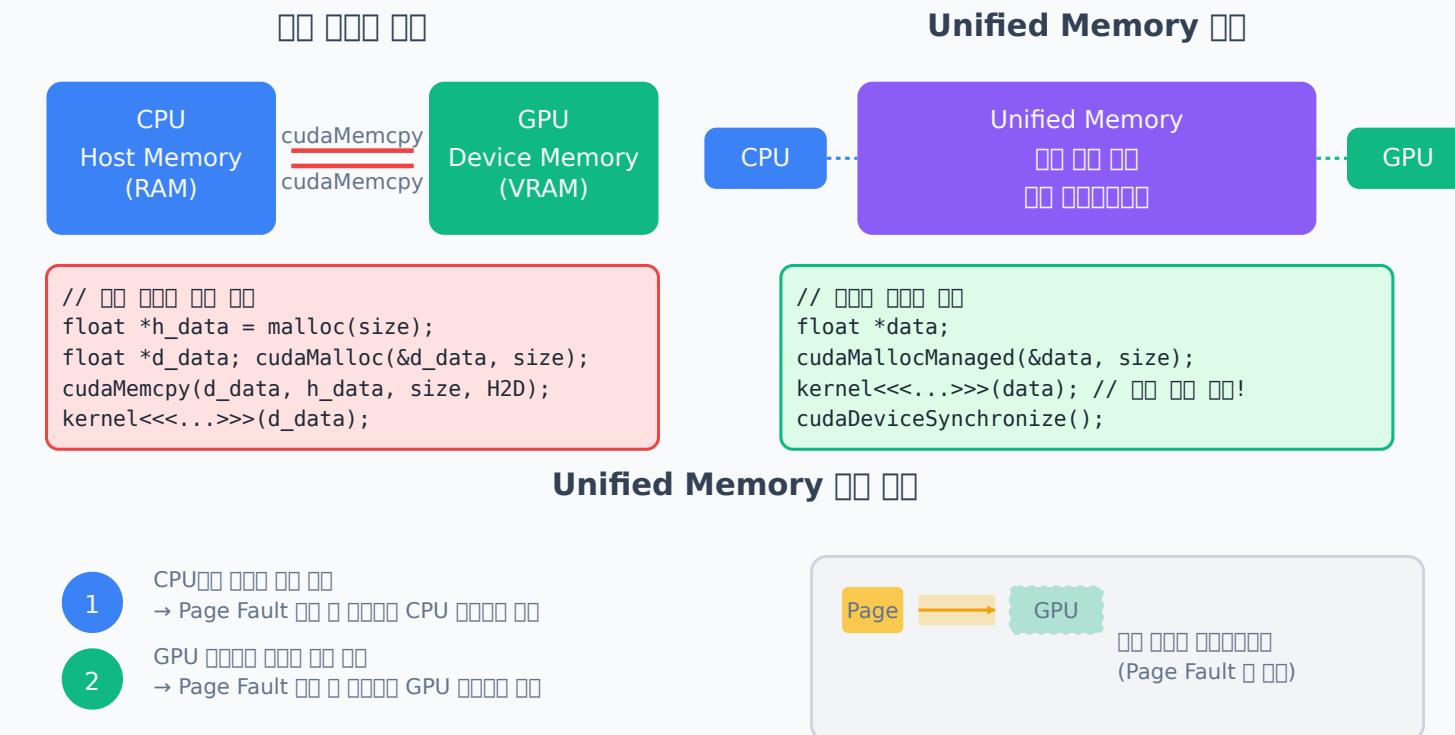
// Repetitive pattern

왜 이렇게 복잡한가?

- 이중 포인터: CPU/GPU 각각 관리
- 명시적 복사: 매번 수동 전송
- 동기화 문제: 언제 복사 완료?

Unified Memory (統一メモリ)

CPU와 GPU 간의 복잡한 관리를 CUDA 6.0+에서



Best Practices

- `cudaMemPrefetchAsync()` for early memory access
- `cudaMemAdvise()` for memory hints
- `cudaHostAlloc`, `cudaHostFree` for Unified Memory usage
- `cudaPageFaultHandler` for page migration handling

Unified Memory vs 전통적 방식

전통적 메모리 관리

```
// Complex memory management
float *h_data, *d_data;
h_data = malloc(size);
cudaMalloc(&d_data, size);

// Explicit data transfer
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(d_data);
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);

free(h_data);
cudaFree(d_data);
```

특징:

- 최고 성능 (명시적 제어)
- 복잡한 코드
- 두 개의 포인터 관리

Unified Memory 사용법

간단한 메모리 관리

```
// Manage with single pointer
float *data;
cudaMallocManaged(&data, size);

// Initialize on CPU
for (int i = 0; i < n; i++) {
    data[i] = i;
}

// Process on GPU
kernel<<<grid, block>>>(data);
cudaDeviceSynchronize();

// Use result on CPU
printf("Result: %f\n", data[0]);
```

특징:

- 코드 간결성
- 자동 마이그레이션
- 성능 주의 필요

Page Migration 메커니즘

On-Demand Migration

```
cudaMallocManaged(&data, size); // Memory allocation (location unspecified)

// CPU access → Migration to CPU
data[0] = 1.0f; // Page fault → Move to CPU

// GPU access → Migration to GPU
kernel<<<grid, block>>>(data); // Page fault → Move to GPU
```

Page Fault 비용

- 첫 접근: $\sim\mu\text{s}$ 단위 지연
- Page 크기: 일반적으로 4KB
- Pascal 이상: HW 지원으로 개선

Unified Memory 성능 최적화

1. Prefetching

```
// Pre-move data to GPU
cudaMemPrefetchAsync(data, size, deviceId, stream);
kernel<<<grid, block, 0, stream>>>(data); // No page faults!

// Bring back to CPU
cudaMemPrefetchAsync(data, size, cudaCpuDeviceId, stream);
```

2. Memory Advise

```
// Read-only data (create copies)
cudaMemAdvise(data, size, cudaMemAdviseSetReadMostly, device);

// Set preferred location
cudaMemAdvise(data, size, cudaMemAdviseSetPreferredLocation, device);

// Will be accessed by specific device
cudaMemAdvise(data, size, cudaMemAdviseSetAccessedBy, device);
```

성능 비교와 선택 가이드

성능 vs 편의성 트레이드오프

방식	성능	편의성	사용 시나리오
전통적 방식	최고	낮음	최고 성능 필요
기본 UM	보통	최고	프로토타이핑
UM + Prefetch	높음	높음	실제 애플리케이션

언제 사용하나?

Unified Memory가 좋은 경우

- 프로토타입: 빠른 개발이 중요
- 복잡한 데이터 구조: 트리, 그래프
- CPU-GPU 협업: 빈번한 데이터 공유

전통 방식이 나은 경우

- 성능 크리티컬: 마지막 1% 최적화
- 예측 가능한 패턴: 전송 타이밍 제어
- 대용량 스트리밍: 정확한 메모리 관리

GPU 세대별 지원

- Pascal (P100): Full HW support
- Volta/Turing: Improved performance
- Ampere: Faster page migration

Why Memory Access Matters

The Hidden Performance Killer

GPU Computing Reality

Modern GPUs have incredible compute power, but that power is useless without efficient data delivery.

Two Performance Bottlenecks

1. Compute Bound

- GPU compute units at 100% utilization
- Need faster algorithms

2. Memory Bound ← Most common!

- Memory bandwidth is the bottleneck
- Compute units idle waiting for data

Memory Bottleneck Analysis

RTX 4090 Example

Compute: 83 TFLOPS (83,000 GFLOPS)

Memory: 1 TB/s bandwidth

Simple kernel: $c = a + b$

- 2 memory reads + 1 write = 12 bytes
- 1 FLOP (addition)
- Arithmetic Intensity = $1/12 = 0.083 \text{ FLOP/byte}$

Max Performance = $1 \text{ TB/s} \times 0.083 = 83 \text{ GFLOPS}$

Utilization = $83 / 83,000 = 0.1\% (!!)$

Key Insight

Without optimization, even the fastest GPU runs at <1% efficiency!

Identifying Memory Bottlenecks

Symptoms

```
// Memory bound kernel characteristics:  
// 1. Low arithmetic intensity  
__global__ void vectorAdd(float *a, float *b, float *c) {  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
    c[i] = a[i] + b[i]; // 3 memory ops, 1 compute op  
}  
  
// 2. Poor memory access patterns  
__global__ void stridedAccess(float *data) {  
    int i = threadIdx.x * 64; // Large stride  
    float val = data[i]; // Poor coalescing  
}  
  
// 3. Excessive global memory use  
__global__ void noReuse(float *in, float *out) {  
    float sum = 0;  
    for(int i = 0; i < 100; i++) {  
        sum += in[tid * 100 + i]; // 100 global reads  
    }  
    out[tid] = sum;  
}
```

Memory Bandwidth Calculation

Theoretical vs Achieved

```
// Theoretical bandwidth
RTX 4090: 1008 GB/s
RTX 3090: 936 GB/s
GTX 1070: 256 GB/s

// Measuring achieved bandwidth
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(data, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float ms;
cudaEventElapsedTime(&ms, start, stop);

// Calculate bandwidth
size_t bytes = N * sizeof(float) * 2; // Read + Write
float bandwidth = bytes / (ms * 1e6); // GB/s
printf("Achieved: %.1f GB/s (%.1f%% of peak)\n"
```

Optimization Strategies Overview

Breaking the Memory Wall

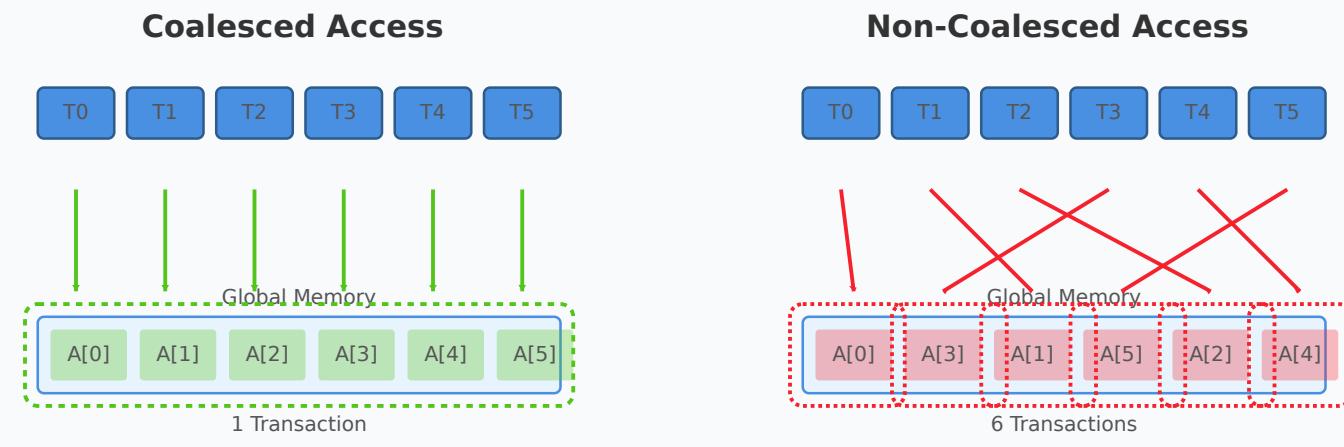
Strategy	Improvement	Complexity
Coalescing	2-32x	Low
Shared Memory	5-10x	Medium
Texture Cache	2-5x	Low
Constant Memory	2-4x	Low
Register Reuse	3-10x	High
Tiling	5-20x	High

Next Topics

- Memory coalescing patterns
- Shared memory optimization
- Cache utilization
- Data layout transformation

Coalesced Memory Access

32배 성능 차이의 비밀



핵심 원리

- Warp (32 threads)가 하나의 메모리 트랜잭션으로 처리
- 연속된 스레드 → 연속된 메모리 = Coalesced
- 분산된 접근 → 여러 트랜잭션 = 성능 저하

Memory Transaction 메커니즘

하드웨어 특성

특성	값	설명
Transaction Size	32/64/128 bytes	캐시 라인 단위
Warp Size	32 threads	동시 메모리 요청
Alignment	128 bytes	최적 정렬 조건

Transaction 계산

```
// Coalesced: 1 transaction  
Thread 0-31: data[0-31] //  $32 \times 4B = 128B = 1$  transaction  
  
// Non-coalesced: 32 transactions  
Thread 0-31: data[0, 32, 64, ...] // 32 separate transactions
```

