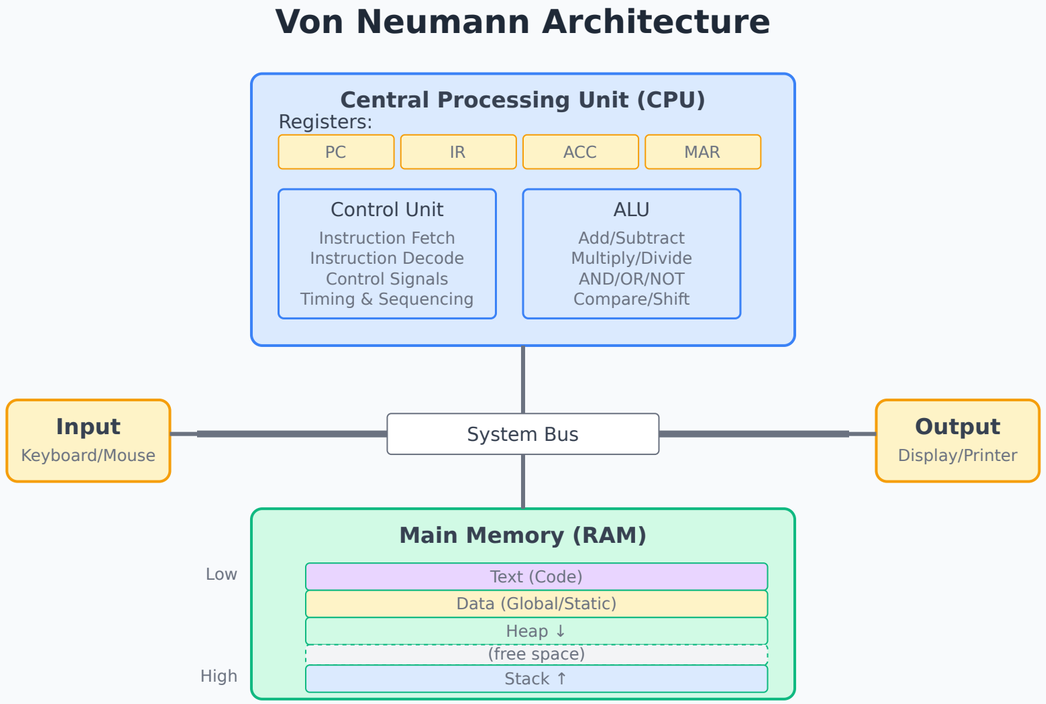
Part 0. Parallel Programming Fundamentals

Computer Architecture and Parallelization Theory

CUDA를 시작하기 전 필수 기초 지식

폰 노이만 구조 - 현대 컴퓨터의 기초

왜 GPU가 필요한지 이해하기



폰 노이만 병목 (Von Neumann Bottleneck)

구성요소	역할	속도	문제점
CPU	연산 처리	5 GHz	매우 빠름
Memory	데이터 저장	3200 MHz	CPU보다 느림
Bus	데이터 전송	제한적	**병목 지점**

핵심 문제: CPU는 대부분 메모리 대기에 시간 소비!

즉시 실습: 병목 계산

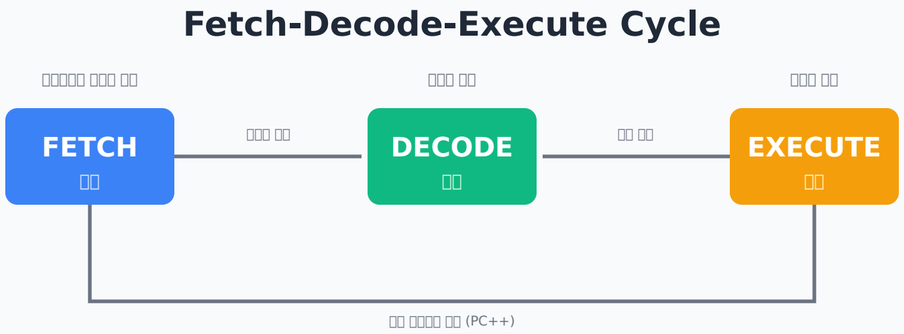
시나리오: 100만 개 float 배열 처리

- CPU 연산: 1 cycle/element = 0.2μs (5GHz)
- 메모리 접근: 100 cycles/element = 20μs
- **대기 시간 비율**: 99%!

해결책: 병렬 처리로 대기 시간 숨기기 → GPU!

