

Part 1. CUDA 입문: GPU 프로그래밍의 첫걸음

GPU 프로그래밍의 세계로!

문제: 내 프로그램이 너무 느리다!

실제 사례

```
# Deep learning model training
for epoch in range(1000):
    for batch in dataset: # 1 million data points
        forward_pass()      # 10ms
        backward_pass()     # 20ms

# Total time: 1000 * 1M * 30ms = 347 hours (14 days!)
```

왜 이렇게 느린가?

- CPU는 순차 처리에 최적화
- 병렬 연산에는 비효율적
- 대부분의 코어가 놀고 있음

CUDA가 해결책

```
// GPU parallelization
kernel<<<1000, 1024>>>(...); // Process 1 million simultaneously
// Total time: 8 hours (42x faster!)
```

왜 CUDA인가? 다른 대안과 비교

병렬화 기술 비교

기술	성능	학습 난이도	호환성	생태계
OpenMP	2-8x	낮음	CPU만	제한적
OpenCL	10-50x	매우 높음	모든 GPU	복잡
CUDA	10-100x	중간	NVIDIA	강력
OpenACC	5-20x	낮음	다양	기본적

CUDA의 강점

- 성능: 최고 수준의 성능
- 생태계: cuDNN, cuBLAS, Thrust 등
- 커뮤니티: 방대한 자료와 예제
- 산업 표준: AI/ML/HPC 분야 필수

이 파트에서 배울 내용

1. 왜 병렬화가 필요한가 → 실제 문제 해결
2. CUDA가 어떻게 해결하는가 → 핵심 개념
3. 실제 코드 작성 → 즉시 적용 가능

문제: CPU가 더 이상 빨라지지 않는다!

충격적인 현실

2000년: "우리 20년 전 CPU 뭐"

2024년: "우리 20년 뭐..."

이유:

1. 더 빠른 CPU 필요 (우리)
2. 더 빠른 뭐 (우리!)

왜 CPU 속도가 정체됐나?

시대	클럭 속도	향상률
1990-2000	25MHz → 1GHz	40배
2000-2010	1GHz → 3.8GHz	3.8배
2010-2024	3.8GHz → 5GHz	1.3배

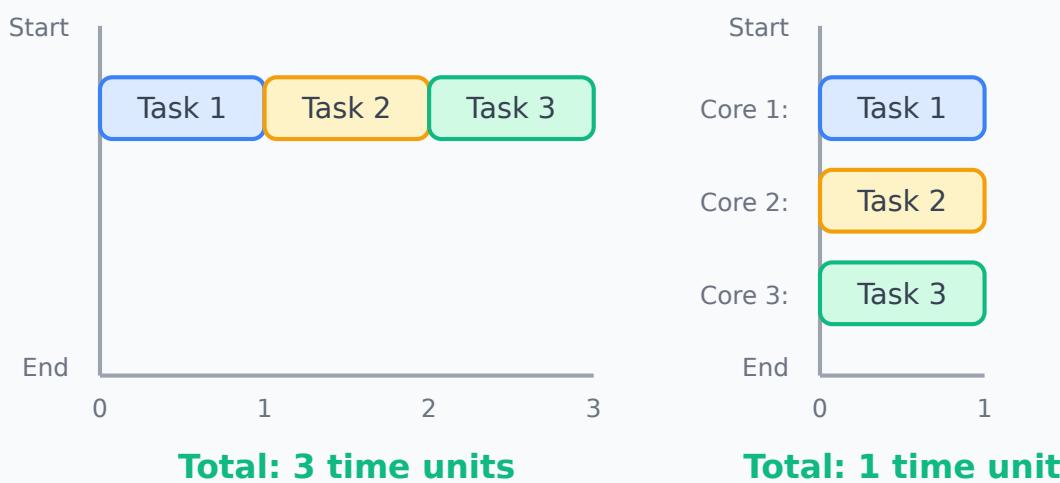
물리적 한계 도달

- 전력 벽: 더 빠름 = 열 폭발
- 발열 벽: 5GHz = 100°C (물 끓는 온도)
- 전압 벽: 더 낮추면 신호 오류

해결책: 병렬 처리

"빠른 일꾼 1명" → "보통 일꾼 1000명"

Sequential vs Parallel Execution
Sequential (Single Core) Parallel (3 Cores)



의문: 코어 많으면 무조건 빠른가?

실제 사례: 비디오 인코딩

```
# 90% 병렬화 가능, 10% 순차 처리 필수  
video = load_video()          # 순차 (10%)  
frames = split_frames()        # 병렬 가능 (90%)  
for frame in frames:  
    encode(frame)              # 병렬 처리
```

코어 수	계산식	속도 향상
2	$1/(0.1 + 0.9/2)$	1.8배
10	$1/(0.1 + 0.9/10)$	5.3배
∞	$1/0.1$	10배 (한계!)

Amdahl의 법칙이 알려주는 진실

순차 부분 10% = 최대 10배 제한

GPU가 답인 이유

- 데이터 병렬: 100% 병렬화 가능
- 같은 연산 반복: SIMD에 완벽
- 순차 부분 최소화: 0.1% 미만

결과: CPU 10배 vs GPU 1000배

왜 GPU가 AI/그래픽에만 좋다고 하는가?

실제 비교: 레스토랑 예시

CPU = 100 100 100

- 1000 100 100
- 100 1000 100
- 100 100 100

GPU = 1000 1000 1000

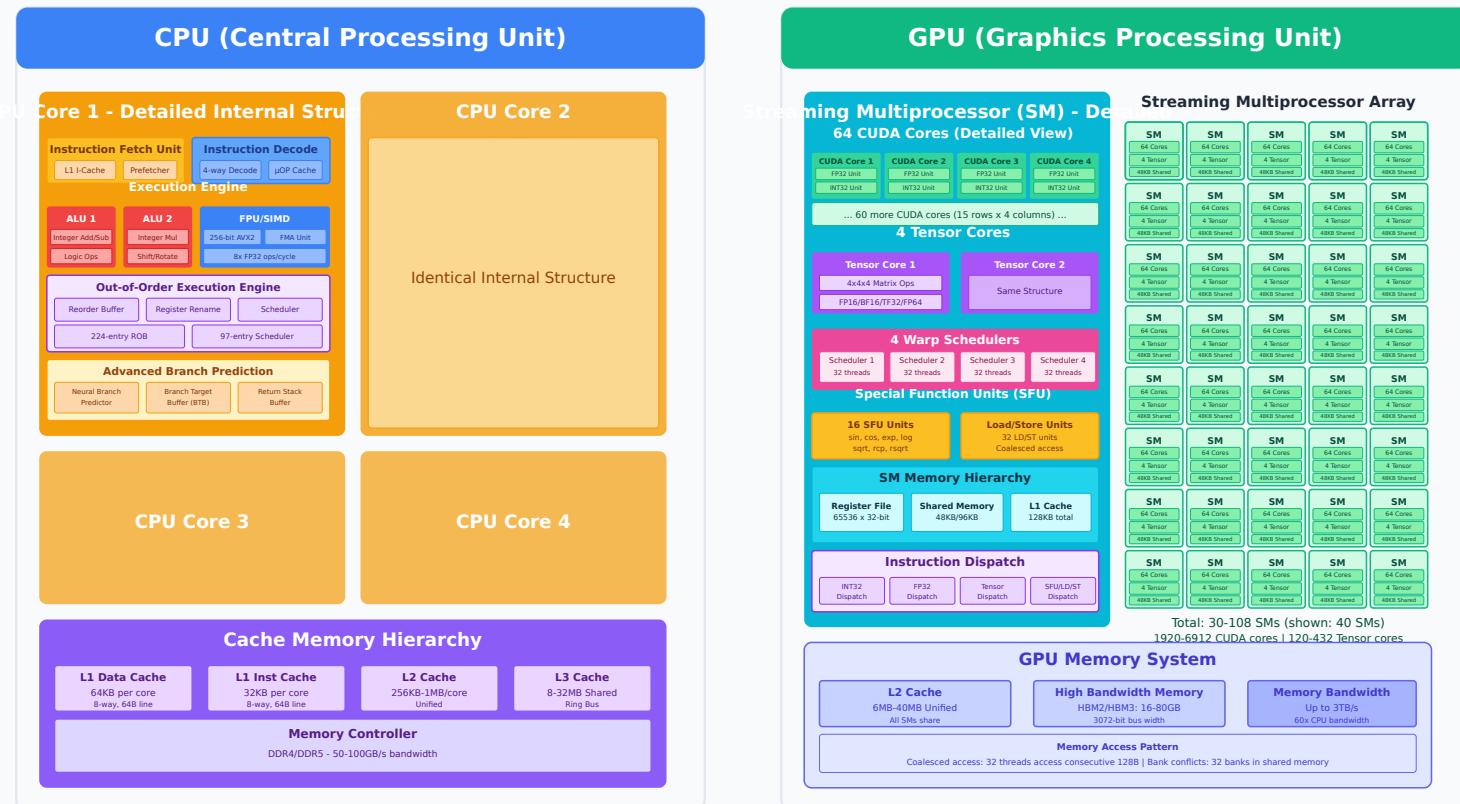
- 1000 1000 1000
- 1000 1000 1000
- 1000 1000 1000

어떤 상황에 누가 유리한가?