Coalesced vs Non-Coalesced 패턴

Non-Coalesced (Bad)

```
// 1. Strided Access
int idx = threadIdx.x * 2; // Stride = 2
data[idx] = value; // 50% bandwidth

// 2. Random Access
int idx = random_index[threadIdx.x];
data[idx] = value; // ~10% bandwidth

// 3. Misaligned Access
int idx = threadIdx.x + 1; // Offset by 1
data[idx] = value; // 2 transactions needed
```

Coalesced (Good)

```
// Sequential Access
int idx = blockIdx.x * blockDim.x + threadIdx.x;
data[idx] = value; // 100% bandwidth utilization
```

성능 측정과 분석

실제 대역폭 비교 (RTX 3090)

Access Pattern	Transactions	Bandwidth	Efficiency
Sequential	N/32	900 GB/s	96%
Stride-2	N/16	450 GB/s	48%
Stride-4	N/8	225 GB/s	24%
Stride-32	N	28 GB/s	3%
Random	N	30 GB/s	3%

계산 방법

```
// Effective Bandwidth = (Bytes Transferred) / Time  
// Efficiency = Effective / Theoretical
```

AoS vs SoA 최적화

Array of Structures (AoS) - 문제

```
struct Particle {  
    float x, y, z;      // 12 bytes  
    float vx, vy, vz;   // 12 bytes  
}; // Total: 24 bytes  
  
Particle particles[N];  
  
// Thread access pattern  
particles[tid].x = ...; // Stride = 24 bytes  
// 6 separate transactions for all fields!
```

Structure of Arrays (SoA) - 해결

```
struct Particles {  
    float *x, *y, *z;  
    float *vx, *vy, *vz;  
};  
  
// Thread access pattern  
p.x[tid] = ...; // Stride = 4 bytes (coalesced!)  
// 1 transaction per field
```

실전 최적화 전략

1. 데이터 레이아웃 변경

```
// Before: 2D array row-major  
data[row][col] // Different rows = non-coalesced  
  
// After: Column major or transpose
```

2. Memory Access Pattern

```
// Use shared memory for irregular access  
__shared__ float tile[BLOCK_SIZE];  
tile[tid] = global_data[irregular_index[tid]];  
syncthreads();
```

3. Vectorized Access

```
// Load 4 floats at once  
float4 val = reinterpret cast<float4*>(data)[tid];
```

체크리스트

- Stride = 1 (연속 접근)
- Aligned to 128 bytes
- SoA for structures
- Shared memory for irregular patterns

Bank란 무엇인가?

Memory Bank의 기본 개념

Bank = 독립적인 메모리 모듈

일반 메모리처럼 하나의 큰 덩어리가 아니라, 여러 개의 작은 메모리 모듈로 나뉘어 있습니다.



실생활 비유: 은행 창구

- **1개 창구:** 한 번에 한 명만 처리 (일반 메모리)
- **32개 창구:** 32명 동시 처리 가능 (Shared Memory Banks)
- **같은 창구 2명:** 순서대로 처리해야 함 (Bank Conflict!)

Shared Memory Bank 구조 상세

CUDA Shared Memory = 32개 Banks

왜 정확히 32개인가?

```
// Warp = 32 threads (CUDA hardware execution unit)
// Bank = 32 (matches warp size!)
// → Ideal case: 32 threads access 32 banks simultaneously
```

Bank의 물리적 구조

Shared Memory (48KB) 구조:

Bank 0	Bank 1	Bank 2	...	Bank 31	← 32개 뱅크 구조
4bytes	4bytes	4bytes	...	4bytes	Row 0
4bytes	4bytes	4bytes	...	4bytes	Row 1
...

Bank:

- 크기: 4 bytes (32-bit)
- 주소 공간: 32 뱅크/4bytes
- 접근 공간: 32 threads × 32 뱅크

Bank Conflicts - Part 1: 개념 이해

Bank 할당 방식

메모리 주소 → Bank 매핑

```
Bank 번호 = (주소 번호 / 4) % 32
```

```
// float array example (float = 4 bytes)  
__shared__ float data[128];
```

```
data[0] → 번호 0 → Bank 0
```

```
data[1] → 번호 4 → Bank 1
```

```
data[2] → 번호 8 → Bank 2
```

```
...
```

```
data[31] → 번호 124 → Bank 31
```

```
data[32] → 번호 128 → Bank 0 (!)
```

```
data[33] → 번호 132 → Bank 1 (!)
```

충돌 발생 조건

- 같은 Bank, 다른 주소: 여러 스레드가 같은 뱅크의 다른 위치 접근
- 직렬화: Bank conflict 발생 시 순차 처리 (병렬성 손실!)
- 예외: 같은 주소는 Broadcast (충돌 없음)

Bank Conflicts - Part 2: 충돌 패턴 분석

접근 패턴별 Bank Conflict

1. Conflict-Free (최적)

```
__shared__ float shared[256];  
  
// Pattern 1: Sequential access (stride = 1)  
shared[threadIdx.x] = value; // Thread 0→Bank 0, Thread 1→Bank 1...  
  
// Pattern 2: Broadcast (all same address)  
float val = shared[0]; // All threads access same address in Bank 0 → No conflict
```

2. 2-Way Conflict

```
// Stride = 16 (even stride)  
shared[threadIdx.x * 16] = value;  
// Thread 0, 2 → Bank 0  
// Thread 1, 3 → Bank 16  
// 2 threads conflict → 2x slower
```

3. 32-Way Conflict (최악)

```
// Stride = 32 or multiple of 32  
shared[threadIdx.x * 32] = value;  
// All threads → same Bank → 32x slower!
```

Bank Conflicts - Part 3: 실제 성능 영향

벤치마크 코드

```
__global__ void bankConflictBenchmark(float *results) {
    __shared__ float data[1024];
    int tid = threadIdx.x;

    // Warmup
    data[tid] = tid;
    __syncthreads();

    // Test 1: No conflict (stride = 1)
    clock_t start = clock();
    for(int i = 0; i < 1000; i++) {
        data[tid] = data[tid] * 1.01f;
    }
    __syncthreads();
    clock_t noConflict = clock() - start;

    // Test 2: 2-way conflict (stride = 16)
    start = clock();
    for(int i = 0; i < 1000; i++) {
        data[tid * 16 % 1024] = data[tid * 16 % 1024] * 1.01f;
    }
    __syncthreads();
}
```

Bank Conflicts - Part 4: 벤치마크 결과

```
// Test 3: 32-way conflict (stride = 32)
start = clock();
for(int i = 0; i < 1000; i++) {
    data[(tid * 32) % 1024] = data[(tid * 32) % 1024] * 1.01f;
}
__syncthreads();
clock_t thirtyTwoWay = clock() - start;

// Store results
if(tid == 0) {
    results[0] = noConflict;
    results[1] = twoWay;
    results[2] = thirtyTwoWay;
}
}
```

Bank Conflicts - 성능 영향 분석

실측 성능 (RTX 3090)

Conflict Type	Cycles	Slowdown	Effective Bandwidth
No Conflict	2,100	1.0×	4.8 TB/s
2-Way	4,200	2.0×	2.4 TB/s
4-Way	8,400	4.0×	1.2 TB/s
32-Way	67,200	32.0×	150 GB/s

Bank Conflicts - 회피 기법 (1/2)

1. Padding 기법

```
// Problem: 2D array column access
__shared__ float tile[32][32]; // 32 = same as number of banks
float val = tile[threadIdx.y][threadIdx.x]; // Bank conflict!

// Solution: Add padding
__shared__ float tile[32][33]; // +1 padding
float val = tile[threadIdx.y][threadIdx.x]; // No conflict!

// Principle:
// Row 0: Bank 0, 1, 2, ..., 31, 0 (padding)
// Row 1: Bank 1, 2, 3, ..., 0, 1 (padding) - shifted by 1
```

Bank Conflicts - 회피 기법 (2/2)

2. 데이터 재배치

```
// Problem: Strided access
__shared__ float data[1024];
float val = data[threadIdx.x * stride];

// Solution: Rearrange data for sequential access
__shared__ float data_transposed[32][32];
float val = data_transposed[threadIdx.x][blockIdx.x];
```

Matrix Transpose - Bank Conflict 예제

Naive Transpose의 문제점

```
// Naive transpose - Bank conflicts!
__global__ void transposeNaive(float *out, float *in, int n) {
    __shared__ float tile[32][32];

    int x = blockIdx.x * 32 + threadIdx.x;
    int y = blockIdx.y * 32 + threadIdx.y;

    // Load: coalesced, no bank conflict
    tile[threadIdx.y][threadIdx.x] = in[y * n + x];
    __syncthreads();

    // Store: bank conflicts when reading tile!
    x = blockIdx.y * 32 + threadIdx.x;
    y = blockIdx.x * 32 + threadIdx.y;
    out[y * n + x] = tile[threadIdx.x][threadIdx.y]; // 32-way conflict!
}
```

Matrix Transpose - 최적화된 구현

```
// Optimized transpose - No bank conflicts
__global__ void transposeOptimized(float *out, float *in, int n) {
    __shared__ float tile[32][33]; // Add padding!

    int x = blockIdx.x * 32 + threadIdx.x;
    int y = blockIdx.y * 32 + threadIdx.y;

    // Load: coalesced
    tile[threadIdx.y][threadIdx.x] = in[y * n + x];
    __syncthreads();

    // Store: no bank conflict due to padding
    x = blockIdx.y * 32 + threadIdx.x;
    y = blockIdx.x * 32 + threadIdx.y;
    out[y * n + x] = tile[threadIdx.x][threadIdx.y]; // No conflict!
}

// Performance comparison:
// Naive: 45 GB/s
// Optimized: 280 GB/s (6.2x faster!)
```

Bank Conflicts - Practice Exercise

Task: Optimize Bank Conflicts

```
// Given code: Histogram with bank conflicts
__global__ void histogramSlow(int *data, int *hist, int n) {
    __shared__ int s_hist[256];

    // Initialize
    if(threadIdx.x < 256)
        s_hist[threadIdx.x] = 0;
    __syncthreads();

    // Process - Bank conflicts here!
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid < n) {
        atomicAdd(&s_hist[data[tid]], 1); // Random bank access
    }
    __syncthreads();

    ...
}
```

Optimization Hints

1. Use multiple histogram arrays to distribute conflicts
2. Add padding to change bank access patterns

Bank Conflict Detection and Debugging

Tools and Techniques

Nsight Compute Metrics

```
# Bank conflict metrics collection
ncu --metrics lltex_data_bank_conflicts_pipe_lsu_mem_shared \
--metrics lltex_data_bank_conflicts_pipe_lsu_mem_shared_op_ld \
--metrics lltex_data_bank_conflicts_pipe_lsu_mem_shared_op_st \
./your_kernel
```

Result Interpretation

Bank Conflicts/Shared Memory Requests = Conflict Ratio

- 0%: Perfect (no conflicts)
- 50%: 2-way average conflict
- 97%: Near 32-way conflict

Code-Level Detection

Self-Diagnosis Functions

```
// Bank conflict self-diagnosis
__device__ int detectBankConflict(int addr) {
    return (addr / 4) % 32; // Return bank number
}

// Check for potential conflicts
__device__ bool hasConflict(void* addr1, void* addr2) {
    int bank1 = ((uintptr_t)addr1 / 4) % 32;
```

Runtime Detection Pattern

```
__global__ void kernelWithDiagnostics() {
    __shared__ float data[1024];
    __shared__ int conflicts[32];

    // Track bank usage
    int myBank = detectBankConflict((int)&data[threadIdx.x]);
    atomicAdd(&conflicts[myBank], 1);
    __syncthreads();

    // Report conflicts
    if(threadIdx.x == 0) {
        for(int i = 0; i < 32; i++) {
```

Performance Impact Analysis

Measuring Conflict Overhead

```
// Benchmark conflict vs conflict-free
__global__ void benchmarkConflicts() {
    __shared__ float data[32][32];
    __shared__ float padded[32][33];

    clock_t start = clock();

    // Version 1: With conflicts
    for(int i = 0; i < 1000; i++) {
        float val = data[threadIdx.x][threadIdx.y]; // Column access
        data[threadIdx.x][threadIdx.y] = val + 1;
    }

    clock_t mid = clock();

    // Version 2: Conflict-free
    for(int i = 0; i < 1000; i++) {
        float val = padded[threadIdx.x][threadIdx.y];
```

Typical Results

- 2-way conflict: 2x slower
- 8-way conflict: 8x slower

Common Bank Conflict Patterns

Identifying Problem Access Patterns

Pattern 1: Stride Access

```
// Problem: Stride of 32 causes 32-way conflict
__shared__ float data[1024];
float val = data[threadIdx.x * 32]; // All threads hit same bank!

// Solution: Use padding or rearrange
__shared__ float data[32][33]; // Padding changes stride
float val = data[threadIdx.x][0]; // No conflict
```