CPU의 기본 동작: Fetch-Decode-Execute 사이클

CPU는 어떻게 명령을 실행하는가?



Fetch-Decode-Execute 상세

1. Fetch (인출)

- **Program Counter(PC)**에서 다음 명령어 주소 확인
- 메모리에서 명령어를 **Instruction Register(IR)**로 가져옴
- PC를 다음 명령어 주소로 업데이트

2. Decode (해독)

- Control Unit이 명령어 해석
- 필요한 데이터 위치 파악
- 실행할 연산 결정

3. Execute (실행)

- ALU에서 실제 연산 수행
- 결과를 레지스터나 메모리에 저장
- 다음 사이클 준비

왜 이것이 중요한가?

CPU의 한계

□ 번에 하나의 명령만 처리
→ 순차적 실행의 근본적 □계

GPU의 해결책

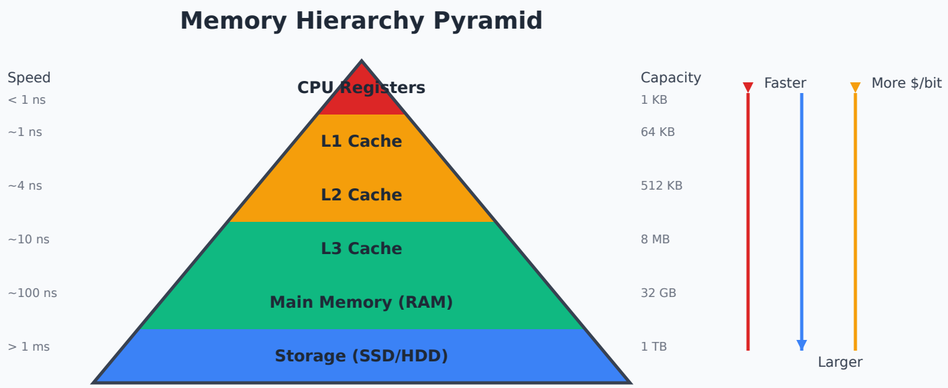
수천 개의 간단□ 실행 유닛
→ 같은 명령을 여러 데이터에 동시 실행 (SIMT)

성능 차이의 원인

CPU	GPU
복잡한 명령 처리	단순한 명령 반복
깊은 파이프라인	얕은 파이프라인
분기 예측 중요	분기 회피 중요

메모리 계층 구조 - 속도와 용량의 트레이드오프

왜 캐시가 중요한가?



메모리 계층별 특성

레벨	크기	속도	지연시간	가격/GB
Register	<1 KB	1 cycle	0.2 ns	-
L1 Cache	32-64 KB	4 cycles	0.9 ns	-
L2 Cache	256-512 KB	12 cycles	3 ns	-
L3 Cache	8-32 MB	40 cycles	10 ns	$10,000
RAM	16-64 GB	200 cycles	50 ns	$10
SSD	512 GB-2 TB	10,000 cycles	100 µs	$0.1

지역성 원리 (Locality Principle)

종류	설명	예시	캐시 효과
시간 지역성	최근 접근한 데이터 재사용	Loop 변수	90% Hit
공간 지역성	인접 데이터 연속 접근	배열 순회	95% Hit

즉시 실습: 캐시 미스 비용

문제: 100만 요소 stride 접근

- Sequential (stride=1): 캐시라인 활용 100% → 10ms
- Random (stride=random): 캐시라인 활용 3% → 300ms
- **성능 차이**: 30배!

CPU 발전과 한계 - 무어의 법칙 종말

클럭 속도의 물리적 한계

시대	클럭 속도	전력 소비	주요 문제
1990-2000	25MHz → 1GHz (40배)	10W → 30W	발열 관리 가능
2000-2010	1GHz → 3.8GHz (3.8배)	30W → 100W	Power Wall
2010-2024	3.8GHz → 5GHz (1.3배)	100W → 250W	**한계 도달**

3가지 Wall (장벽)

장벽	원인	결과
Power Wall	전력 밀도 한계 (100W/cm^2)	더 이상 클럭 증가 불가
Memory Wall	CPU-Memory 속도 격차	성능 향상 제한
ILP Wall	명령어 병렬성 한계	단일 스레드 성능 정체

즉시 실습: 성능 향상 계산

문제: 다음 10년 성능 예측

- 클럭 향상: 5GHz → 5.5GHz (1.1배)
- 코어 증가: 8 → 128 cores (16배)

핵심: 수직 확장(클럭) 한계 → 수평 확장(병렬)으로!