- 코스 요리 10개: CPU 승 (다양한 처리)
- 감자튀김 1000개: GPU 승 (같은 작업 반복)

CPU vs GPU: 하드웨어 차이

CPU vs GPU Architecture - Ultra Detailed



CPU vs GPU 아키텍처 비교

구분	CPU	GPU
코어 수	4~16개 강력한 코어	수천 개 단순한 코어
캐시	큰 캐시 (MB 단위)	작은 캐시 (KB 단위)
제어 논리	복잡한 예측/분기	단순한 제어
처리 방식	순차적 처리	병렬 처리
최적화 대상	Latency	Throughput
적합한 작업	복잡한 논리, 분기	대규모 단순 연산

설계 철학의 차이

CPU: 강력한 단일 처리

- 각 코어가 고성능 (OoO, 분기 예측, 캐시)
- 복잡한 논리 처리 가능
- 대기 시간(Latency) 최소화

GPU: 대규모 병렬 처리

- 단순한 코어 수천 개
- 같은 명령 동시 실행(SIMT)
- 처리량(Throughput) 최대화

적합한 사용 사례

작업 특성	CPU	GPU	이유
if/else 많음	O	X	Warp divergence
동일 연산 반복	X	O	SIMT 효율
메모리 랜덤 접근	O	X	Cache 활용
대용량 행렬	X	O	병렬성

병렬 처리 실제 예시: 이미지 흑백 변환

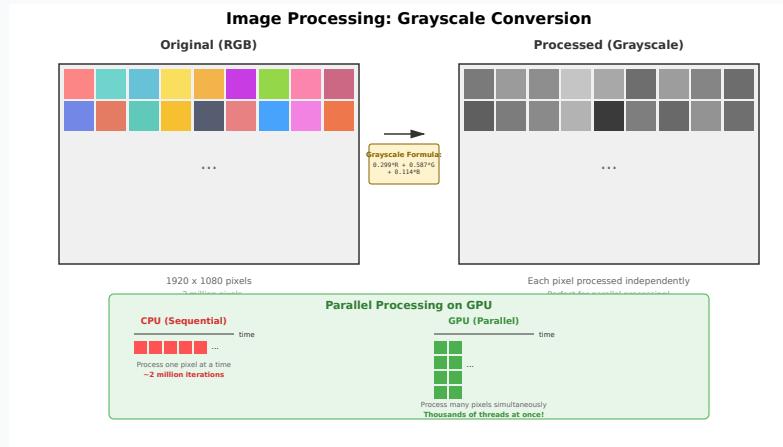
CPU vs GPU 처리

CPU: 순차 처리

```
for (int i = 0; i < height; i++) {  
    for (int j = 0; j < width; j++) {  
        int idx = i * width + j;  
        gray[idx] = 0.299*R[idx] +  
                    0.587*G[idx] +  
                    0.114*B[idx];  
    }  
}
```

시간: 2,073,600 픽셀 × 3연산 = ~50ms

병렬화 장점



GPU: 병렬 처리

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;  
if (idx < total_pixels) {  
    gray[idx] = 0.299*R[idx] +  
                0.587*G[idx] +  
                0.114*B[idx];  
}
```

시간: 모든 픽셀 동시 처리 = ~0.1ms

500배 속도 향상!

CUDA란 무엇인가? - GPU 프로그래밍의 표준

CUDA 생태계 구성요소

CUDA = Compute Unified Device Architecture

구성요소	역할	예시
언어 확장	C/C++에 GPU 기능 추가	<code>__global__</code> , <code><<<grid, block>>></code>
런타임 API	GPU 제어 및 관리	<code>cudaMalloc()</code> , <code>cudaMemcpy()</code>
라이브러리	최적화된 기능 제공	cuBLAS, cuDNN, Thrust
개발 도구	디버깅/프로파일링	nvcc, Nsight, cuda-gdb

즉시 실습: CUDA vs 다른 프레임워크

질문: 왜 CUDA가 산업 표준인가?

- OpenCL: 범용성 but 성능 (CUDA 대비 70%)
- ROCm (AMD): 호환성 (생태계 10%)
- **CUDA**: 성능 생태계 도구

결론: 90% AI/ML 프레임워크가 CUDA 우선 지원!

CUDA 프로그래밍 모델 - 병렬 컴퓨팅의 구조

헤테로지니어스 컴퓨팅

Host (CPU)와 Device (GPU)의 협력

- **Host:** 순차 처리와 제어 흐름 담당
- **Device:** 대규모 병렬 처리 담당
- 각자의 메모리 공간과 최적화된 역할

커널과 스레드 - 병렬 실행의 핵심

커널(Kernel)이란?

GPU에서 병렬로 실행되는 특수 함수

특징	설명	예시
병렬 실행	N개 스레드가 동시 실행	10,000개 픽셀 동시 처리
SIMT 모델	Single Instruction, Multiple Thread	같은 코드, 다른 데이터
스레드 ID	각 스레드가 고유 ID로 데이터 접근	<code>threadIdx.x + blockIdx.x * blockDim.x</code>

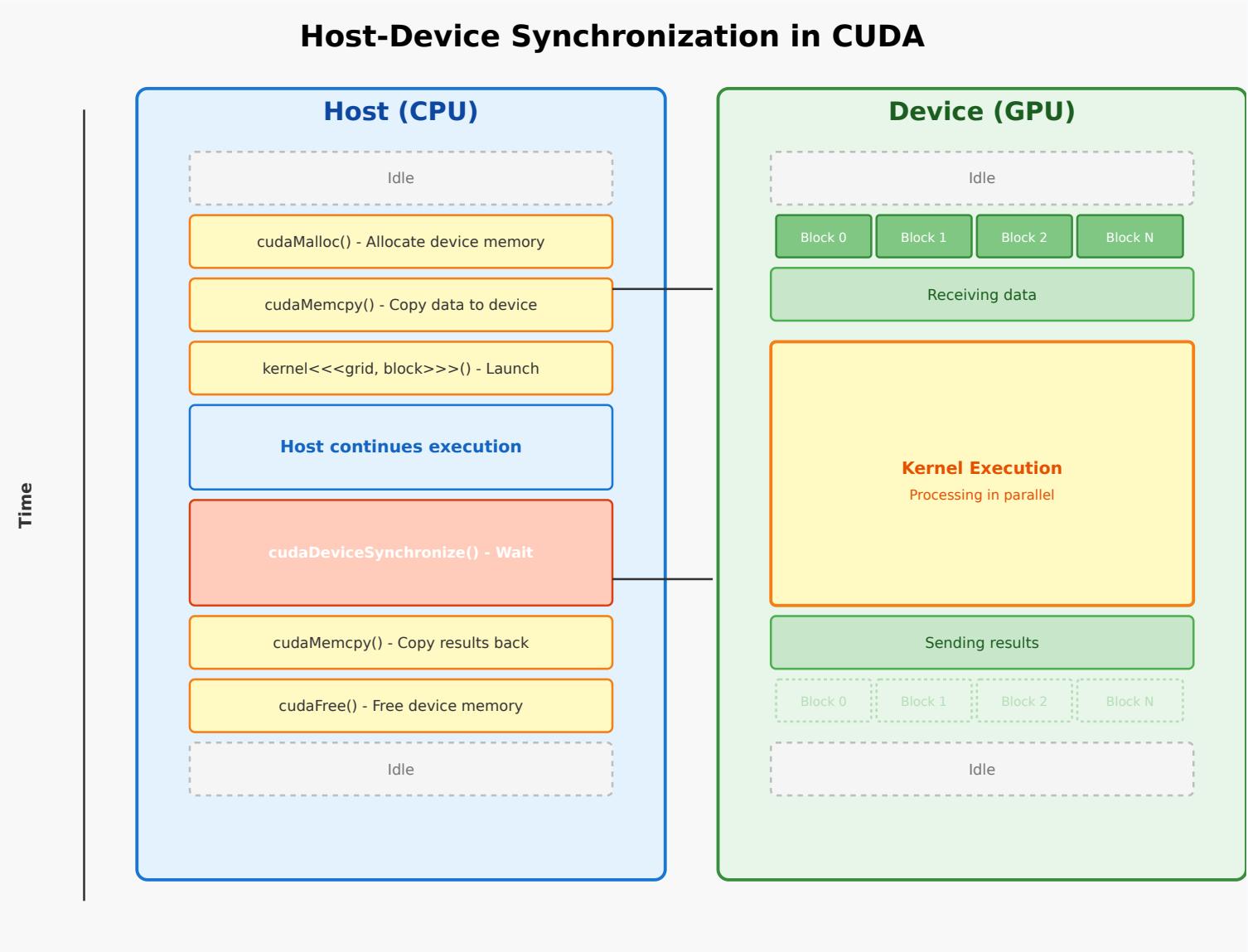
즉시 실습: 병렬화 가능성 판단

다음 중 CUDA 커널로 적합한 작업은?

1. 이미지 필터링 (각 픽셀 독립적)
2. 피보나치 수열 (이전 값 의존)
3. 벡터 덧셈 (각 요소 독립적)
4. 이진 탐색 (순차적 비교)

핵심: 데이터 독립성이 병렬화의 조건!

Host와 Device 역할 - 효율적인 협업의 핵심



CPU와 GPU의 최적 역할 분배

구분	Host (CPU)	Device (GPU)
강점	복잡한 로직, 분기 처리	대규모 병렬 연산
메모리	크고 유연한 (RAM)	빠르고 특화됨 (VRAM)
주 역할	제어, I/O, 스케줄링	대량 데이터 처리
실행 방식	순차적 + 멀티스레드	SIMT (Single Instruction Multiple Thread)

실제 역할 분담 패턴

Host의 책임

영역	구체적 역할	코드 예시
프로그램 제어	main(), 커널 호출	<code>kernel<<<grid, block>>>()</code>
메모리 관리	할당/해제	<code>cudaMalloc()</code> , <code>cudaFree()</code>
데이터 전송	H D 복사	<code>cudaMemcpy()</code>
동기화	GPU 완료 대기	<code>cudaDeviceSynchronize()</code>

Device의 책임

영역	구체적 역할	특징
병렬 연산	커널 실행	수천 개 스레드 동시
메모리 접근	Global/Shared/Local	계층적 메모리 활용
동기화	<code>__syncthreads()</code>	블록 내 스레드 동기화

실습: 역할 판단

Host vs Device 선택하기

작업	적합한 처리	이유
파일 데이터 읽기	Host	파일 I/O
100만 개 행렬 곱셈	Device	대규모 병렬
에러 처리 및 복구	Host	복잡한 로직
이미지 필터링	Device	픽셀 병렬

문제: 100만 개 데이터를 어떻게 처리?

CPU 접근: 비효율적