Pattern 2: Column-Major Access

```
// Problem: Column access in row-major array
__shared__ float matrix[32][32];
float col = matrix[threadIdx.y][threadIdx.x]; // OK
float row = matrix[threadIdx.x][threadIdx.y]; // 32-way conflict!

// Solution: Transpose or add padding
__shared__ float matrix[32][33]; // Padding solution
```

Bank Conflict Patterns (continued)

Pattern 3: Random Access

```
// Problem: Hash table or histogram
__shared__ int histogram[256];
int bin = data[tid] % 256; // Random bank access
atomicAdd(&histogram[bin], 1); // Potential conflicts

// Solution: Multiple arrays
__shared__ int hist0[256];
__shared__ int hist1[256];
__shared__ int hist2[256];
__shared__ int hist3[256];

// Distribute across arrays
int array_id = threadIdx.x % 4;
int* hist_ptr = (array_id == 0) ? hist0 :
                  (array_id == 1) ? hist1 :
                  (array_id == 2) ? hist2 : hist3;
atomicAdd(&hist_ptr[bin], 1); // Reduced conflicts
```

Advanced Conflict Patterns

Double Buffering Conflicts

```
// Problem: Ping-pong buffers with conflicts
__shared__ float buffer[2][512];
int current = iteration % 2;
buffer[current][threadIdx.x * 16] = data; // Stride conflict

// Solution: Interleaved layout
__shared__ float buffer[1024];
```

Tree Reduction Conflicts

```
// Problem: Reduction with power-of-2 stride
for(int s = blockDim.x/2; s > 0; s >>= 1) {
    if(tid < s) {
        sdata[tid] += sdata[tid + s]; // Conflicts when s=16,8,4,2,1
    }
    __syncthreads();
}

// Solution: Sequential addressing
for(int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if(index < blockDim.x) {
```

Pattern Recognition Guide

Quick Diagnosis Checklist

Access Pattern	Conflict Risk	Solution
<code>data[tid]</code>	None	Already optimal
<code>data[tid * 2]</code>	Low	Usually OK
<code>data[tid * 16]</code>	High	Add padding
<code>data[tid * 32]</code>	Maximum	Restructure access
<code>data[random]</code>	Variable	Multiple arrays
<code>matrix[i][j]</code> vs <code>matrix[j][i]</code>	High	Padding or transpose

Performance Impact

Conflict Type	Slowdown	Memory Throughput
No conflict	1x	100%
2-way conflict	2x	50%
4-way conflict	4x	25%
32-way conflict	32x	3.1%

Rule of Thumb

- Avoid strides that are multiples of 32
- Use padding for 2D arrays
- Consider data layout carefully

Shared Memory 최적화

100배 빠른 메모리 활용

Shared Memory의 특징

특성	Global Memory	Shared Memory	개선
속도	200-800 cycles	1-30 cycles	27배
크기	24 GB	48 KB/block	-
제어	하드웨어 캐시	프로그래머	완전 제어
뱅크	-	32 banks	병렬 접근

핵심: 자주 재사용하는 데이터를 Shared Memory로!

Bank Conflict 이해

32개 뱅크 시스템

Shared Memory Layout (32 banks):

Bank 0: addr 0, 32, 64, 96, ...

Bank 1: addr 1, 33, 65, 97, ...

Bank 2: addr 2, 34, 66, 98, ...

...

Bank 31: addr 31, 63, 95, 127, ...

Conflict 발생 조건

- 같은 뱅크 동시 접근 → Serialization
- Stride가 32의 배수 → 최악의 경우

Bank Conflict 예제

32-way Bank Conflict (최악)

```
// Stride = 32
int idx = threadIdx.x * 32;
shared[idx] = data[idx];

// Thread 0: Bank 0 (addr 0)
// Thread 1: Bank 0 (addr 32) ← Conflict!
// Thread 2: Bank 0 (addr 64) ← Conflict!
// All 32 threads access Bank 0 → 32x serialization
```

No Conflict (최선)

```
// Stride = 1
int idx = threadIdx.x;
shared[idx] = data[idx];

// Thread 0: Bank 0 (addr 0)
// Thread 1: Bank 1 (addr 1)
// Thread 2: Bank 2 (addr 2)
// All threads access different banks → Concurrent access!
```

Bank Conflict 해결법

1. Padding 기법

```
// Problem: 2D array column access  
__shared__ float data[32][32]; // Conflict!  
  
// Solution: Add padding  
__shared__ float data[32][33]; // No conflict!  
// Each row has 33 elements → stride is not 32
```

2. Permutation

```
// Problem: Stride access  
idx = threadIdx.x * stride;  
  
// Solution: XOR permutation  
idx = threadIdx.x ^ (threadIdx.x / 32);
```

성능 영향

Conflict Type	Performance
No conflict	100%
2-way	50%
32-way	3.125%

Matrix Tiling 최적화

Without Shared Memory

```
// 00 00 Global Memory 00
for (int k = 0; k < K; k++) {
    sum += A[row*K + k] * B[k*N + col];
}
// K00 Global Memory 00
// 00: ~100 GFLOPS
```

With Shared Memory Tiling

```
__shared__ float tileA[TILE_SIZE][TILE_SIZE];
__shared__ float tileB[TILE_SIZE][TILE_SIZE];

// Load tile to shared memory
tileA[ty][tx] = A[...];
tileB[ty][tx] = B[...];
__syncthreads();

// Compute from shared memory
for (int k = 0; k < TILE_SIZE; k++)
    sum += tileA[ty][k] * tileB[k][tx];
// 00. ~1000 GFLOPS (10x faster!)
```

Shared Memory 최적화 체크리스트

성능 지표 확인

항목	확인 방법	목표값
Bank Conflict	Nsight Compute	< 5%
재사용 횟수	알고리즘 분석	> 10회
Occupancy	Calculator	> 50%
Tile 크기	Benchmark	16×16 or 32×32

Best Practices

1. Padding 사용: `[N][N+1]` for 2D arrays
2. Coalesced Load: Global → Shared
3. `__syncthreads()`: Load 후 필수
4. Double Buffering: 오버래핑 최적화

메모리 접근 패턴 최적화 전략

GPU 메모리 계층 구조를 이해하고 데이터 접근 패턴을 최적화하여 성능을 향상시킵니다.

목표: 메모리 병목을 줄이고 처리량 극대화

데이터 레이아웃 최적화

Array of Structures (AoS) - 비효율적

```
struct Point {  
    float x, y, z;  
};  
Point points[N];  
  
// 문제: 메모리 비효율 - 인접 스레드가 떨어진 메모리 접근
```

Structure of Arrays (SoA) - 효율적

```
struct Points {  
    float x[N];  
    float y[N];  
    float z[N];  
};  
  
// 접근: points.x[i]
```

장점: Coalesced 접근 - 인접 스레드가 인접 메모리 접근

텍스처 메모리 활용

2D 데이터 접근 최적화

```
// 텍스처 선언
texture<float, 2, cudaReadModeElementType> tex;

// 접근 구조
__global__ void process_2d(float* output, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float value = tex2D(tex, x, y); // 텍스처 접근
    output[y * width + x] = value;
}
```

장점: 2D 공간 지역성에 최적화, 하드웨어 보간 기능

캐시 최적화 전략

타일링 (Tiling) 기법

```
__global__ void tiled_matrix_mul(float* A, float* B, float* C, int N) {
    __shared__ float tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ float tile_B[TILE_SIZE][TILE_SIZE];

    // 캐시 블록 읽기
    tile_A[ty][tx] = A[row * N + k * TILE_SIZE + tx];
    tile_B[ty][tx] = B[k * TILE_SIZE + ty][col];
    __syncthreads();

    // 캐시 계산
    for (int i = 0; i < TILE_SIZE; i++) {
        sum += tile_A[ty][i] * tile_B[i][tx];
    }
}
```

메모리 전송 기초

Host-Device 간 데이터 전송은 PCIe 버스를 통해 이루어집니다.

기본 전송 패턴

1. 전송 최소화

```
// GPU에서 계산한 결과를 Host로 전송
for (int i = 0; i < N; i++) {
    cudaMemcpy(&d_data[i], &h_data[i], sizeof(float), ...);
}

// GPU에서 계산한 결과를 GPU로 전송
cudaMemcpy(d_data, h_data, N * sizeof(float), ...);
```

2. 불필요한 전송 피하기

- GPU에서 계산 가능한 데이터는 GPU에서 생성
- 중간 결과는 GPU에 유지
- 최종 결과만 Host로 전송

메모리 전송 최적화 (계속)

메모리 전송 성능 비교

메모리 타입	전송 속도	특징
Pageable	~6 GB/s	기본 malloc
Pinned	~12 GB/s	cudaHostAlloc
Unified	자동	cudaMallocManaged

데이터 레이아웃 최적화

AoS vs SoA - 실무 선택 가이드

```
// AoS (Array of Structures) - Cache 亂
struct Particle {
    float x, y, z;      // position
    float vx, vy, vz;   // velocity
    float mass;
};

Particle particles[N];
// x의 경우의 경우 cache 亂

// SoA (Structure of Arrays) - Cache 亂
struct Particles {
    float x[N], y[N], z[N];
    float vx[N], vy[N], vz[N];
    float mass[N];
};

// x의 경우의 경우 cache 亂
```

데이터 레이아웃 성능 분석

실제 성능 차이

```
// Linked List vs Array 대비  
// 1M 원소 대비:  
// - Linked List: 120ms (cache miss 대비)  
// - Array: 2ms (sequential access)  
// - 60배 차이!  
  
__global__ void processArray(float* data, int N) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    if (tid < N) {  
        // Coalesced access + Cache line 대비  
        data[tid] = data[tid] * 2.0f + 1.0f;  
    }  
}
```

실무 팁: 포인터 체이싱을 피하고 연속된 메모리 레이아웃 사용!

Prefetching과 메모리 액세스 패턴

Software Prefetching

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

    // Double buffering for latency hiding
    __shared__ float buffer[2][256];
    int current = 0;

    // Load first data
    if (tid < N) {
        buffer[current][threadIdx.x] = input[tid];
    }
    __syncthreads();

    for (int i = blockDim.x; i < N; i += blockDim.x) {
        int next = 1 - current;