결론: 병렬화만이 답! → GPU (10,000 cores)

성능 측정 지표 - Latency vs Throughput

Latency (지연시간)

한 작업을 완료하는 시간

```
Task: [====] 4ms
```

- 개별 작업 속도
- 단위: ms, μs, ns
- CPU 최적화 목표
- 예: 게임 응답속도

계산: 처리시간 / 작업

Throughput (처리량)

단위 시간당 처리 작업 수

```
Task1: [====]
Task2:  [====]
Task3:   [====]
Task4:    [====]
Total: 7ms for 4 tasks
```

- 전체 처리 능력
- 단위: ops/sec, GB/s
- GPU 최적화 목표
- 예: 서버 처리량

계산: 작업수 / 시간

CPU vs GPU 비교

프로세서	Latency	Throughput	적합한 작업
CPU (1 core)	1ms/task	1000 tasks/s	순차 처리
GPU (1000 cores)	10ms/task	100,000 tasks/s	**병렬 처리**

병렬화 기초 개념 - 동시성의 이해

병렬(Parallel) vs 동시(Concurrent)

구분	Parallel	Concurrent
정의	물리적으로 동시 실행	논리적으로 동시 실행
하드웨어	다중 코어 필수	단일 코어 가능
실행	진짜 동시	Time-slicing
예시	GPU 10,000 threads	OS multitasking

병렬화 레벨과 실습

병렬화 레벨

레벨	범위	GPU 활용
Bit-level	32-bit → 64-bit	-
ILP	명령어 파이프라인	SM 내부
DLP	SIMD 연산	Warp 실행
TLP	멀티스레딩	**CUDA threads**
Task-level	독립 작업	Kernel 분리

병렬화 가능성 판단

작업	병렬화	이유
배열 합계	가능 (Reduction)	독립적 덧셈
정렬	부분 가능	비교 의존성
피보나치	어려움	순차 의존성
이미지 필터	완벽	픽셀 독립

Amdahl vs Gustafson - 병렬화의 두 관점

Amdahl의 법칙

"고정 문제 크기"

```
Speedup = 1 / (s + p/N)
```

- s: 순차 부분 (병렬 불가)
- p: 병렬 부분 (1-s)
- N: 프로세서 수

예시: 95% 병렬화

- 10 cores: 6.9x
- 100 cores: 16.8x
- ∞ cores: **20x (한계!)**

비관적 관점: "순차 부분이 한계 결정"

Gustafson의 법칙

"확장 가능한 문제"

```
Speedup = s + p×N
```

- s: 순차 부분
- p: 병렬 부분
- N: 프로세서 수

예시: 5% 순차, 95% 병렬

- 10 cores: 9.5x
- 100 cores: 95x
- 1000 cores: **950x**

낙관적 관점: "문제를 키우면 확장 가능"

언제 어느 법칙을 적용할까?

실습: 어느 법칙이 맞을까?

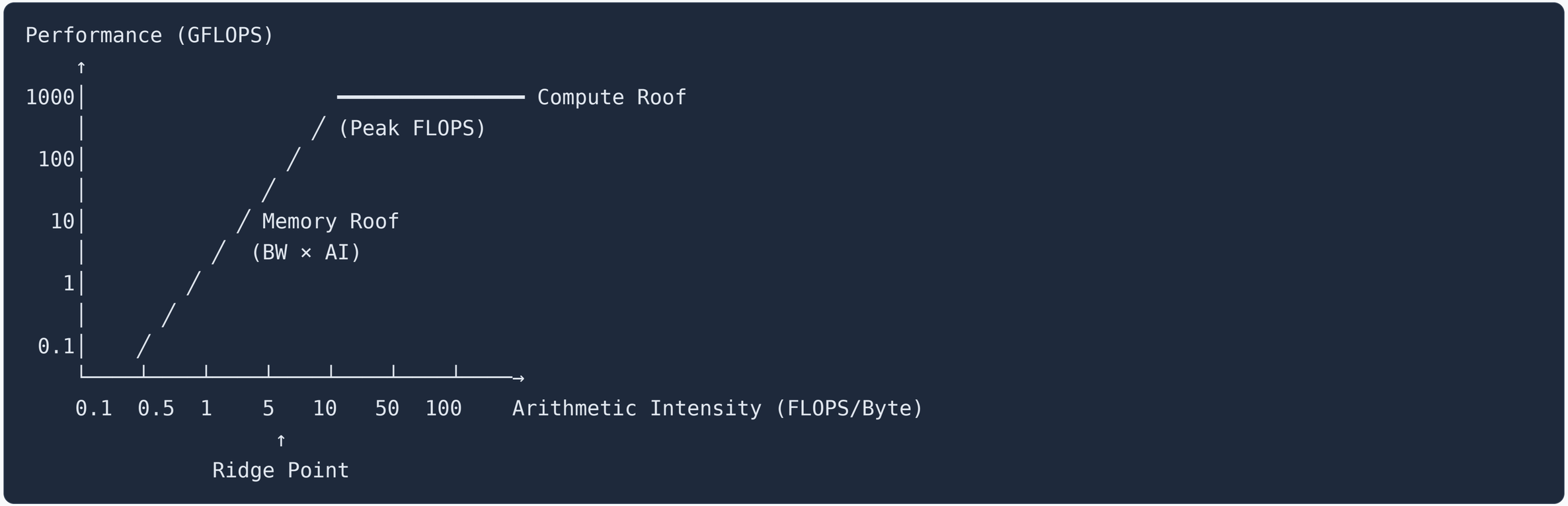
시나리오	적용 법칙	이유
이미지 1장 처리	Amdahl	고정 크기
빅데이터 분석	**Gustafson**	데이터 확장 가능
게임 프레임 렌더링	Amdahl	16ms 제약
AI 모델 학습	**Gustafson**	배치 크기 증가