```
for (int i = 0; i < 1000000; i++) {  
    process(data[i]); // 1개씩 순차 처리  
}  
// 시간: 1000000 * 0.001ms = 1000ms
```

GPU의 해결책: Thread 계층 구조

```
// 1000 Blocks × 1000 Threads = 1 million concurrent processing  
kernel<<<1000, 1000>>>(data);  
// Time: 1ms (1000x faster!)
```

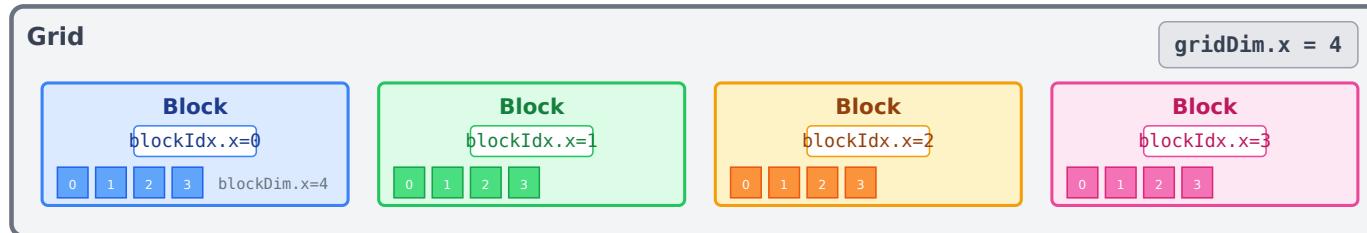
왜 Thread 계층이 필요한가?

- 하드웨어 제약: SM당 최대 2048 스레드
- 효율적 관리: 그룹 단위 스케줄링
- 메모리 공유: Block 내 협업 가능

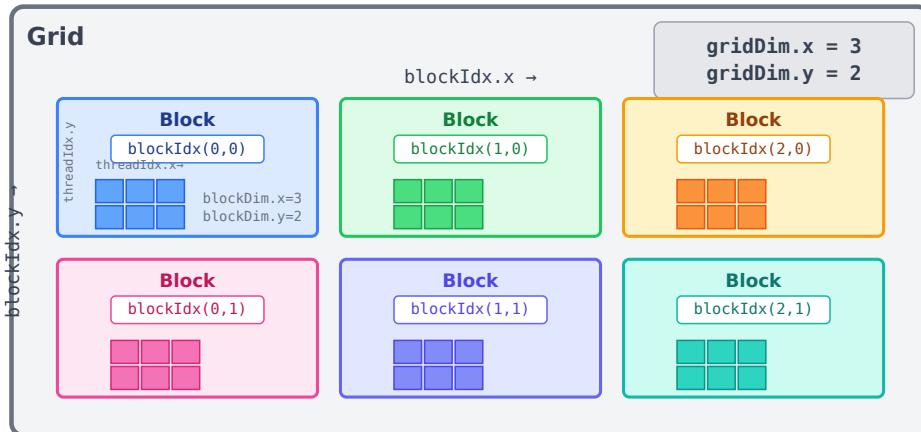
Thread 계층 구조 - CUDA의 핵심

CUDA Grid and Block Dimensions

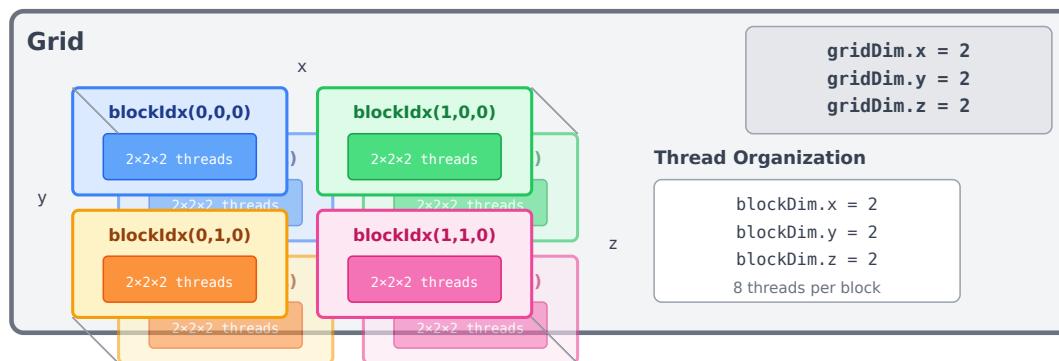
1D Configuration



2D Configuration



3D Configuration



3단계 계층 구조 (왜 필요한가?)

계층	역할	최대 크기	특징
Grid	전체 작업	$2^{31}-1$ 블록	독립적 실행
Block	협력 단위	1024 스레드	Shared Memory 공유
Thread	실행 단위	-	Warp(32개) 단위 실행

스레드 구성 예제

1920x1080 이미지 처리

```
dim3 blockSize(32, 32);      // 1024 threads/block
dim3 gridSize(60, 34);       // 60x34 = 2040 blocks
kernel<<<gridSize, blockSize>>>( . . . ); // 2,088,960 threads
```

인덱싱 공식

1D Grid

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

2D Grid

```
int x = blockIdx.x * blockDim.x + threadIdx.x;  
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

문제: GPU가 빠른데 전체 프로그램은 느리다?

실제 사례

GPU 연산: 1ms (1000회!)

데이터 이동: 50ms (20회?)

?? 데이터!!