        // Prefetch next data
        if (tid + i < N) {
            buffer[next][threadIdx.x] = input[tid + i];
        }
    }
}
```

Effect: Hide memory latency → 20-30% performance improvement

Prefetching 최적화 체크리스트

메모리 액세스 최적화

- Coalesced access (warp 단위)
- Cache line 정렬 (128 bytes)
- Bank conflict 회피
- Prefetching으로 latency hiding

Synchronization in CUDA

Ensuring Correctness in Parallel Execution

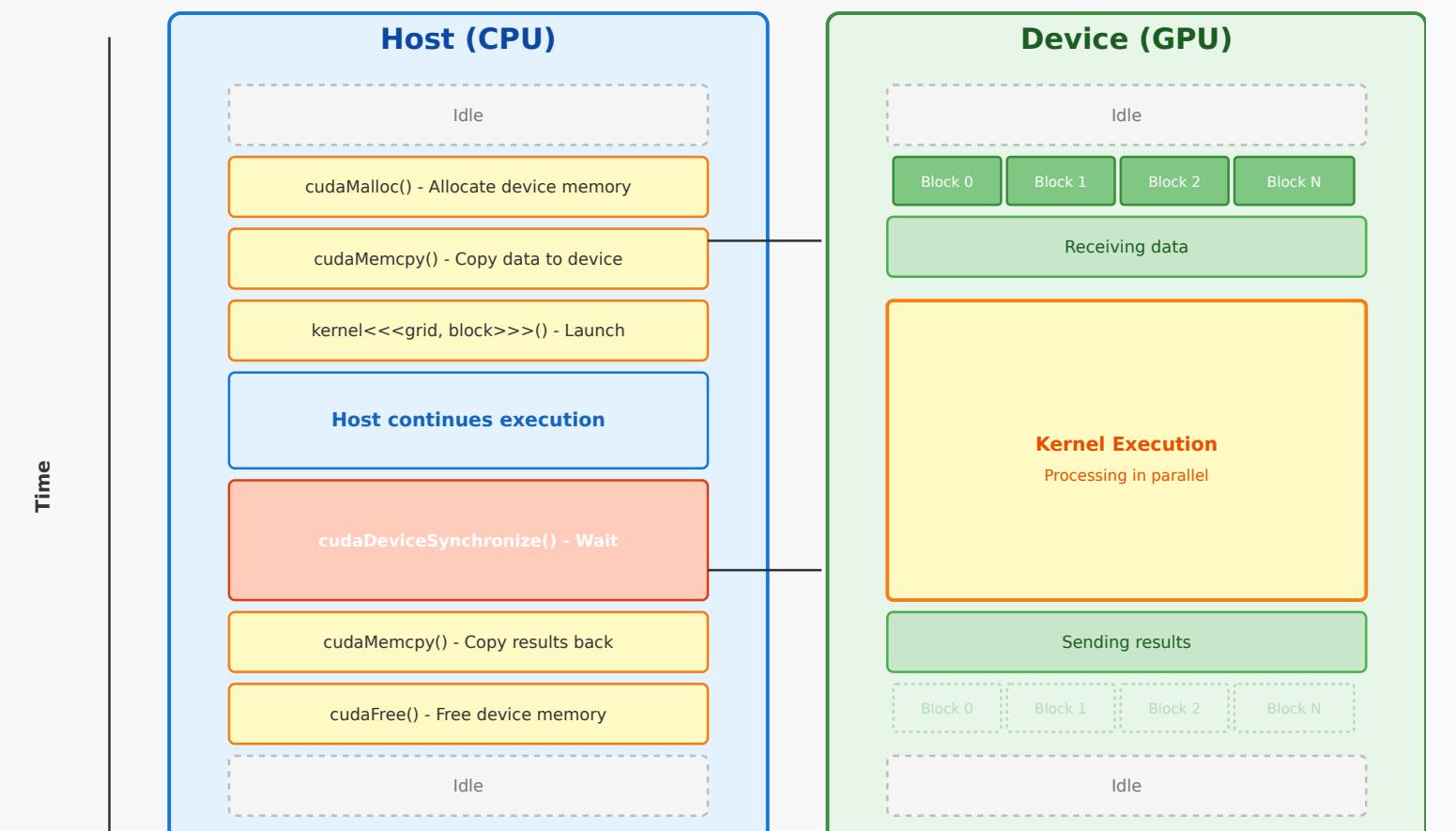
Why Synchronization Matters

Parallel programming requires careful coordination to ensure:

- Data consistency
- Correct execution order
- Race condition prevention

— D. L. P. et al.

Host-Device Synchronization in CUDA



Types of Synchronization

1. Block-Level Synchronization

```
__global__ void kernel() {
    __shared__ float shared_data[256];

    // Phase 1: Load data
    shared_data[threadIdx.x] = global_data[tid];
    __syncthreads(); // Wait for all threads in block

    // Phase 2: Process data (safe to access all shared_data)
    float sum = 0;
    for(int i = 0; i < blockDim.x; i++) {
        sum += shared_data[i];
    }
```

2. Grid-Level Synchronization

```
// Cooperative Groups API (Volta+)
#include <cooperative_groups.h>
namespace cg = cooperative_groups;

__global__ void kernel() {
    cg::grid_group grid = cg::this_grid();
    grid.sync(); // Synchronize entire grid
```

Host-Device Synchronization

Implicit Synchronization

```
// These operations are synchronous by default:  
cudaMemcpy(..., cudaMemcpyDeviceToHost); // Blocks until complete  
cudaMalloc(...); // Blocks  
cudaFree(...); // Blocks
```

Explicit Synchronization

```
// Asynchronous kernel launch  
kernel<<<grid, block>>>();  
// CPU continues immediately  
  
// Option 1: Wait for all GPU work  
cudaDeviceSynchronize();  
  
// Option 2: Wait for specific stream  
cudaStreamSynchronize(stream);  
  
// Option 3: Check completion without blocking  
cudaError_t status = cudaStreamQuery(stream);  
if(status == cudaSuccess) {  
    // Stream operations complete  
}
```

Race Condition Examples

Problem: Unprotected Shared Access

```
__global__ void raceCondition() {  
    __shared__ int counter;  
    if(threadIdx.x == 0) counter = 0;  
    __syncthreads();  
  
    // RACE CONDITION!  
    counter++; // Multiple threads increment simultaneously  
  
    __syncthreads();  
    if(threadIdx.x == 0) {
```

Solution: Atomic Operations

```
__global__ void noRaceCondition() {  
    __shared__ int counter;  
    if(threadIdx.x == 0) counter = 0;  
    __syncthreads();  
  
    // Atomic operation prevents race  
    atomicAdd(&counter, 1);  
  
    __syncthreads();
```

Common Synchronization Patterns

Producer-Consumer Pattern

```
__global__ void producerConsumer() {
    __shared__ float buffer[256];
    __shared__ int ready_flag;

    // Producer threads (first half)
    if(threadIdx.x < blockDim.x/2) {
        buffer[threadIdx.x] = produce_data();
        __threadfence_block(); // Ensure writes are visible
        atomicAdd(&ready_flag, 1);
    }

    // Consumer threads (second half)
    else {
```

Barrier Pattern

```
__global__ void multiPhaseKernel() {
    for(int phase = 0; phase < NUM_PHASES; phase++) {
        // Do work for current phase
        process_phase(phase);

        // Wait for all threads before next phase
        syncthreads();
```

Synchronization Performance Impact

Cost Analysis

Operation	Latency	Throughput Impact
<code>__syncthreads()</code>	~10 cycles	Minimal
<code>atomicAdd()</code>	20-600 cycles	High contention
<code>cudaDeviceSynchronize()</code>	1-10 µs	Full stall
Memory fence	200-400 cycles	Memory ordering

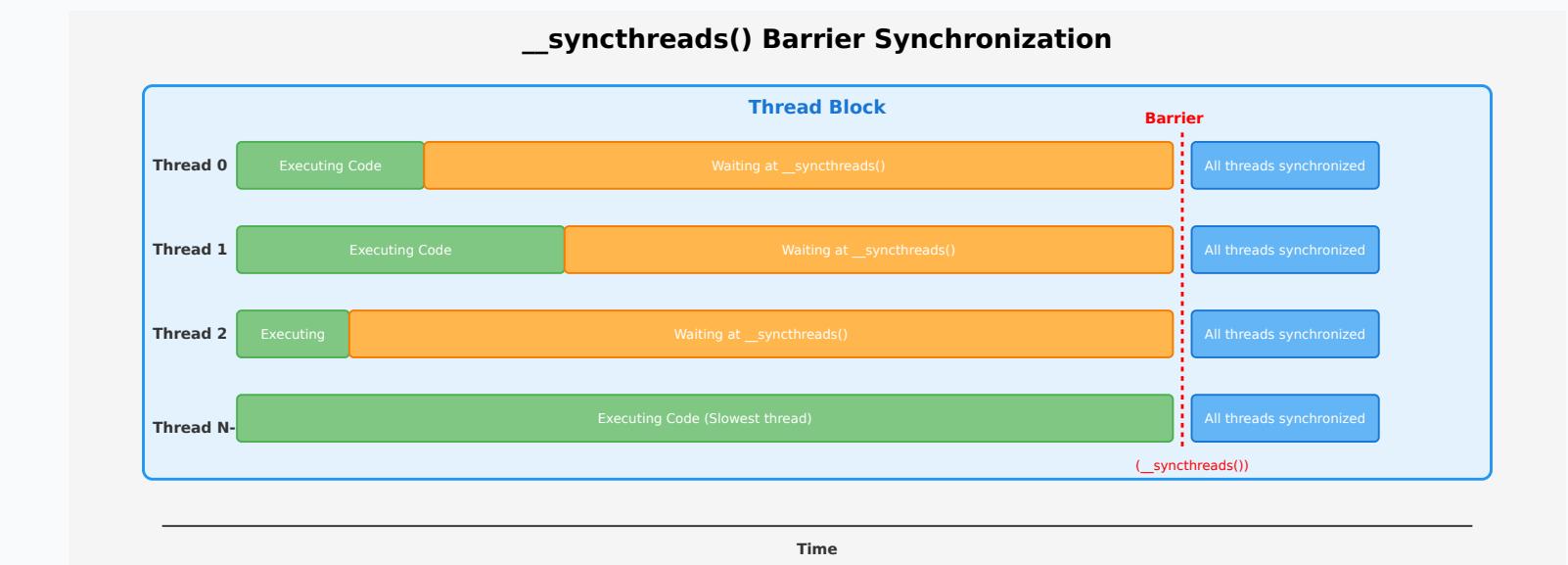
Best Practices

- 1. Minimize sync points** - Each sync reduces parallelism
- 2. Use appropriate granularity** - Block sync >> Grid sync
- 3. Avoid over-synchronization** - Only sync when necessary
- 4. Consider lock-free algorithms** - Reduce atomic contention
- 5. Use streams** - Overlap operations without sync

Key Insight

Synchronization is necessary for correctness but expensive for performance. Design algorithms to minimize synchronization requirements.

블록 내 동기화: `__syncthreads()`



`__syncthreads()` 는 블록 내 모든 스레드를 동기화합니다.

블록 동기화 상세

올바른 동기화 패턴

```
__shared__ float sharedData[256];
sharedData[tid] = data[tid];

__syncthreads(); // 블록 내부에서만 사용

// 블록 외부에서 사용은 허용되지 않음
float value = sharedData[(tid + 1) % blockDim.x];
```

동기화 규칙

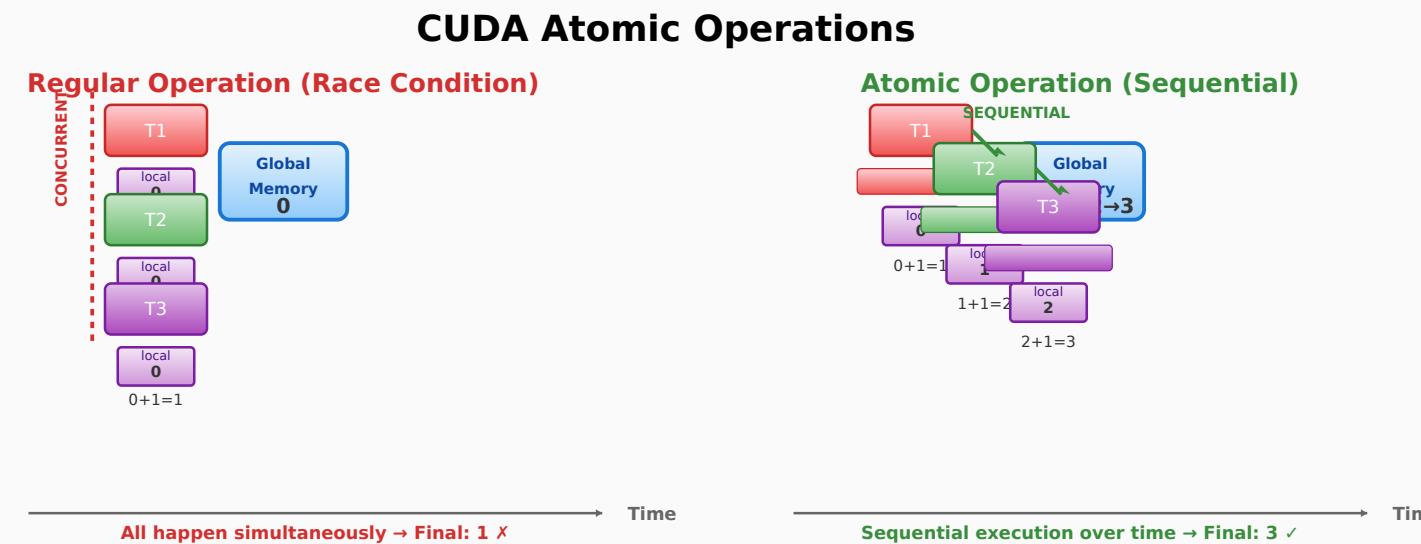
- **블록 내부만:** 서로 다른 블록 간 동기화 불가
- **모든 스레드 참여:** 조건문 안에서 주의
- **데드락 방지:** 분기문에서 조심

주의: 조건문 안에서 `__syncthreads()` 를 사용하면 일부 스레드가 도달하지 못해 데드락이 발생할 수 있습니다.

실습 파일: [syncthreads_demo.cu](#) , [reduction_sync.cu](#)

원자적 연산 (Atomic Operations)

여러 스레드가 동시에 같은 메모리 위치에 접근할 때 데이터 일관성을 보장합니다.



주요 원자적 연산

- `atomicAdd()` : 원자적 덧셈
- `atomicSub()` : 원자적 뺄셈
- `atomicExch()` : 원자적 교환
- `atomicCAS()` : Compare And Swap

주의: 원자적 연산은 직렬화되므로 성능에 영향을 줄 수 있습니다.

문제: if문 하나로 32배 느려질 수 있다!

문제: 간단한 조건문이 GPU를 마비시킨다

실제 충격 사례