GPU의 철학: Gustafson처럼 생각하라!

- 더 많은 데이터 처리
- 더 높은 해상도
- 더 복잡한 시뮬레이션

Roofline Model - 성능 한계 시각화

성능 병목 지점 찾기



Roofline 활용 - 하드웨어별 최적화

하드웨어 스펙과 Ridge Point

GPU	Peak FLOPS	Bandwidth	Ridge Point
RTX 3090	35.6 TFLOPS	936 GB/s	38 FLOPS/B
A100	19.5 TFLOPS	1555 GB/s	12.5 FLOPS/B
H100	67 TFLOPS	3350 GB/s	20 FLOPS/B

알고리즘별 최적화 방향

알고리즘	AI 계산	병목	최적화 전략
벡터 덧셈	0.08	**Memory**	Coalescing
행렬 곱셈	N	**Compute** (N>38)	Tensor Core
Reduction	log(N)/4N	**Memory**	Shared Memory
FFT	5log(N)/8	**Balanced**	둘 다 최적화

병렬 프로그래밍 패턴 - 재사용 가능한 해결책

주요 병렬 패턴과 GPU 매핑

패턴	설명	GPU 구현
Map	각 요소 독립 변환	`kernel<<<blocks,threads>>>`
Reduce	여러 값을 하나로	Warp shuffle
Scan	Prefix sum	Work-efficient
Fork-Join	분할 후 합치기	Kernel launch/sync
Pipeline	단계별 처리	CUDA Streams
Stencil	이웃 요소 접근	Shared memory

패턴별 성능과 선택

패턴별 성능 특성

패턴	병렬 효율	메모리 패턴	GPU 적합도
Map	100%	Coalesced	최고
Reduce	O(log n)	Tree access	높음
Scan	O(log n)	Bank conflict 주의	높음
Stencil	높음	Halo exchange	높음

문제별 최적 패턴

문제	최적 패턴	이유
이미지 밝기 조정	**Map**	픽셀 독립
배열 최댓값	**Reduce**	집계 연산
누적 합	**Scan**	Prefix 필요
블러 필터	**Stencil**	이웃 픽셀

데이터 의존성 - 병렬화의 적

의존성 종류 (Hazards)

종류	약어	설명
Read After Write	RAW	진짜 의존성
Write After Read	WAR	역의존성
Write After Write	WAW	출력 의존성
Read After Read	RAR	의존성 없음

예시:

- RAW: `b=a+1; c=b+1;`
- WAR: `b=a+1; a=c+1;`
- WAW: `a=b+1; a=c+1;`
- RAR: `b=a+1; c=a+1;` ✓병렬가능

의존성 그래프 분석

```
A: x = input + 1   [□□] → □□
B: y = input * 2   [□□] → □□
C: z = x + y       [A,B □□]
D: w = z * z       [C □□]
```

최소 실행: 3단계 (A&B → C → D)