병목 분석

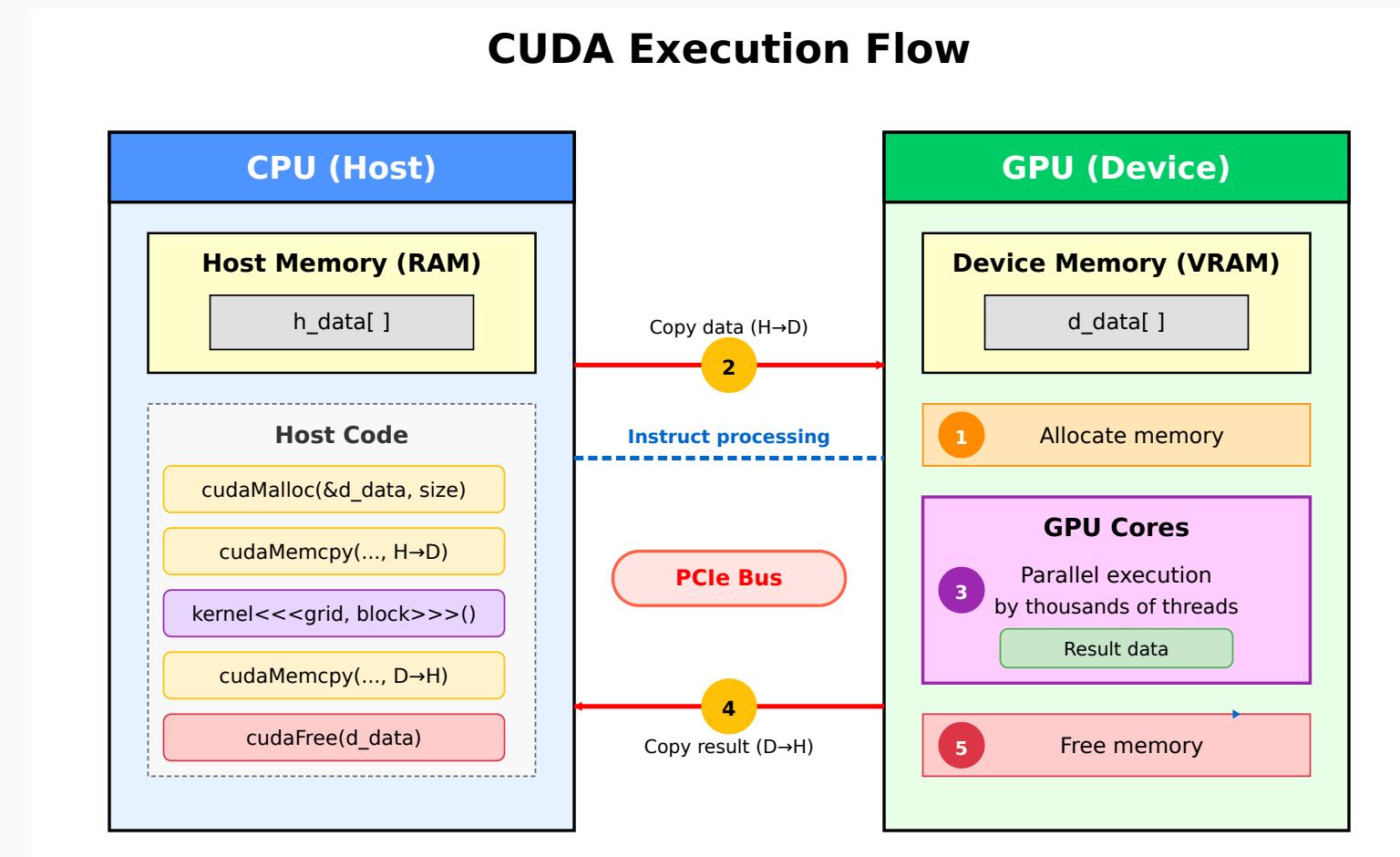
- H→D 전송: 20ms (40%)
- GPU 연산: 1ms (2%)
- D→H 전송: 20ms (40%)
- 기타: 9ms (18%)

98%의 시간이 데이터 이동!

이 장에서 배울 내용

- CUDA 프로그램의 5단계 흐름
- 각 단계의 성능 영향
- 병목 해결 방법

CUDA 프로그램 흐름 - 필수 5단계



필수 5단계와 성능 영향

단계	API	시간 비중	최적화 포인트
1. 메모리 할당	<code>cudaMalloc()</code>	~1%	Memory Pool 사용
2. H→D 전송	<code>cudaMemcpy()</code>	20-40%	Pinned Memory
3. 커널 실행	<code>kernel<<<>>()</code>	40-60%	핵심 최적화 대상
4. D→H 전송	<code>cudaMemcpy()</code>	10-20%	비동기 전송
5. 메모리 해제	<code>cudaFree()</code>	~1%	재사용 고려

실제 코드 패턴

기본 패턴

```
// 1. Allocate
float *d_data;
cudaMalloc(&d_data, size);

// 2. Copy H→D
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);

// 3. Launch
kernel<<<grid, block>>>(d_data, N);

// 4. Copy D→H
cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);

// 5. Free
cudaFree(d_data);
```

성능 병목 해결 전략

문제 진단

100MB 대비 시간:

- H→D 대비: 6.25ms
- GPU 대비: 10ms
- D→H 대비: 6.25ms
- 총 대비: 22.5ms

GPU 대비: $10/22.5 = 44\% (56\% 대비!)$

해결 방법

1. **Pinned Memory**: 전송 2배 가속
2. **CUDA Streams**: 전송/연산 동시 실행
3. **Unified Memory**: 자동 데이터 이동

결과: 22.5ms → 12ms (2배 향상)

CUDA Toolkit 설치

CUDA 개발을 시작하려면 먼저 CUDA Toolkit을 설치해야 합니다.

시스템 요구사항 확인

```
# Linux에서 GPU 확인  
lspci | grep -i nvidia
```

```
# 드라이버 버전 확인  
nvidia-smi
```

설치 과정

- NVIDIA 드라이버 설치:** GPU에 맞는 최신 드라이버 다운로드 (<https://www.nvidia.com/drivers>)
- CUDA Toolkit 다운로드:** 운영체제에 맞는 버전 선택 (<https://developer.nvidia.com/cuda-downloads>)
- 환경 변수 설정:** `PATH` 및 `LD_LIBRARY_PATH` (Linux/Mac) 또는 시스템 환경 변수 (Windows) 설정
- 설치 확인:** `nvcc --version` 및 `nvidia-smi` 명령어로 확인

Docker CUDA 개발 환경

Dockerfile

```
FROM nvidia/cuda:12.0-devel-ubuntu22.04

RUN apt-get update && \
    apt-get install -y \
    build-essential \
    cmake git vim

WORKDIR /workspace
```

실행 명령

```
# 컨테이너 실행
docker run --gpus all \
    -it -v $(pwd):/workspace \
    cuda-dev

# GPU 확인
nvidia-smi
```

장점

- 일관된 환경
- 의존성 관리 간편
- 팀 공유 용이

CUDA Development Tools

Essential Tools for GPU Programming

Core Development Tools

Tool	Purpose	Common Usage
<code>nvcc</code>	CUDA Compiler	<code>nvcc -O3 -arch=sm_86 file.cu</code>
<code>nvidia-smi</code>	GPU Monitoring	<code>nvidia-smi -l 1</code>
<code>cuda-gdb</code>	CUDA Debugger	<code>cuda-gdb ./program</code>
<code>nvprof</code>	Legacy Profiler	<code>nvprof ./program</code>

Build Flags

```
nvcc -g -G          # Debug build  
nvcc -O3 -use_fast_math # Release build  
nvcc -arch=native    # Auto-detect GPU
```

Nsight Tool Suite

Professional Grade Analysis

System-Level Analysis

Nsight Systems - Timeline profiling

```
nsys profile -o report ./program  
nsys stats report.qdrep
```

- CPU-GPU interaction
- Memory transfers
- Kernel launches
- API calls

Kernel-Level Analysis

Nsight Compute - Detailed kernel metrics

```
ncu -o report ./program  
ncu --set full --target-processes all ./program
```

- Occupancy analysis
- Memory throughput
- Instruction mix
- Register usage

Development Workflow

1. Build and Test

```
# Compile with debugging  
nvcc -g -G -o program program.cu
```

```
# Run with error checking
```

2. Profile and Analyze

```
# Quick profile  
nvprof --print-gpu-trace ./program
```

```
# Detailed analysis
```

3. Optimize

```
# Compiler reports  
nvcc --ptxas-options=-v program.cu
```

```
# Occupancy calculator
```

Best Practices

- Always use `-arch` flag for target GPU
- Enable all warnings: `-Wall -Wextra`
- Use debug builds during development