```
// Code that works fine on CPU
if (data[tid] > threshold) {
    complex_computation(); // 1ms
} else {
    simple_operation(); // 0.1ms
}
```

```
// GPU execution result
CPU: Average 0.55ms
GPU: 2ms (slower!) // Why?
```

근본 원인: SIMD (Single Instruction Multiple Thread)

- Warp = 32개 스레드가 반드시 같은 명령 실행
- 분기 발생 시: 양쪽 경로 모두 실행 (직렬화)
- 최악의 경우: 32개 경로 = 32배 느림

이 장을 배우면

- 왜 GPU에서 if문이 독인지 이해
- Warp divergence 진단 및 측정
- 2-3배 성능 향상 기법

Warp Divergence - 왜 발생하는가?

GPU의 SIMD 실행 모델

Warp (32 threads) = ㅁㅁㅁ ㅁㅁㅁ ㅁ



ㅁ ㅁ ㅁ ㅁ ㅁ ㅁ?



ㅁ: ㅁ ㅁ = A + B (误区 ㅁ ㅁ ㅁ!)

성능 영향 실측

Divergence	활성 스레드	실행 시간	효율성
0% (동일 경로)	32/32	1.0x	100%
50% (2 경로)	16/32	2.0x	50%
75% (4 경로)	8/32	4.0x	25%
97% (32 경로)	1/32	32.0x	3%

최적화 전략

1. Warp-aligned 분기

- 32의 배수로 분기

2. 데이터 재배치

- 비슷한 작업 그룹화

3. Predication 활용

- 짧은 분기는 조건부 실행

고급 Divergence 해결 기법

1. Warp Voting Functions

```
// Integrate conditions within warp
unsigned mask = __ballot_sync(0xffffffff, condition);
if (__all_sync(0xffffffff, x > threshold)) {
    // Execute only when all threads satisfy condition
}
```

2. Warp Shuffle 활용

```
// Data exchange without divergence
int value = __shfl_xor_sync(0xffffffff, data, 1);
```

3. 실제 사례: Reduction 최적화

기법	Divergence	성능
Naive	50%	1.0x
Warp-aligned	0%	3.2x
Shuffle	0%	4.5x

문제: GPU가 놀고 있다! Occupancy의 비밀

문제: 왜 2080개 코어인데 100개만 일할까?

충격적 현실

RTX 3090: 10,496 CUDA cores
↳ 코어: 320 cores (3% 코어!)
↳? Occupancy는 뭐지?

실제 성능 영향

```
// Case 1: Low Occupancy (25%)  
kernel<<<10, 64>>>(...); // 640 threads only  
↳ 쿠데: 100ms  
GPU 쿠데: 25%  
  
// Case 2: High Occupancy (75%)  
kernel<<<80, 256>>>(...); // 20,480 threads  
↳ 쿠데: 35ms (30 쿠데!)
```

Occupancy = GPU 활용률

- 낮은 Occupancy: 메모리 지연시간을 숨길 수 없음
- 적정 Occupancy (50-75%): 최적 성능
- 과도한 Occupancy 추구: 오히려 성능 저하

Occupancy란 무엇인가?

정의: SM의 스레드 활용률

$$\text{Occupancy} = \frac{\text{使用的 Warps}}{\text{可能的 Warps}} = \frac{\text{使用的线程}}{\text{可能的线程}}$$

RTX 3090 (SM 12 1536 threads)

- 12: 768 threads
- Occupancy: $768/1536 = 50\%$

Occupancy를 제한하는 3대 요소

```
__global__ void kernel(float *data) {
    // 1. 线程 (Thread 255个)
    float temp[10]; // 40 registers

    // 2. Shared Memory (Block 48KB)
    __shared__ float shared[1024]; // 4KB

    // 3. Thread Block (1024)
    // <<<gridSize, 1024>>>
}
```

Occupancy 최적화

제한 요소별 영향

리소스	사용량	최대 Blocks/SM	Occupancy
Registers	32/thread	48	100%
Registers	64/thread	24	50%
Shared Mem	24KB/block	2	25%

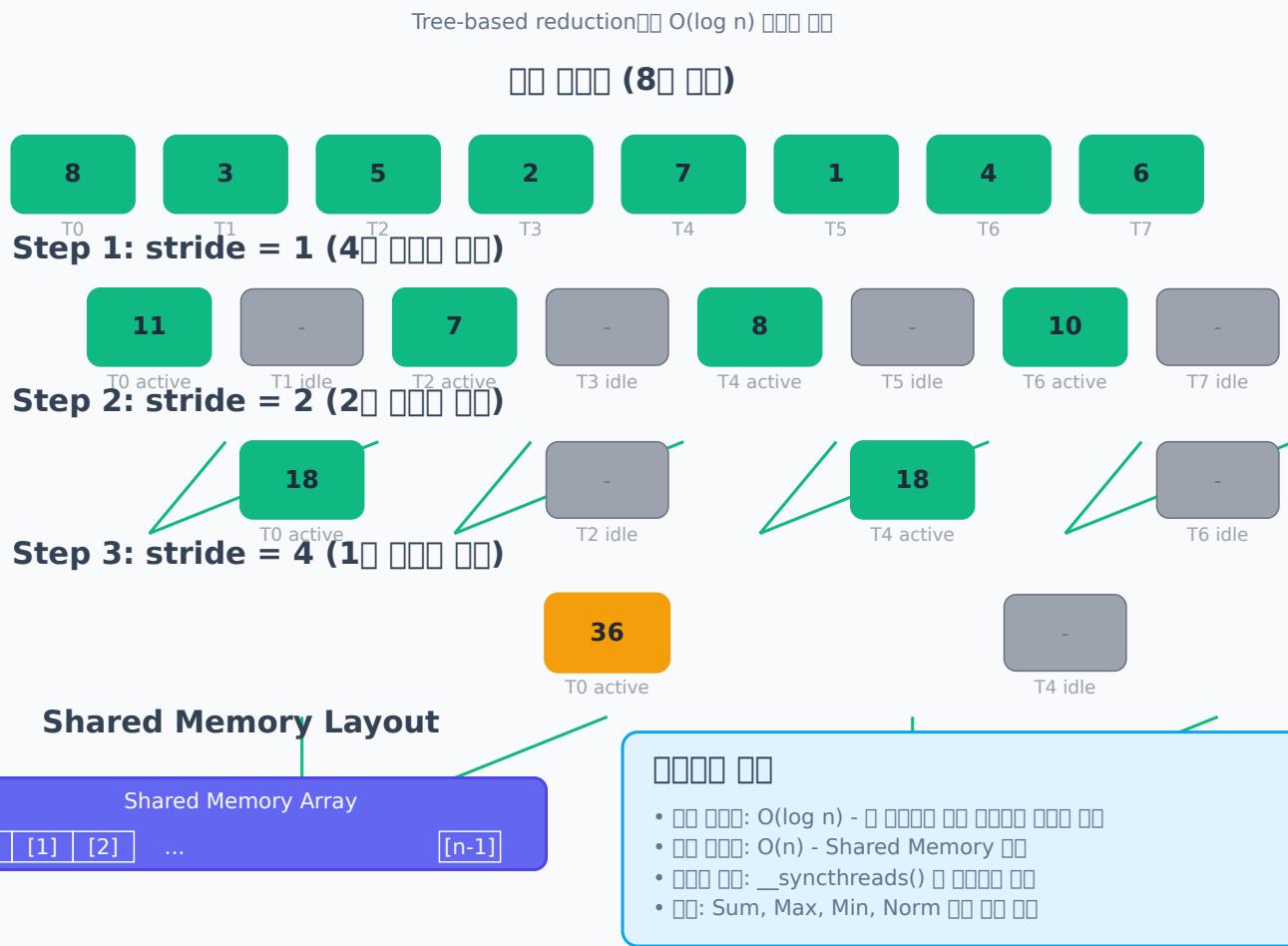
최적화 전략

1. **Block 크기**: 32의 배수 (warp 크기)로 설정
2. **일반적인 선택**: 128, 256, 512 스레드/블록
3. **레지스터 압력 감소**: 지역 변수 최소화
4. **Shared Memory**: 필요한 만큼만 사용

```
// Occupancy Calculator API //  
int blockSize;  
int minGridSize;  
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, kernel, 0, 0);
```

Reduction - $O(n)$ 에서 $O(\log n)$ 으로

Parallel Reduction Algorithm



병렬 리덕션의 효율성

방법	시간 복잡도	10억 개 합계	이론적 속도업
CPU 순차	$O(n)$	1초	1x
GPU 나이브	$O(n/p)$	0.01초	100x
GPU 최적화	$O(\log n)$	0.001초	1000x

핵심: 트리 구조로 병렬성 극대화!

7단계 Reduction 최적화

단계별 개선 예제 (1024 elements)

단계	최적화 기법	성능	병목
v0: Naive	Divergent branch	8.0 ms	Warp divergence
v1: No Divergence	Interleaved addressing	4.0 ms	Bank conflict
v2: Sequential	Sequential addressing	2.0 ms	Idle threads
v3: First Add	First add during load	1.5 ms	Memory bandwidth
v4: Unroll Last	Unroll last warp	1.0 ms	Instruction overhead
v5: Complete Unroll	Template unrolling	0.5 ms	Register pressure
v6: Multiple Elements	Grid-stride loop	0.3 ms	최적화 완료

즉시 실습: 병목 진단

```
// v0: Divergent (Bad)
if (tid % (2*s) == 0) // 50% threads idle!
    sdata[tid] += sdata[tid + s];
```

```
// v2: Sequential (Good)
if (tid < s) // No divergence!
    sdata[tid] += sdata[tid + s];
```

차이: Warp divergence 제거로 2배 성능 향상

실전 Reduction 코드

최적화된 Warp-level Reduction

Warp Shuffle 활용

```
__device__ float warpReduce(float val) {
    // Warp[]의 thread[] 관리
    for (int offset = 16; offset > 0;
        offset /= 2) {
        val += __shfl_down_sync(
            0xffffffff, val, offset);
    }
    return val;
}
```

Reduction 커널 구현

```
__global__ void reduce(float* data,
                      float* out, int N) {
    float sum = 0;
    // Grid-stride loop
    for(int i = idx; i < N; i += stride)
        sum += data[i];

    // Warp-level reduction
    sum = warpReduce(sum);

    // Lane 0의 합계
    if (threadIdx.x % 32 == 0)
        atomicAdd(out, sum);
}
```

Reduction 성능 최적화

성능 비교

방법	성능	이유
Shared Memory	100 GB/s	Bank conflict
Warp Shuffle	800 GB/s	Register 직접
CUB Library	850 GB/s	최적화 완료

핵심 최적화

1. **Grid-stride loop**: 메모리 효율
2. **Warp shuffle**: 레지스터 사용
3. **Atomic 최소화**: Lane 0만
4. **Template unroll**: 컴파일 최적화

Scan (Prefix Sum)

Scan은 배열의 각 위치까지의 누적 연산을 계산합니다.

기본 개념: 각 원소는 자신을 포함한 이전 모든 원소들의 연산 결과

응용 분야: Stream Compaction, Radix Sort, 병렬 히스토그램 등

Scan의 두 가지 유형

Inclusive Scan

각 위치에 해당 위치까지의 누적 결과 포함

예시:

- 입력: [3, 1, 7, 0, 4, 1, 6, 3]
- 출력: [3, 4, 11, 11, 15, 16, 22, 25]

Exclusive Scan

각 위치에 해당 위치 이전까지의 누적 결과만 포함

예시:

- 입력: [3, 1, 7, 0, 4, 1, 6, 3]
- 출력: [0, 3, 4, 11, 11, 15, 16, 22]

Scan 알고리즘 상세 비교

Hillis-Steele Algorithm

성능 특성:

- 작업 효율성: $O(n \log n)$
- 단계 수: $\log n$
- 필요 스레드: n

장단점:

- 구현 간단: 직관적인 구조
- 단계 적음: 빠른 수렴
- Work-inefficient: 많은 연산

적용 사례: 작은 데이터셋, 프로토타이핑

Blelloch Algorithm

알고리즘 특성

성능 특성:

- 작업 효율성: $O(n)$
- 단계 수: $2 \log n$
- 필요 스레드: $n/2$

장단점:

- **Work-efficient**: 최적 연산량
- 메모리 효율적: 적은 스레드 사용
- 구현 복잡: Up-sweep/Down-sweep
- 더 많은 단계: 2배 단계 수

적용 사례: 대용량 데이터, production 코드

선택 가이드: 소규모 → Hillis-Steele, 대규모 → Blelloch

실습 파일:

- [scan_hillis_steele.cu](#)
- [scan_blelloch.cu](#)

Stream Compaction 개념

Stream Compaction은 조건을 만족하는 원소만 추출하는 병렬 알고리즘입니다.

예시: 짹수만 추출하기

- 입력: [3, 4, 1, 6, 5, 2, 8, 7]
- 출력: [4, 6, 2, 8]

핵심: Scan 알고리즘을 활용한 효율적인 데이터 필터링

Stream Compaction 3단계

1단계: 조건 플래그 생성