의존성 제거 방법

문제	해결
RAW 의존	Grid-stride loop
WAW on sum	Reduction tree
WAR	임시 변수 사용

동기화 메커니즘 - 질서있는 병렬 실행

동기화 개념 비교

개념	CPU 예시	GPU 예시	핵심 차이
Mutex	`pthread_mutex_lock()`	N/A	GPU는 lock-free 선호
Barrier	`pthread_barrier_wait()`	`__syncthreads()`	GPU가 훨씬 빠름 (1-5 cycles)
Atomic	`#pragma omp atomic`	`atomicAdd()`	GPU는 하드웨어 지원
Fence	`std::atomic_thread_fence()`	`__threadfence()`	메모리 순서 보장

동기화 오버헤드 비교

메커니즘	CPU 비용	GPU 비용	사용 시나리오
Mutex	100-1000 cycles	N/A	Critical section
Barrier	50-500 cycles	1-5 cycles	Phase 동기화
Atomic	10-100 cycles	30-300 cycles	공유 변수
Fence	10 cycles	5 cycles	메모리 순서

Race Condition - 병렬 프로그래밍의 함정

Race Condition 발생 원리

단계	Thread A	Thread B	공유 변수	문제점
1	Read: counter=0		0	
2		Read: counter=0	0	두 스레드가 같은 값 읽음
3	Add: 0+1=1		0	
4		Add: 0+1=1	0	
5	Write: counter=1		1	
6		Write: counter=1	1	**결과: 1 (예상: 2)**

핵심: Read-Modify-Write가 원자적(atomic)이지 않음

해결 방법 비교

방법	장점	단점	적합 상황
Mutex/Lock	완전 보호	느림, 오버헤드	Critical section
Atomic 연산	빠름, 간단	제한적 연산	카운터, 플래그
Lock-free	매우 빠름	복잡함	고성능 요구
순차 실행	안전	병렬성 없음	디버깅용

GPU에서의 Race Condition

상황	문제	해결책	성능 영향
Global memory 쓰기	여러 스레드 동시 쓰기	`atomicAdd()`	10-100x 느려짐
Shared memory 쓰기	Bank conflict + race	`__syncthreads()`	1-5 cycles
Warp divergence	조건부 쓰기 충돌	Warp-level primitives	Warp 내 동기화
카운터 업데이트	Lost updates	Atomic operations	하드웨어 지원

메모리 일관성 모델 - 언제 변경이 보이는가?

일관성 모델 비교

모델	보장 수준	성능
Sequential Consistency	프로그램 순서 보장	느림
Total Store Order (TSO)	쓰기 순서 보장	중간
Relaxed Consistency	최소 보장	빠름
CUDA (Weak Ordering)	동기화 시점만 보장	매우 빠름

메모리 순서 문제와 해결

Producer-Consumer 문제

```
Thread 0: data = 42; flag = true;
Thread 1: if(flag) read(data);  // data□ 0□ □ □□!
```

문제: flag가 먼저 보일 수 있음 → data가 아직 안 보임

Memory Fence 종류

Fence	범위	용도
`__threadfence_block()`	Block 내	Shared memory 동기화
`__threadfence()`	Device 전체	Global memory 동기화
`__threadfence_system()`	System 전체	Host-Device 동기화

해결: `data = 42; __threadfence(); flag = true;`

False Sharing - 보이지 않는 성능 킬러

False Sharing 발생 원리

같은 캐시라인(64B)에 있는 변수를 다른 스레드가 수정

상황	Thread 0	Thread 1	성능 영향
초기	count0++	count1++	같은 캐시라인
T0 쓰기	캐시 업데이트	캐시 무효화	100x 느려짐
T1 쓰기	캐시 무효화	캐시 업데이트	Ping-pong

False Sharing 해결 방법

해결 방법

방법	설명	효과
Padding	변수 간 64B 간격	캐시라인 분리
Alignment	64B 경계 정렬	자동 분리
Local Copy	로컬 변수 사용	공유 최소화

GPU에서의 False Sharing

메모리	단위	해결책
L1 Cache	128B line	128B 정렬
Shared Memory	4B bank	Bank conflict 회피
Global Memory	32B sector	Coalescing

성능 개선: Padding 없음(1x) → 64B padding(10x) → 128B padding(10x)