Hello CUDA - 첫 번째 CUDA 프로그램

최소한의 CUDA 프로그램 구조

```
// GPU 코드 - GPU에서 실행되는 코드
__global__ void hello() {
    printf("Hello from GPU Thread %d, Block %d\n",
        threadIdx.x, blockIdx.x);
}

int main() {
    // Launch kernel: 2 blocks, 4 threads per block
    hello<<<2, 4>>>();

    // GPU 코드 종료
    cudaDeviceSynchronize();
    return 0;
}
```

Hello CUDA 실행 흐름

핵심 개념

- `__global__` : GPU에서 실행되는 커널 함수
- `<<<grid, block>>>` : 실행 구성 (grid=블록 수, block=블록당 스레드 수)
- `cudaDeviceSynchronize()` : GPU 작업 완료 대기

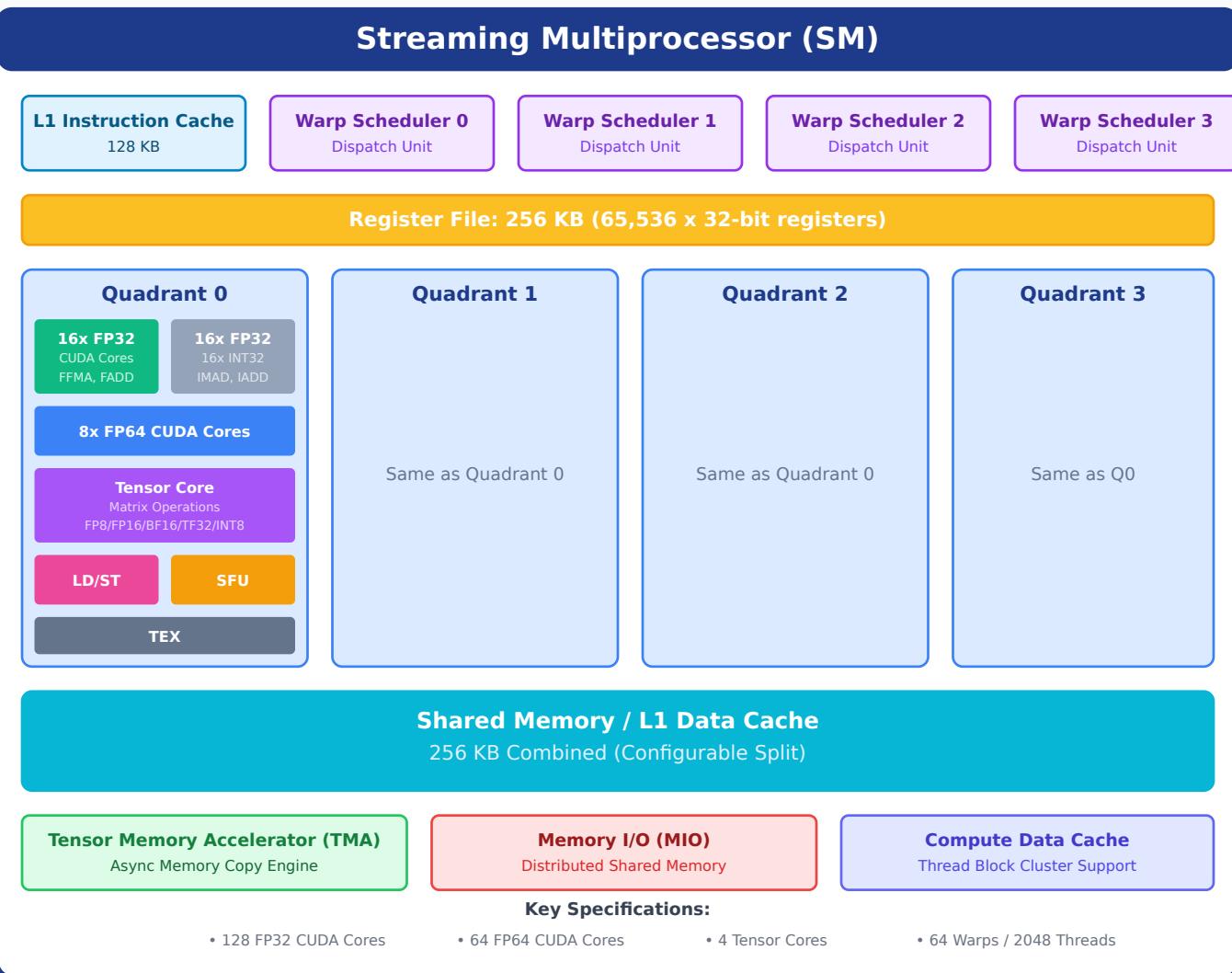
실행 과정

1. **커널 Launch**: Host가 GPU에 작업 요청
2. **GPU 실행**: 8개 스레드가 병렬로 실행
 - Block 0: Thread 0, 1, 2, 3
 - Block 1: Thread 0, 1, 2, 3
3. **동기화**: Host가 GPU 완료 대기

주의: 스레드 실행 순서는 보장되지 않음!

GPU 아키텍처: Streaming Multiprocessor

NVIDIA Hopper SM Architecture



GPU 정보 조회 API

```
cudaGetDeviceCount(&deviceCount);  
cudaGetDeviceProperties(&deviceProp, device);
```

GPU 디바이스 속성

주요 속성 필드

- **name:** GPU 모델명
- **major/minor:** Compute Capability
- **totalGlobalMem:** 전체 메모리
- **multiProcessorCount:** SM 개수

예시 출력

```
Device 0: RTX 3090
  Compute capability: 8.6
  Total memory: 24.00 GB
  Multiprocessors: 82
  CUDA cores: 10496
  Clock rate: 1.70 GHz
  Memory bandwidth: 936 GB/s
```

실행 방법

```
nvcc device_info.cu -o device_info
./device_info
```

Problem: GPU Only Using 10% Capacity!

Real Example

```
// Developer's mistake  
kernel<<<1, 256>>>(data); // Only 1 Block, 256 threads  
  
// GTX 1070 specs  
// Total 15 SMs (Streaming Multiprocessor)  
// 14 SMs are idle! (93% idle)  
// Performance: 15x slower
```

Why Does This Happen?

Not understanding SM (Streaming Multiprocessor)!

What You'll Learn

- Determine Grid size for 100% GPU utilization
- Hardware-specific optimal settings
- 10-15x performance improvement

SM (Streaming Multiprocessor) Understanding

Where Blocks Actually Execute!