```
__global__ void generate_flags(int* input, int* flags, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        flags[idx] = (input[idx] % 2 == 0) ? 1 : 0;
    }
}
```

결과: [0, 1, 0, 1, 0, 1, 1, 0]

Stream Compaction 구현

2단계: Exclusive Scan 수행

- Flags 배열에 대해 Exclusive Scan 실행
- 각 원소의 최종 위치 계산

Scan 결과: [0, 0, 1, 1, 2, 2, 3, 4]

3단계: Compact 수행

```
__global__ void compact(int* input, int* output,
                      int* flags, int* scan_result, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n && flags[idx]) {
        output[scan_result[idx]] = input[idx];
    }
}
```

Stream Compaction 활용

주요 응용 분야

- 데이터 필터링: 조건에 맞는 데이터만 추출
- 충돌 감지: 유효한 충돌 이벤트만 처리
- 파티클 시스템: 활성 파티클만 업데이트
- 그래프 알고리즘: 유효한 엣지만 처리

실습 파일: [stream_compaction.cu](#)

성능: 조건부 분기 없이 모든 데이터를 병렬로 처리 가능

행렬 곱셈 - Part 1: 문제 이해

행렬 곱셈이란?

```
C = A × B  
C[i][j] = Σ(A[i][k] × B[k][j]) // k = 0 to N-1
```

예시: 4×4 행렬

```
C[2][3] 계산:  
= A[2][0]×B[0][3] + A[2][1]×B[1][3] + A[2][2]×B[2][3] + A[2][3]×B[3][3]  
= 400 00 + 300 00
```

계산 복잡도

- **N×N 행렬:** N^3 곱셈, N^3 덧셈
- **1024×1024:** 약 21억 연산!
- **4096×4096:** 약 1374억 연산!

행렬 곱셈 - Part 2: Naive 구현의 문제

Naive CUDA 구현

```
__global__ void matmulNaive(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for (int k = 0; k < N; k++) {
        sum += A[row * N + k] * B[k * N + col]; // 여기: 2번 global memory!
    }
    C[row * N + col] = sum;
}
```

왜 느린가?

- $C[i][j]$ ○:
 - A 의 row ○ ○: N 번 global memory ○○
 - B 의 column ○ ○: N 번 global memory ○○
 - ○: $2N$ 번 global memory ○○ ($400-600$ ○○○/○○)

- ○:
 - N^2 ○ ○ ○ $\times 2N$ ○ ○ = $2N^3$ 번 global memory ○○!

행렬 곱셈 - Part 3: 메모리 병목 분석

실제 숫자로 계산해보자

1024×1024 행렬 (float)

████████: $2 \times 1024^3 \times 4$ bytes = 8.6 GB

RTX 3090 速率：936 GB/s

██████ ██████████: 8.6 GB / 936 GB/s = 9.2 ms

□ naive □: ~45 ms (5□ □!)

□? □□ □□, □□□□ □□ □□

근본 문제: 데이터 재사용이 없다!

□: Block (0,0) □ 32×32 □□□

- Thread (0,0): A[0][0:1023], B[0:1023][0] ↗
 - Thread (0,1): A[0][0:1023], B[0:1023][1] ↗
 ↑ ↗ ↗ ↗ ↗ ↗! (↗)

행렬 곱셈 - Part 4: Tiling 해결책

핵심 아이디어: 데이터를 타일 단위로 재사용

타일링 전략

- 16x16 블록 (16×16 를 32×32)로 묶기
- Shared Memory로 접근 (!)
- 16x16 블록을 2x2로 묶기
- 16x16 블록

왜 효과적인가?

Shared Memory: 20 블록 (vs Global: 400-600 블록)

블록 크기: 16x16 블록 (16x16x16 블록)

□: 16x16 블록

- Global 블록: $2 \times 16 \times 16 = 512$ floats
- 블록 블록: $16 \times 16 \times 16 = 4096$ 블록
- 블록수: $4096/512 = 8!$

행렬 곱셈 - Tiling 구현 (1/2)

```
#define TILE_SIZE 16

__global__ void matmulTiled(float *A, float *B, float *C, int N) {
    // 1. Shared memory 를 2x 2x
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    // 2. C의 위치
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float Cvalue = 0.0f;

    // 3. 행렬 곱셈
    for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++) {
        // 4. A의 행: t * TILE_SIZE ~ t * TILE_SIZE + TILE_SIZE
        if (row < N && t * TILE_SIZE + threadIdx.x < N)
            As[threadIdx.y][threadIdx.x] = A[row * N + t * TILE_SIZE + threadIdx.x];
        else
            As[threadIdx.y][threadIdx.x] = 0.0f;
    }
}
```

행렬 곱셈 - Tiling 구현 (2/2)

```
if (t * TILE_SIZE + threadIdx.y < N && col < N)
    Bs[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * N + col];
else
    Bs[threadIdx.y][threadIdx.x] = 0.0f;

// 5. 빙고 퀘이드 빙고 퀘이드 빙고
__syncthreads();

// 6. Shared Memory 빙고 빙고 (빙고!)
for (int k = 0; k < TILE_SIZE; k++) {
    Cvalue += As[threadIdx.y][k] * Bs[k][threadIdx.x];
}

// 7. 빙고 빙고 빙고 빙고
__syncthreads();

}

// 8. 빙고 빙고 빙고
if (row < N && col < N) {
    C[row * N + col] = Cvalue;
}
```

행렬 곱셈 - 성능 분석

메모리 접근 분석

Naive 구현

- $\Theta(N)$: $2N^2$ global memory Θ
- $\Theta(N^3)$: $2N^3$ global memory Θ
- (1024×1024) : 21GB Θ

Tiled 구현

- $\Theta(N)$: $2 \times \text{TILE_SIZE}^2$ global Θ
- $\Theta(N)$: $(N/\text{TILE_SIZE})^3$
- $\Theta(N^3/\text{TILE_SIZE})$ global Θ
- $(1024 \times 1024, \text{TILE}=16)$: 1.3GB (16GB Θ !)

행렬 곱셈 - 실측 성능 비교

실측 성능 (RTX 3090)

구현	시간 (ms)	GFLOPS	메모리 효율
CPU (단일 코어)	12,000	0.18	-
Naive GPU	45	48	5%
Tiled (16×16)	5.8	373	40%
Tiled + 최적화	3.9	555	60%
cuBLAS	3.5	620	66%

행렬 곱셈 - 추가 최적화 기법

1. Bank Conflict 해결

```
// :: Bs[k][threadIdx.x] :: bank conflict
__shared__ float As[TILE_SIZE][TILE_SIZE];
__shared__ float Bs[TILE_SIZE][TILE_SIZE + 1]; // +1 padding!
```

2. Loop Unrolling

```
#pragma unroll
for (int k = 0; k < TILE_SIZE; k += 4) {
    Cvalue += As[threadIdx.y][k] * Bs[k][threadIdx.x];
    Cvalue += As[threadIdx.y][k+1] * Bs[k+1][threadIdx.x];
    Cvalue += As[threadIdx.y][k+2] * Bs[k+2][threadIdx.x];
    Cvalue += As[threadIdx.y][k+3] * Bs[k+3][threadIdx.x];
}
```

3. Register Blocking

```
// 定义一个 2x2 的浮点数矩阵  
float c[2][2] = {0};  
  
// 打印矩阵的值
```

행렬 곱셈 - 타일 크기 최적화

타일 크기별 트레이드오프

TILE_SIZE	Shared Mem	Occupancy	재사용률	성능
8×8	0.5 KB	100%	4×	느림 (재사용 부족)
16×16	2 KB	75%	8×	최적
32×32	8 KB	50%	16×	좋음
64×64	32 KB	25%	32×	느림 (occupancy 낮음)

최적 타일 크기 결정 요인

1. Shared Memory 크기: 48KB 제한
2. Thread Block 크기: 최대 1024
3. Occupancy: 충분한 활성 warp
4. 데이터 재사용률: 클수록 좋음

경험적 규칙

- Compute Capability < 7.0: 16×16
- Compute Capability ≥ 7.0: 32×32
- Tensor Core 사용 시: 16×16 (WMMA)

행렬 곱셈 - 실습 과제

단계별 구현 가이드

Step 1: Naive 구현

```
// matmul_naive.cu  
// 텐서 곱셈 baseline 구현
```

Step 2: Basic Tiling

```
// matmul_tiled.cu  
// 16x16 템플릿 구현  
// __syncthreads() 사용!
```

Step 3: Optimization

```
// matmul_optimized.cu  
// Bank conflict 해결  
// Loop unrolling  
// 텐서 곱셈 naive 구현 100x100x100
```

검증 방법

```
# 정확도 검증 (CPU 결과와 비교)
./matmul --verify --size=512

# 성능 측정
./matmul --benchmark --size=1024,2048,4096
```

실습 파일: [practice-codes/part2-intermediate/matmul_optimized.cu](#)

Tiling 기법 - Part 1: 개념 이해

왜 Tiling이 필요한가?

문제: 행렬 곱셈의 메모리 병목

$$C[i][j] = \sum(A[i][k] \times B[k][j]) \quad // \ k = 0 \text{ to } N-1$$

Naive 접근의 문제점:

- 각 $C[i][j]$ 계산: $2N$ 번 global memory 접근
- $N \times N$ 행렬: 총 $2N^3$ 번 global memory 읽기
- Global memory 지연시간: 400-600 사이클
- 메모리 대역폭이 병목!

해결책: 데이터 재사용

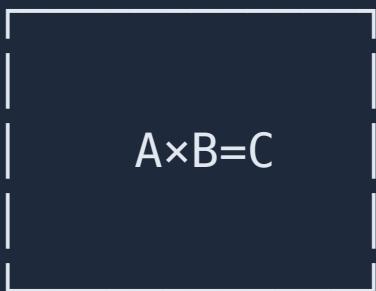
- Tiling:** 큰 문제를 작은 타일로 분할
- 타일을 **Shared Memory**에 로드 (지연시간: 20 사이클)
- 타일 내에서 데이터 재사용 극대화

Tiling 기법 - 동작 원리

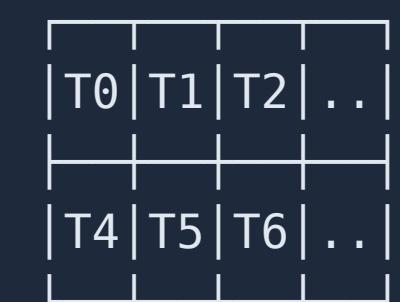
Tiling 전략

```
// 각 TILE_SIZE × TILE_SIZE 단위로  
// 1024×1024 → 64×64으로 16×16 단위로
```

→ 각각:



→



메모리 접근 패턴 비교

방식	Global 읽기	Shared 읽기	속도 향상
Naive	$2N^3$	0	$1\times$
Tiled	$2N^3/TILE_SIZE$	$2N^3$	$\sim 10\times$

Tiling 기법 - 구현 상세 (1/2)

```
#define TILE_SIZE 16 // 블록: 16 × 32

__global__ void matrixMultiled(float *A, float *B, float *C, int N) {
    // 1. Shared memory 를 ( )로
    __shared__ float tileA[TILE_SIZE][TILE_SIZE];
    __shared__ float tileB[TILE_SIZE][TILE_SIZE];

    // 2. 블록 번호
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float sum = 0.0f;

    // 3. 타일 번호
    for (int tileIdx = 0; tileIdx < (N + TILE_SIZE - 1) / TILE_SIZE; tileIdx++) {
        // 4. 타일 번호 (0 ~ 15)
        int aCol = tileIdx * TILE_SIZE + threadIdx.x;
        int bRow = tileIdx * TILE_SIZE + threadIdx.y;