Part 0 정리 - CUDA를 위한 준비 완료

핵심 개념 체크리스트

영역	개념	CUDA 연관성
컴퓨터 구조	폰 노이만 병목	GPU가 필요한 이유
메모리 계층	Cache, 지역성	Coalesced Access
CPU 한계	Power/Memory/ILP Wall	병렬화 필수
성능 지표	Latency vs Throughput	GPU 최적화 목표
병렬화 이론	Amdahl/Gustafson	확장성 이해
Roofline	AI, Ridge Point	최적화 방향
병렬 패턴	Map, Reduce, Scan	Kernel 설계
의존성	RAW, WAR, WAW	병렬화 가능성
동기화	Barrier, Atomic	__syncthreads()
Race Condition	동시 접근 문제	atomicAdd()
False Sharing	캐시라인 충돌	Bank Conflict

다음 단계: Part 1 CUDA 기초 → CUDA 프로그래밍 모델, GPU 코드 작성, 첫 커널 실행

CUDA Parallelization Techniques Overview

Essential Techniques Before Moving to GPU

Why These Techniques Matter

```
CPU code: for(i=0; i<1000000; i++) process(data[i]);
Time: 100ms

Simple GPU port: kernel<<<1, 1>>>(...);
Time: 200ms (slower!)

Proper parallelization: kernel<<<1000, 1000>>>(...);
Time: 0.1ms (1000x faster!)
```

Key Parallelization Techniques

1. Loop-Level Optimizations

- **Loop Unrolling** - Reduce branch overhead
- **Loop Fusion** - Combine multiple loops
- **Loop Tiling** - Improve cache locality

2. Thread-Level Parallelism

- **Work Distribution** - Divide work among threads
- **Thread Cooperation** - Shared memory patterns
- **Synchronization** - Barriers and atomics

3. Memory Optimizations

- **Coalescing** - Sequential memory access
- **Caching** - Shared/constant memory
- **Prefetching** - Hide memory latency

4. Execution Optimizations

- **Occupancy** - Maximize active warps
- **Divergence** - Minimize branch divergence
- **Streams** - Overlap computation and transfer

Loop Unrolling

Reducing Loop Overhead

Problem: Loop Control Cost

```
for (int i = 0; i < N; i++) {
    sum += data[i];  // N loop overhead iterations
}
```

- Branch instruction every iteration
- Loop counter update
- Condition check

Loop Unrolling Example

4x Unrolling

```
for (int i = 0; i < N; i += 4) {
    sum += data[i];
    sum += data[i+1];
    sum += data[i+2];
    sum += data[i+3];
}
```

Performance Impact

Aspect	Before	After	Improvement
Branch Instructions	N	N/4	75% reduction
Memory Access	Sequential	Sequential burst	Better cache
ILP Opportunity	Low	High	Compiler optimization

Advanced Unrolling Techniques

Template-based Unrolling

```
template<int UNROLL>
__global__ void kernel(float *data, int n) {
    float sum = 0;
    #pragma unroll UNROLL
    for(int i = 0; i < UNROLL; i++) {
        sum += data[idx + i];
    }
}
```

CUDA Compiler Directives

- `#pragma unroll` - Full unrolling
- `#pragma unroll N` - Unroll N times
- `#pragma unroll 1` - Disable unrolling

Thread-Level Parallelism

Distributing Work Across Threads

Sequential Processing Problem

```
// CPU: 1 thread processes 1 million items
for (int i = 0; i < 1000000; i++)
    process(data[i]);  // 100ms total
```

Parallel Solution

```
__global__ void kernel(float *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    process(data[idx]);  // Each thread processes 1 item
}

// 1000 blocks × 1000 threads = 1 million concurrent
kernel<<<1000, 1000>>>(data);  // 0.1ms total
```

Work Distribution Strategies

Static Distribution

```
// Each thread gets fixed portion
int tid = blockIdx.x * blockDim.x + threadIdx.x;
int stride = gridDim.x * blockDim.x;
for(int i = tid; i < N; i += stride) {
    process(data[i]);
}
```

Dynamic Distribution

```
// Work stealing approach
__shared__ int work_counter;
if(threadIdx.x == 0) work_counter = atomicAdd(global_counter, 32);
__syncthreads();
int my_work = work_counter + threadIdx.x;
```

Thread Cooperation Patterns

Reduction Pattern

```
__shared__ float shared[256];
shared[tid] = data[global_tid];
__syncthreads();

// Tree reduction
for(int s = blockDim.x/2; s > 0; s >>= 1) {
    if(tid < s) {
        shared[tid] += shared[tid + s];
    }
    __syncthreads();
}
```

Producer-Consumer Pattern

```
// Producer threads
if(tid < NUM_PRODUCERS) {
    produce_data();
}
// Consumer threads
else {
    consume_data();
}
```