Core Concepts

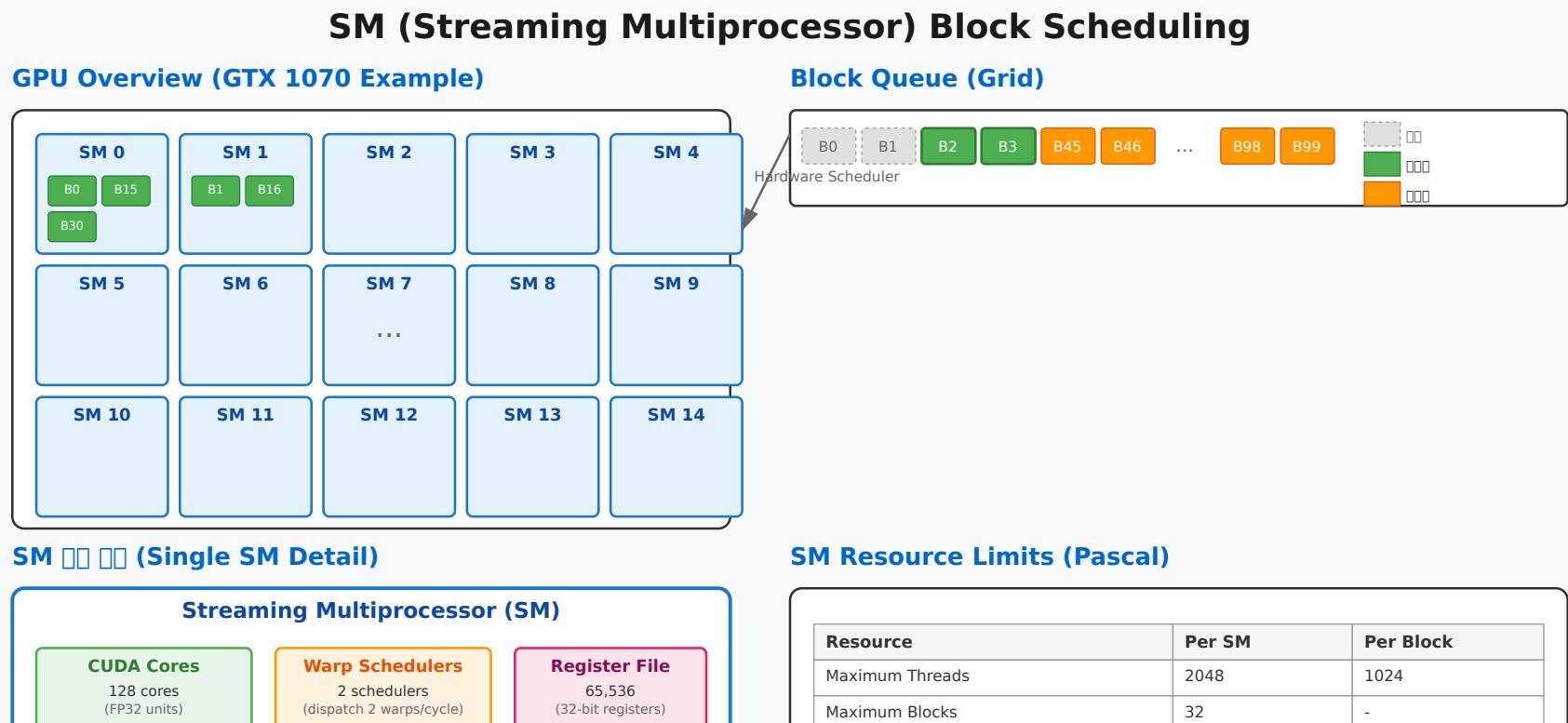
GPU = Multiple SMs

SM = Physical processor that executes Blocks

GTX 1070: 15 SMs

Block → SM Assignment Rules

1. Each Block runs on only one SM
2. One SM can run multiple Blocks concurrently



cudaDeviceProp and Hardware Mapping

Reading and Using GPU Information

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);

// Key hardware information
printf("SM count: %d\n", prop.multiProcessorCount);          // 15 (GTX 1070)
printf("Max threads per SM: %d\n", prop.maxThreadsPerMultiProcessor); // 2048
printf("Max blocks per SM: %d\n", prop.maxBlocksPerMultiProcessor);    // 32
printf("Max threads per block: %d\n", prop.maxThreadsPerBlock);        // 1024
printf("Warp size: %d\n", prop.warpSize);                      // 32
printf("Registers per SM: %d\n", prop.regsPerMultiprocessor);      // 65536
printf("Shared memory per block: %d\n", prop.sharedMemPerBlock);    // 49152
```

What to Decide with This Information?

```
// Determine optimal block size
int blockSize = 256; // Must be <= prop.maxThreadsPerBlock

// Determine optimal grid size
int gridSize = prop.multiProcessorCount * 2; // 2 blocks per SM
// Example: GTX 1070 = 15 SM * 2 = 30 blocks (details in Chapter 21)
```

SM 리소스 제한

GTX 1070 (Pascal) 예시

Resource	Per SM	Per Block
Max Threads	2048	1024
Max Blocks	32	-
Registers	65536	65536
Shared Memory	96 KB	48 KB
Max Warps	64	32

리소스 제한 예시

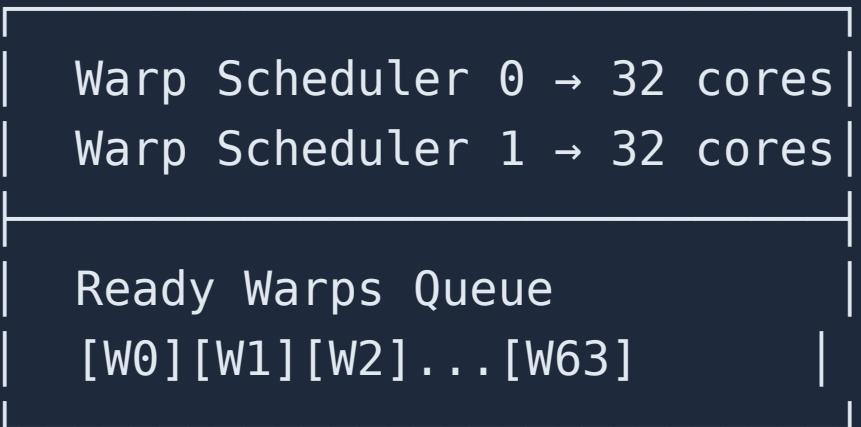
```
// Register limited
__global__ void kernel() {
    float regs[64]; // 64 registers/thread
    // 65536 / 64 = 1024 threads max per SM
}
```

핵심: 레지스터 사용량이 SM당 스레드 수를 제한

Warp Scheduling

Warp 실행 메커니즘

SM with 2 Warp Schedulers (Pascal)



Warp 상태

- **Eligible:** 실행 준비 완료
- **Stalled:** 메모리 대기, 동기화 대기
- **Active:** 현재 실행 중

Occupancy 계산

Occupancy = Active Warps / Max Warps

```
// GTX 1070: Max 64 warps per SM

// Case 1: Small blocks
kernel<<<15, 128>>>(); // 128 threads = 4 warps
// Per SM: 16 blocks × 4 warps = 64 warps
// Occupancy: 64/64 = 100%

// Case 2: Large blocks
kernel<<<15, 1024>>>(); // 1024 threads = 32 warps
// Per SM: 2 blocks × 32 warps = 64 warps
// Occupancy: 64/64 = 100%

// Case 3: Resource limited (48KB shared memory)
__shared__ float data[12288]; // 48KB
// Per SM: 2 blocks only
// If blockSize=256: 2 × 8 = 16 warps
// Occupancy: 16/64 = 25%
```

Block 크기 최적화

고려 요소

1. Warp의 배수

```
// Good: Multiple of 32
dim3 block(256); // 8 warps
dim3 block(128); // 4 warps

// Bad: Not multiple of 32
dim3 block(100); // 3.125 warps (waste!)
```

리소스 균형과 GPU별 최적값

리소스 균형

- Register 사용량 체크
- Shared memory 사용량 체크
- Block당 최소 4 warps 권장

GPU별 최적값

GPU	Block Size
GTX 1070	128-256
RTX 3090	256-512

Wave와 Tail Effect

Wave 개념

```
// GTX 1070: 15 SMs  
kernel<<<150, 256>>>();  
  
// Wave 1: Block 0-14  (all 15 SMs busy)  
// Wave 2: Block 15-29  (all 15 SMs busy)  
// ...  
// Wave 10: Block 135-149 (all 15 SMs busy)
```

Tail Effect 문제

```
kernel<<<151, 256>>>();  
  
// Wave 1-10: 150 blocks (150 150)  
// Wave 11: 1 block only (14 SMs idle!)  
// 总计: 151/165 = 91.5%
```

Tail Effect 해결책

- Grid size = SM 개수의 배수 사용
- Dynamic parallelism 활용
- Grid-stride loop 패턴 적용

실전 최적화 전략

Occupancy Calculator

```
nvcc --ptxas-options=-v kernel.cu  
# Used 32 registers, 4096 bytes smem
```

동적 최적화 기법

동적 Block 크기

```
int blockSize, minGridSize;
cudaOccupancyMaxPotentialBlockSize(
    &minGridSize, &blockSize, kernel, 0, 0);
kernel<<<minGridSize, blockSize>>>();
```

Launch Bounds

```
__global__ __launch_bounds__(256, 8)
void kernel() {
    // 256 threads/block, 8 blocks/SM
}
```

SM 성능 모니터링

Nsight Compute 메트릭

SM Efficiency: 85%

Achieved Occupancy: 72%

Warp Execution Efficiency: 93%

병목 지점 분석

문제	해결 방법
Low Occupancy	Block 크기 조정
Register Spilling	Register 사용 줄이기
Bank Conflict	Padding 추가
Warp Divergence	분기 최적화

최종 체크리스트

성능 확인 포인트

✓ Block size는 warp(32)의 배수 ✓ Occupancy 50% 이상 유지 ✓ 모든 SM 활용 확인 ✓ Resource 사용 균형 확인

권장 설정

- **Block Size:** 128, 256, 512
- **Grid Size:** SM 개수 × N (N=2,4,8...)
- **Shared Memory:** 48KB 이하
- **Registers:** 64개 이하/thread

SM, Block, Thread 관계 정리

계층 구조

GPU → SM → Block → Warp → Thread

핵심 제한 사항 (GTX 1070 기준)

구분	최대값	설명
Block당 최대 Thread	1024	하드웨어 한계
SM당 최대 Thread	2048	동시 실행 가능한 스레드
SM당 최대 Block	32	동시 상주 가능한 블록
Warp 크기	32	변경 불가, 모든 GPU 동일

왜 이런 제한이 있는가?

Block당 최대 1024 Thread

- 이유: 하드웨어 레지스터/리소스 제한
- 의미: 한 Block은 최대 1024개 스레드만 가능
- 예시: dim3 block(1024)
✓
/ dim3 block(2048)
✗

SM당 최대 2048 Thread

- 이유: SM의 물리적 실행 유닛 한계
- 의미: 한 SM에서 동시에 실행 가능한 총 스레드 수
- 예시:
 - 2 blocks × 1024 threads = 2048
✓
 - 4 blocks × 512 threads = 2048
✓
 - 3 blocks × 1024 threads = 3072
✗
(초과)

SM당 최대 32 Block

왜 32개인가?

- 하드웨어 스케줄러 한계: Block 관리 유닛 제한
- 메모리 관리: Block별 컨텍스트 저장 공간

실제 제한 예시