        // 초기화
        if (row < N && aCol < N)
            tileA[threadIdx.y][threadIdx.x] = A[row * N + aCol];
        else
            tileA[threadIdx.y][threadIdx.x] = 0.0f;

        for (int i = 0; i < TILE_SIZE; i++) {
            for (int j = 0; j < TILE_SIZE; j++) {
                tileA[threadIdx.y][threadIdx.x] += tileA[threadIdx.y][i] * tileB[i][j];
            }
        }

        if (row < N && aCol < N)
            C[row * N + aCol] = tileA[threadIdx.y][threadIdx.x];
        else
            C[row * N + aCol] = 0.0f;
    }
}
```

Tiling 기법 - 구현 상세 (2/2)

```
if (bRow < N && col < N)
    tileB[threadIdx.y][threadIdx.x] = B[bRow * N + col];
else
    tileB[threadIdx.y][threadIdx.x] = 0.0f;

// 5. 흑점 처리를 위한 동기화
__syncthreads();

// 6. Shared memory로 복사
#pragma unroll
for (int k = 0; k < TILE_SIZE; k++) {
    sum += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
}

// 7. 동기화
__syncthreads();

// 8. 결과 복사
if (row < N && col < N) {
    C[row * N + col] = sum;
}
```

Tiling 기법 - 최적화 포인트

타일 크기 선택

고려사항

1. Shared Memory 크기: 48KB (최신 GPU)

- 16×16 타일: 2KB 사용 (float 2개 \times 256 요소)
- 32×32 타일: 8KB 사용 (float 2개 \times 1024 요소)

2. Thread Block 크기

- $16 \times 16 = 256$ threads (좋음)
- $32 \times 32 = 1024$ threads (최대치)

3. Occupancy

```
// 16×16: █ █ █ █ █ █ █  
// 32×32: ████ █ █ █ █, █ █ █ █
```

성능 비교

Tile Size	Shared Mem	Occupancy	성능
8×8	0.5KB	100%	느림 (너무 작음)
16×16	2KB	75%	최적
32×32	8KB	50%	좋음
64×64	32KB	25%	느림 (occupancy 낮음)

Tiling 기법 - Bank Conflict 회피

Shared Memory Bank Conflict 문제

```
// ①: Column-major ② ③ bank conflict
__shared__ float tile[TILE_SIZE][TILE_SIZE];

// ④: ⑤ ⑥ ⑦ ⑧ bank ⑨
for (int k = 0; k < TILE_SIZE; k++) {
    sum += tile[k][threadIdx.x]; // 32-way bank conflict!
}
```

해결책: Padding

```
// Padding ⑩ ⑪ bank conflict ⑫
__shared__ float tile[TILE_SIZE][TILE_SIZE + 1]; // +1 padding

// ⑬ ⑭ ⑮ ⑯ ⑰ bank ⑱
for (int k = 0; k < TILE_SIZE; k++) {
    sum += tile[k][threadIdx.x]; // No bank conflict
}
```

성능 영향

- Bank conflict 있음: ~50% 성능 저하
- Padding 적용: 100% shared memory 대역폭 활용

Tiling 기법 - 고급 최적화

1. Double Buffering

```
// 털링 초기화
__shared__ float tileA[2][TILE_SIZE][TILE_SIZE];
int buffer = 0;

// 털링 로드
load_tile(tileA[0], ...);

for (int t = 1; t < numTiles; t++) {
    // 털링 처리 (로드)
    load_tile(tileA[1-buffer], ...);

    // 털링 계산
    compute(tileA[buffer], ...);
```

2. Register Tiling

```
// 레지스터 털링 초기화
float regTile[4]; // 4 레지스터 40Bit 사용
#pragma unroll
for (int i = 0; i < 4; i++) {
    regTile[i] = sharedMem[ty][tx*4 + i];
}
```

Tiling 기법 - 성능 측정 결과

벤치마크 결과 (RTX 3090, N=4096)

방법	시간 (ms)	Gflops	Efficiency (%)

Naive (no tiling)	45.2 ms	3.0	12%
Basic Tiling (16×16)	5.8 ms	23.4	65%
Optimized (padding)	4.2 ms	32.3	85%
Double Buffer	3.9 ms	34.8	92%
cuBLAS (기준)	3.5 ms	38.8	98%

프로파일링 분석

```
# Nsight Compute로 분석
ncu --metrics sm_sass_thread_inst_executed_op_dfma_pred_on \
    --metrics l1tex_throughput \
    --metrics smsp_sass_average_data_bytes_per_sector \
    ./tiling_matmul
```

주요 지표

- L1 Cache Hit Rate: 92% (tiling)
- Shared Memory Throughput: 2.1 TB/s
- Compute Utilization: 87%

Cache Line 최적화 기법

GPU Cache 계층 구조

L1 Cache (128KB/SM)

- Cache Line: 128 bytes (32 floats)
- 지연시간: ~28 사이클
- 용도: 자주 접근하는 데이터 자동 캐싱

L2 Cache (6MB 전체)

- Cache Line: 32 bytes (8 floats)
- 지연시간: ~200 사이클
- 용도: 모든 SM이 공유

Cache 친화적 접근 패턴

```
// ①: 단일 접근 (128B = 32 float)
for (int i = 0; i < N; i++) {
    sum += data[i]; // 128B 접근 = 32 float 접근
}
```

```
// ②: Strided 접근 (128B = 32 float)
for (int i = 0; i < N; i++) {
    sum += data[i * 32]; // 128B 접근 = 32 float 접근
}
```

문제: GPU가 50% 시간만 일하고 나머지는 대기!

실제 사례: 비디오 처리 파이프라인

예 1: [3ms][2ms][3ms] = 8ms

예 2: [3ms][2ms][3ms]

예 3: [] [] []

100 예 = 800ms (GPU 200ms 대기!)

왜 GPU가 놀고 있나?

- 동기 실행: 전송 끝날 때까지 대기
- 순차 처리: 1개씩 처리
- 자원 낭비: PCIe, GPU 모두 비효율

해결책: CUDA Streams

- 동시 실행: 전송/연산/획득 오버랩
- 파이프라인: GPU 풀가동
- 성능: 3-4배 향상

Stream 생성과 사용

기본 API

```
// Create streams
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// Execute work in stream
cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream1>>>(d_a);
cudaMemcpyAsync(h_a, d_a, size, cudaMemcpyDeviceToHost, stream1);

// Stream synchronization
cudaStreamSynchronize(stream1); // Specific stream
cudaDeviceSynchronize(); // All streams

// Destroy stream
cudaStreamDestroy(stream1);
```

Stream Overlap 패턴

Timeline 비교

문제: Single Stream (비효율)

```
Time →  
[H→D 3ms] [Kernel 2ms] [D→H 3ms] = 8ms  
[H→D 3ms] [Kernel 2ms] [D→H 3ms] = 8ms  
[H→D 3ms] [Kernel 2ms] [D→H 3ms] = 8ms  
Total: 24ms (GPU 6ms ⇄ 3 = 25% ⇄ )
```

해결: Multi-Stream (효율적)

```
Stream 0: [H→D][Kernel][D→H]  
Stream 1: [H→D][Kernel][D→H]  
Stream 2: [H→D][Kernel][D→H]  
Total: 10ms (GPU 6ms ⇄ 2 = 60% ⇄ )  
⇨ 2.4ms!
```

효율적인 Stream 사용법

Depth-First vs Breadth-First

Depth-First (Good)

```
for (int i = 0; i < nStreams; i++) {  
    cudaMemcpyAsync(d[i], h[i], size, H2D, stream[i]);  
    kernel<<<grid, block, 0, stream[i]>>>(d[i]);  
    cudaMemcpyAsync(h[i], d[i], size, D2H, stream[i]);  
}  
// Each stream forms complete pipeline
```

Breadth-First (Bad)

```
for (int i = 0; i < nStreams; i++)  
    cudaMemcpyAsync(d[i], h[i], size, H2D, stream[i]);  
for (int i = 0; i < nStreams; i++)  
    kernel<<<grid, block, 0, stream[i]>>>(d[i]);  
for (int i = 0; i < nStreams; i++)  
    cudaMemcpyAsync(h[i], d[i], size, D2H, stream[i]);  
// Pipeline formation failed
```

Pinned Memory 와 Stream

Pinned Memory 필수!

```
// Regular memory: No overlap
float *h_data = (float*)malloc(size);

// Pinned memory: Enable overlap
float *h_pinned;
cudaHostAlloc(&h_pinned, size, cudaHostAllocDefault);
// Or
cudaMallocHost(&h_pinned, size);

// Free after use
cudaFreeHost(h_pinned);
```

성능 영향

Memory Type	Transfer	Overlap	Speed
Pageable	3 GB/s	No	1x
Pinned	12 GB/s	Yes	4x

Stream 성능 최적화

최적 Stream 수 결정

```
// Check GPU properties
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);

// Number of concurrent kernels possible
int maxConcurrentKernels = prop.concurrentKernels;

// Recommended number of streams
```

성능 향상 체크리스트

항목	효과	중요도
Pinned Memory	4x 전송 속도	필수
Stream 개수 최적화	2x 효율	높음
Depth-first 순서	1.5x 개선	중간
Chunk 크기 조정	1.3x 개선	중간

실전 예시: 비디오 처리

- 문제: 4K 비디오 30fps 실시간 처리
- 단순 방법: 100ms/frame (실패)
- Stream 사용: 25ms/frame (성공!)

스트림 동기화 필요성

스트림 간 동기화는 작업의 올바른 순서를 보장하는 데 중요합니다.

문제: 다른 스트림의 결과를 기다려야 하는 경우

예시: Stream A의 계산 결과를 Stream B에서 사용해야 할 때

스트림 동기화 방법 비교

StreamSynchronize

```
// CPU로 대기하는 코드  
cudaStreamSynchronize(stream1);  
process_results();
```

특징:

- **블로킹**: CPU가 완전히 대기
- **단순함**: 사용하기 쉬움
- **성능**: CPU 유휴 시간 발생

사용 시기: 반드시 결과가 필요한 경우

Event 기반 동기화

효율적인 동기화 방법

```
cudaEvent_t event;
cudaEventCreate(&event);

// 쿠데어 쿠데어
kernel1<<<grid, block, 0, stream1>>>();
cudaEventRecord(event, stream1);

// 쿠데어 쿠데어 쿠데어
cudaStreamWaitEvent(stream2, event, 0);
kernel2<<<grid, block, 0, stream2>>>();
```

특징:

- **비블로킹:** CPU는 다른 작업 가능
- **효율적:** GPU 내부에서만 동기화
- **유연함:** 세밀한 의존성 제어

고급 스트림 관리

우선순위 스트림

```
int minPriority, maxPriority;  
cudaDeviceGetStreamPriorityRange(&minPriority, &maxPriority);  
  
cudaStream_t highPriorityStream;  
cudaStreamCreateWithPriority(&highPriorityStream,  
                           cudaStreamNonBlocking,  
                           maxPriority);
```

성능 팁: 우선순위가 높은 스트림이 더 빨리 스케줄링됩니다.

스트림과 동시 커널 실행

최신 GPU는 여러 커널을 동시에 실행할 수 있습니다.

동시 실행 가능한 커널들

```
// Kernels processing different data
__global__ void kernel_A(float *data, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        for (int i = 0; i < 100; i++) {
            data[idx] = sqrtf(data[idx]) + sinf(data[idx]);
        }
    }
}

__global__ void kernel_B(float *data, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        for (int i = 0; i < 100; i++) {
            data[idx] = logf(data[idx] + 1.0f) * 2.0f;
        }
    }
}
```

동시 커널 실행 구현

```
void concurrent_kernels() {
    const int N = 1 << 20;
    float *d_A, *d_B;
    cudaMalloc(&d_A, N * sizeof(float));
    cudaMalloc(&d_B, N * sizeof(float));

    cudaStream_t streamA, streamB;
    cudaStreamCreate(&streamA);
    cudaStreamCreate(&streamB);