```
// Case 1: Thread 수 2048  
blockSize = 1024 → SM 2 blocks로 2048/1024 = 2  
  
// Case 2: Block 수 2048  
blockSize = 64 → SM 32 blocks로 2048/64 = 32  
// (2048/64 = 32로 나뉨)  
  
// Case 3: Block 수 2048  
blockSize = 32 → SM 32 blocks로 2048/32 = 64 blocks로 32로 나뉨
```

GPU 세대별 차이

GPU	Compute Capability	Block당 최대 Thread	SM당 최대 Thread	SM당 최대 Block
GTX 1070	6.1	1024	2048	32
RTX 3090	8.6	1024	1536	16
A100	8.0	1024	2048	32
H100	9.0	1024	2048	32

주의: 최신 GPU가 항상 더 많은 것은 아님!

실전 최적 설정 가이드

1. Block 크기 선택

```
// 높이: 256 × 512  
// - Warp(32) × ×  
// - 블록 × × ×  
// - 블록 × ×  
  
dim3 block(256); // 높이 × × ×
```

2. Grid 크기 계산

```
cudaDeviceProp prop;  
cudaGetDeviceProperties(&prop, 0);  
  
// SM count × blocks per SM  
int gridSize = prop.multiProcessorCount * 4; // 4 blocks per SM
```

Occupancy 계산 예시

Occupancy = 활성 Warp / 최대 Warp

GTx 1070 예시

- 최대 Warp/SM: 64 (2048 threads / 32)
- Block 크기: 256 threads = 8 warps

```
SM[] Block [] = min(  
    2048 / 256, // Thread #: 8  
    32           // Block #: 32  
) = 8 blocks
```

Warp = 8 blocks × 8 warps = 64
Occupancy = 64 / 64 = 100%

자주 하는 질문

Q: Block 크기를 크게 하면 빠른가?

A: 아님. Occupancy와 리소스 사용량 균형이 중요

Q: SM이 많으면 무조건 좋은가?

A: 병렬화 가능한 작업량이 충분해야 함

Q: Thread를 최대한 많이 만들어야 하나?

A: 데이터 크기만큼만. Grid-Stride Loop 활용

Q: 왜 32의 배수가 중요한가?

A: Warp 단위(32) 실행. 남는 스레드는 낭비

Occupancy Optimization

Maximizing GPU Utilization

What is Occupancy?

Occupancy = Active Warps / Maximum Warps

Why It Matters

- Higher occupancy → Better latency hiding
- More warps → More parallelism
- Better resource utilization

Occupancy Calculation Examples

GTX 1070: Max 64 warps per SM

Case 1: Small Blocks (High Occupancy)

```
kernel<<<15, 128>>>(); // 4 warps per block  
// Per SM: 16 blocks × 4 warps = 64 warps  
// Occupancy: 64/64 = 100%
```

Case 2: Large Blocks (Still Good)

```
kernel<<<15, 1024>>>(); // 32 warps per block  
// Per SM: 2 blocks × 32 warps = 64 warps  
// Occupancy: 64/64 = 100%
```

Case 3: Resource Limited (Poor)

```
__shared__ float data[12288]; // 48KB shared memory  
// Per SM: Only 2 blocks possible  
// If blockSize=256: 2 × 8 = 16 warps  
// Occupancy: 16/64 = 25%
```

Occupancy Optimization Strategies

1. Dynamic Block Size Selection

```
int blockSize, minGridSize;
cudaOccupancyMaxPotentialBlockSize(
    &minGridSize, &blockSize, kernel, 0, 0);
kernel(minGridSize, blockSize);
```

2. Launch Bounds Directive

```
__global__ __launch_bounds__(256, 8)
void kernel() {
    // Hint: 256 threads/block, 8 blocks/SM
}
```

3. Register Pressure Reduction

```
// Check register usage
nvcc --ptxas-options=-v kernel.cu
// Output: Used 32 registers

// Reduce if needed
```

Target Occupancy

- **Minimum:** 50% for latency hiding
- **Sweet spot:** 66-75%

Wave and Tail Effect

Understanding Block Distribution Patterns

Wave Concept

```
// GTX 1070: 15 SMs
kernel<<<150, 256>>>();

// Wave 1: Block 0-14  (all 15 SMs busy)
// Wave 2: Block 15-29  (all 15 SMs busy)
// ...
// Wave 10: Block 135-149 (all 15 SMs busy)
```

Execution Timeline

Time →

Wave 1: [B0][B1][B2]...[B14] All SMs active

Wave 2: [B15][B16]...[B29] All SMs active

Wave 3: [B30][B31]...[B44] All SMs active

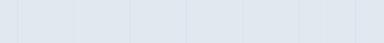
Tail Effect Problem

Inefficient Last Wave

```
kernel<<<151, 256>>>(); // 151 blocks on 15 SMs  
  
// Wave 1-10: 150 blocks (even distribution)  
// Wave 11: 1 block only (14 SMs idle!)  
// Efficiency: 151/165 = 91.5%
```

Visual Example

SM Usage Pattern:

Wave 1-10:  (100% utilization)

Wave 11:  (6.7% utilization)
└ 14 SMs wasted!

Performance Impact

- Last wave can waste up to $(SM_count - 1)$ SMs
- Overall efficiency drops significantly
- GPU resources underutilized

Solving Tail Effect

Strategy 1: Multiple of SM Count

```
int numSMs = prop.multiProcessorCount; // 15 for GTX 1070
int blocksPerSM = 4;
int gridSize = numSMs * blocksPerSM; // 60 blocks
```

Strategy 2: Grid-Stride Loop

```
__global__ void kernel(float* data, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    // Process multiple elements per thread
    for(int i = idx; i < N; i += stride) {
        process(data[i]);
```

Strategy 3: Dynamic Work Distribution

```
// Calculate optimal grid size
int elementsPerThread = 4;
int totalThreads = (N + elementsPerThread - 1) / elementsPerThread;
int gridSize = (totalThreads + blockSize - 1) / blockSize;

// Round to SM multiple
```

Wave Optimization Examples

Poor Distribution (71% efficiency)

```
// 100 blocks on 15 SMs
kernel<<<100, 256>>>();
// Wave 1-6: 90 blocks (15x6)
// Wave 7: 10 blocks (5 SMs idle)
```

Good Distribution (100% efficiency)

```
// 120 blocks on 15 SMs
kernel<<<120, 256>>>();
// Wave 1-8: 120 blocks (15x8)
// All waves fully utilize SMs
```

Best Practice Formula

```
int waves = 4; // Target number of waves
int gridSize = numSMs * waves;
kernel<<<gridSize, blockSize>>>();
```

Performance Guidelines

- Minimum 2-4 waves for latency hiding
- Balance between waves and block size
- Consider dynamic parallelism for irregular workloads

SM Resource Limits

Hardware Constraints on Parallelism

GTX 1070 (Pascal) Example

Resource	Per SM	Per Block	Impact
Max Threads	2048	1024	Occupancy ceiling
Max Blocks	32	-	Small block advantage
Registers	65536	65536	Spilling risk
Shared Memory	96 KB	48 KB	Block limit
Max Warps	64	32	Scheduling limit

Resource Competition

Multiple blocks share SM resources
Block 1: Uses 16KB shared, 8192 registers
Block 2: Uses 16KB shared, 8192 registers
→ SM can fit 6 blocks (96KB/16KB)

Register Pressure

Register Limitation Example

```
__global__ void heavy_kernel() {  
    float regs[64]; // 64 registers per thread  
    // 65536 registers / 64 = 1024 threads max per SM  
  