    // Two kernels can execute simultaneously
    kernel_A<<<gridSize, blockSize, 0, streamA>>>(d_A, N);
    kernel_B<<<gridSize, blockSize, 0, streamB>>>(d_B, N);

    cudaStreamSynchronize(streamA);
    cudaStreamSynchronize(streamB);

    cudaFree(d_A);
    cudaFree(d_B);
}
```

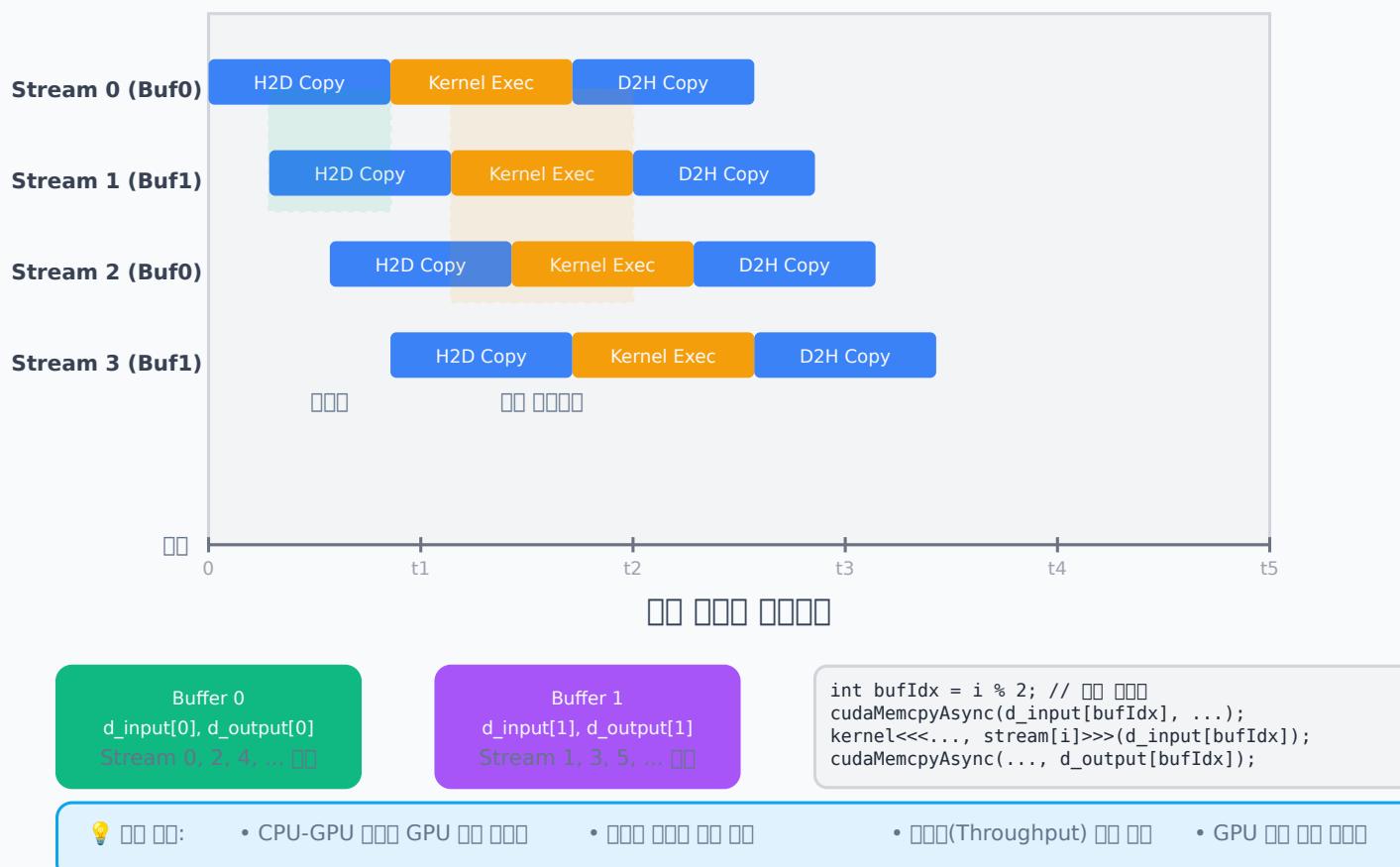
동시 실행 조건: 서로 다른 스트림에서 실행되고, 리소스 충돌이 없어야 함

실습: 파이프라인 처리

데이터를 청크로 나누어 처리하는 파이프라인을 구현해봅시다.

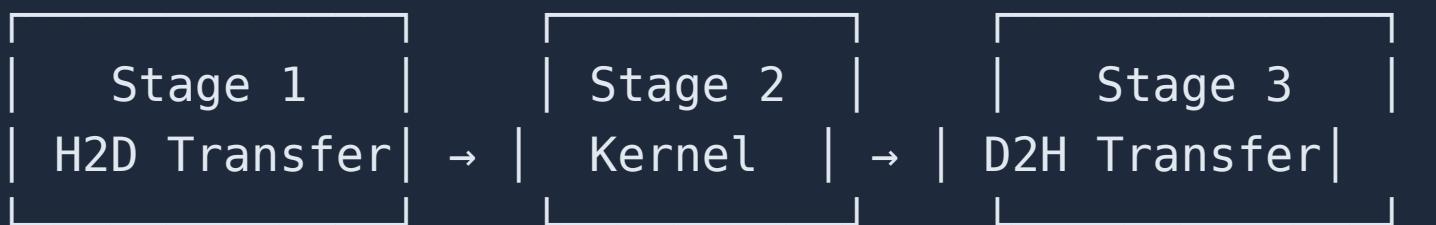
Pipeline Processing with Double Buffering

데이터를 청크로 나누어 처리하는 파이프라인 구현



파이프라인 처리 핵심 기법

파이프라인 구조 설계



핵심 요소:

- 데이터 분할: 큰 데이터를 청크로 분할
- 스트림 병렬 처리: 각 청크를 독립 처리
- 더블 버퍼링: 전송과 처리 동시 진행
- Pinned Memory: 비동기 전송 지원

파이프라인 구현 코드

구현 패턴

```
// Pinned memory //  
float* h_data;  
cudaMallocHost(&h_data, total_size);  
  
// 데이터 분할 //  
for (int i = 0; i < num_chunks; i++) {  
    int stream_id = i % num_streams;  
  
    // Stage 1: H2D Transfer  
    cudaMemcpyAsync(d_chunk[stream_id],  
        h_data + i * chunk_size, chunk_size,  
        cudaMemcpyHostToDevice,  
        streams[stream_id]);  
  
    // Stage 2: Kernel Execution  
    process_kernel<<<grid, block, 0,  
        streams[stream_id] >>>(d_chunk[stream_id]).
```

파이프라인 성능 효과

- **오버래핑:** 전체 처리 시간 30-50% 단축
- **GPU 활용률:** 60% → 90% 향상
- **대역폭 효율:** PCIe + GPU 동시 활용

파이프라인 구현 핵심 API

```
// 데이터 복사
cudaMemcpyAsync(d_data, h_data, size,
                cudaMemcpyHostToDevice, stream);

// 케일러
kernel<<<blocks, threads, 0, stream>>>(d_data);

// 동기화
cudaStreamSynchronize(stream);
```

핵심: 각 API 호출 시 적절한 스트림을 지정하여 병렬 처리

Hyper-Q와 동적 병렬성

최신 GPU는 더 고급 기능을 제공합니다.

Hyper-Q: 여러 CPU 스레드에서 GPU 사용

```
void hyperq_example() {
    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();
        cudaStream_t stream;
        cudaStreamCreate(&stream);

        float *d_data;
        cudaMalloc(&d_data, 1024 * 1024 * sizeof(float));

        // Each CPU thread uses GPU independently
        kernel<<<100, 256, 0, stream>>>(d_data, tid);

        cudaStreamSynchronize(stream);
        cudaFree(d_data);
        cudaStreamDestroy(stream);
    }
}
```

콜백 함수

스트림 완료 시 콜백 실행

```
void CUDART_CB myCallback(cudaStream_t stream, cudaError_t status,
                           void *userData) {
    printf("Stream %p completed with status %d\n", stream, status);
    // Post-processing work
}

void callback_example() {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    kernel<<<100, 256, 0, stream>>>();

    // Call callback when stream work completes
    cudaStreamAddCallback(stream, myCallback, nullptr, 0);

    cudaStreamDestroy(stream);
}
```

콜백 주의사항: CUDA API 호출 금지, 간단한 작업만 수행

Multi-Stream 패턴

효율적인 스트림 활용

Multi-Stream 동기화

```
// 모든 스트림 동기화
for (int i = 0; i < nStreams; i++) {
    cudaStreamSynchronize(streams[i]);
}
```

Stream 우선순위와 의존성

Priority Streams

```
// CUDA Stream API
cudaStream_t highPriority, lowPriority;
int leastPriority, greatestPriority;

cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);

cudaStreamCreateWithPriority(&highPriority, cudaStreamNonBlocking, greatestPriority);
cudaStreamCreateWithPriority(&lowPriority, cudaStreamNonBlocking, leastPriority);

// CUDA kernel launch with high priority stream
criticalKernel<<<blocks, threads, 0, highPriority>>>();

// CUDA kernel launch with low priority stream
backgroundKernel<<<blocks, threads, 0, lowPriority>>>();
```

Stream 간 동기화

```
// Event로 Stream을 관리
cudaEvent_t event;
cudaEventCreate(&event);

// Stream 1로 Event를 기록
kernel1<<<blocks, threads, 0, stream1>>>();
cudaEventRecord(event, stream1);

// Stream 2로 Event를 기다림
cudaStreamWaitEvent(stream2, event, 0);
kernel2<<<blocks, threads, 0, stream2>>>();
```

Best Practice: Stream 수는 보통 2-8개가 최적

Stream Callbacks 실무 활용

비동기 실행 제어

```
// Define callback function
void CUDART_CB myStreamCallback(cudaStream_t stream, cudaError_t status, void* data) {
    printf("Stream %p completed with status %d\n", stream, status);

    // Process user data
    int* counter = (int*)data;
    (*counter)++;

    // Can trigger next task
    if (*counter == 4) {
        printf("All streams completed!\n");
        // Start post-processing work
    }
}

// Add callback to stream
int completedStreams = 0;

for (int i = 0; i < nStreams; i++) {
```

Pipeline 패턴 구현

3단계 파이프라인

```
// Pipeline: input → processing → output
void pipeline(float* h_input, float* h_output, int totalBatches) {
    cudaStream_t stream_h2d, stream_kernel, stream_d2h;
    cudaStreamCreate(&stream_h2d);
    cudaStreamCreate(&stream_kernel);
    cudaStreamCreate(&stream_d2h);

    cudaEvent_t event_h2d[2], event_kernel[2];
    for (int i = 0; i < 2; i++) {
        cudaEventCreate(&event_h2d[i]);
        cudaEventCreate(&event_kernel[i]);
    }

    for (int i = 0; i < totalBatches; i++) {
        int pingpong = i % 2;

        // Stage 1: H2D transfer
        cudaMemcpyAsync(d_input[pingpong], &h_input[i * batchSize],
                       batchBytes, cudaMemcpyHostToDevice, stream_h2d);
    }
}
```

Pipeline 패턴 구현 (계속)

```
// Stage 3: D2H transfer (wait for previous kernel completion)
if (i > 1) {
    cudaStreamWaitEvent(stream_d2h, event_kernel[1-pingpong], 0);
    cudaMemcpyAsync(&h_output[(i-2) * batchSize], d_output[1-pingpong],
                   batchBytes, cudaMemcpyDeviceToHost, stream_d2h);
}
}
```

Performance improvement: Up to 3x throughput increase with pipeline

Part 2 학습 내용 정리

이 장에서 학습한 주요 내용들을 정리해보겠습니다.

목표: CUDA 중급 기법의 핵심 개념 복습

핵심 학습 내용 - 메모리 최적화

메모리 최적화 기법

1. Coalesced 접근

- 인접 스레드의 순차 메모리 접근
- 메모리 대역폭 극대화

2. Shared Memory 활용

- Bank Conflict 회피 기법
- 데이터 재사용 최적화

3. 캐시와 데이터 레이아웃

- AoS vs SoA 구조 선택
- 텍스처 메모리 활용

핵심 학습 내용 - 알고리즘 패턴

병렬 알고리즘 패턴

1. Reduction

- Tree-based 병렬 누적 연산
- $O(\log n)$ 시간 복잡도 달성

2. Scan (Prefix Sum)

- Hillis-Steele vs Blelloch 알고리즘
- 다양한 병렬 알고리즘의 기반

3. Stream Compaction

- 조건부 데이터 필터링
- Scan을 활용한 효율적 구현

스트림과 동시 실행

비동기 처리 기법

- **스트림**: GPU 작업의 비동기 실행 단위
- **오버래핑**: 데이터 전송과 커널 실행 동시 진행
- **파이프라인**: 멀티 스트림을 활용한 처리량 향상

핵심: 메모리 최적화 + 병렬 알고리즘 + 비동기 처리 = 고성능 CUDA

다음 단계: Part 3 준비

Part 3 예고: 동적 메모리 관리, Unified Memory, 템플릿 프로그래밍, 고급 성능 분석 기법, 텐서 코어 등 특수 하드웨어 활용법