    // With 256 threads/block:  
    // 256 × 64 = 16384 registers per block  
    // 65536 / 16384 = 4 blocks max per SM  
}
```

Checking Register Usage

```
nvcc --ptxas-options=-v kernel.cu  
# Output: Used 32 registers, 0 bytes cmem[0]  
  
# Calculate occupancy impact:  
# 65536 / 32 = 2048 threads (full occupancy)  
# 65536 / 64 = 1024 threads (50% occupancy)
```

Shared Memory Constraints

Memory Allocation Impact

```
__global__ void shared_heavy() {
    __shared__ float tile[64][64]; // 16KB
    // 96KB / 16KB = 6 blocks max per SM

    // If block has 256 threads:
    // 6 blocks × 256 = 1536 threads
    // 1536 / 2048 = 75% occupancy
```

Dynamic Shared Memory

```
extern __shared__ float dynamic[];

// Launch with dynamic size
int sharedBytes = blockSize * sizeof(float) * 4;
kernel(<gridSize, blockSize, sharedBytes>);
```

Memory Configuration

```
// Configure L1/Shared split (Volta+)
cudaFuncSetAttribute(kernel,
    cudaFuncAttributeMaxDynamicSharedMemorySize, 96*1024);
cudaFuncSetAttribute(kernel,
    cudaFuncAttributePreferredSharedMemoryCarveout,
```

Resource Limit Calculations

Finding the Bottleneck

```
// Given kernel requirements:  
// - 48 registers per thread  
// - 8KB shared memory per block  
// - 256 threads per block  
  
// Calculate limits:  
// 1. Thread limit: 2048 / 256 = 8 blocks  
// 2. Register limit: 65536 / (256×48) = 5.3 → 5 blocks  
// 3. Shared mem limit: 96KB / 8KB = 12 blocks  
// 4. Block limit: 32 blocks  
  
// Bottleneck: Registers (5 blocks max)  
// Occupancy: (5×256) / 2048 = 62.5%
```

Optimization Strategies

Bottleneck	Solution
Registers	Reduce variables, use shared memory
Shared Memory	Reduce tile size, use texture memory
Threads	Adjust block size
Blocks	Increase threads per block

Resource Optimization Example

Before Optimization

```
__global__ void unoptimized() {
    float a[10], b[10], c[10]; // 30 registers
    __shared__ float shared[1024]; // 4KB

    // Heavy computation
    for(int i = 0; i < 10; i++) {
        c[i] = a[i] * b[i];
    }
}
```

After Optimization

```
__global__ void optimized() {
    float a, b, c; // 3 registers (reuse)
    __shared__ float shared[256]; // 1KB

    // Loop with register reuse
    for(int i = 0; i < 10; i++) {
        a = load(i);
        b = load(i);
        c = a * b;
        store(c, i);
    }
}
```

Result: 1.8x performance improvement

SM Performance Monitoring

Profiling and Analysis Tools

Key Metrics to Monitor

```
SM Efficiency: 85%          // How well SMs are utilized
Achieved Occupancy: 72%      // Active vs max warps
Warp Execution Efficiency: 93% // Non-divergent execution
Memory Throughput: 450 GB/s // Bandwidth utilization
```

Nsight Compute Analysis

```
# Detailed kernel profiling
ncu --set full ./program

# Key metrics only
ncu --metrics sm__throughput.avg,\
sm__warps_active.avg.pct_of_peak_sustained_active ./program
```

Performance Bottleneck Analysis

Common Bottlenecks and Solutions

Problem	Symptoms	Solution
Low Occupancy	< 50% active warps	Adjust block size
Register Spilling	Local memory usage	Reduce register count
Bank Conflicts	Shared memory stalls	Add padding
Warp Divergence	Low efficiency	Optimize branches
Memory Bottleneck	Low SM utilization	Improve coalescing

Diagnostic Commands

```
# Check register usage
nvcc --ptxas-options=-v kernel.cu

# Profile memory access
ncu --metrics l1tex_t_sectors_pipe_lsu_mem_global_op_ld.sum \
    ./program

# Analyze occupancy
ncu --metrics sm_warps_active.avg.pct_of_peak_sustained_active \
    ./program
```

Profiling Workflow

Step 1: Initial Profile

```
# Quick overview  
nsys profile -o report ./program  
nsys stats report.qdrep  
  
# Sample output:  
# CUDA Kernel Statistics:  
# vecAdd: 45% time, 85% occupancy  
# matMul: 35% time, 62% occupancy ← Focus here  
# reduce: 20% time, 71% occupancy
```

Step 2: Deep Dive

```
# Analyze problematic kernel  
ncu -k matMul --set full ./program  
  
# Key findings:  
# - Register usage: 64 (limiting factor)  
# - Shared memory: 16KB (sufficient)  
# - Occupancy: 62% (below target)
```

Performance Metrics Interpretation

Occupancy vs Performance

Occupancy	Performance	Analysis
25%	Poor	Increase blocks/threads
50%	Good	Minimum for latency hiding
75%	Optimal	Sweet spot
100%	Variable	Not always fastest

Warp Efficiency Metrics

```
// Good: Coalesced access
```

```
float val = data[blockIdx.x * blockDim.x + threadIdx.x];
```

```
// Warp Efficiency: 95-100%
```

```
// Bad: Strided access
```

```
float val = data[threadIdx.x * gridDim.x + blockIdx.x];
```

```
// Warp Efficiency: 12-25%
```

Advanced Monitoring Techniques

Custom Performance Counters

```
// Manual timing
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

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

cudaEventSynchronize(stop);
```

Occupancy API

```
int numBlocks;
int blockSize = 256;

// Get occupancy-based launch config
cudaOccupancyMaxActiveBlocksPerMultiprocessor(
    &numBlocks, kernel, blockSize, 0);

printf("Max active blocks: %d\n", numBlocks);
printf("Occupancy: %.2f%%\n");
```

Performance Optimization Checklist

Pre-Launch Verification

- ✓ Block size is multiple of 32 (warp size)
- ✓ Grid size utilizes all SMs
- ✓ Shared memory < 48KB per block
- ✓ Register usage allows target occupancy

Runtime Monitoring

- ✓ Occupancy > 50%
- ✓ No register spilling to local memory
- ✓ Warp efficiency > 80%
- ✓ Memory throughput > 70% of peak

Post-Execution Analysis

- ✓ All SMs were active
- ✓ No serialization bottlenecks
- ✓ Kernel time matches expectations
- ✓ No unnecessary synchronization

Target Metrics

- **Minimum:** 50% occupancy, 70% efficiency
- **Good:** 66% occupancy, 85% efficiency
- **Excellent:** 75% occupancy, 95% efficiency

Thread ID 계산

데이터 맵핑의 핵심

왜 Thread ID가 중요한가?

각 스레드는 자신만의 데이터를 처리해야 함!

```
__global__ void processArray(float* data, int N) {
    // Calculate my thread ID
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Check if I have data to process
    if (tid < N) {
        data[tid] = data[tid] * 2.0f; // Process only my data
    }
}
```

Thread ID 계산 연습

1D Grid 예제

Block ID	Thread ID	Global Thread ID	계산식
0	0	0	$0 \times 256 + 0$
0	255	255	$0 \times 256 + 255$
1	0	256	$1 \times 256 + 0$
2	100	612	$2 \times 256 + 100$

공식

```
int globalThreadId = blockIdx.x * blockDim.x + threadIdx.x;
```

2D/3D Thread ID 계산

2D Grid 예제

```
__global__ void process2D(float* data, int width, int height) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < width && row < height) {
        int idx = row * width + col; // Row-major indexing
        data[idx] = data[idx] * 2.0f;
    }
}
```

3D Grid Thread ID

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;
int idx = z * (width * height) + y * width + x;
```

스레드 ID 계산 공식

다차원 인덱싱에서 글로벌 스레드 ID를 계산하는 것은 매우 중요합니다.

1D 인덱싱

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

2D 인덱싱

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int idx = y * width + x;
```

3D 인덱싱

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int z = threadIdx.z + blockIdx.z * blockDim.z;
int idx = z * width * height + y * width + x;
```

중요: 올바른 인덱스 계산은 메모리 접근 패턴과 성능에 직접적인 영향을 미칩니다.

문제: 빅데이터 연산이 너무 느리다!

실제 사례: 딥러닝 행렬 연산

```
# 10 million vector additions
for i in range(10000000):
    c[i] = a[i] + b[i]
# CPU time: 25ms
# Deep learning 1 epoch = 1000 iterations = 25 seconds
# 100 epochs = 42 minutes!
```

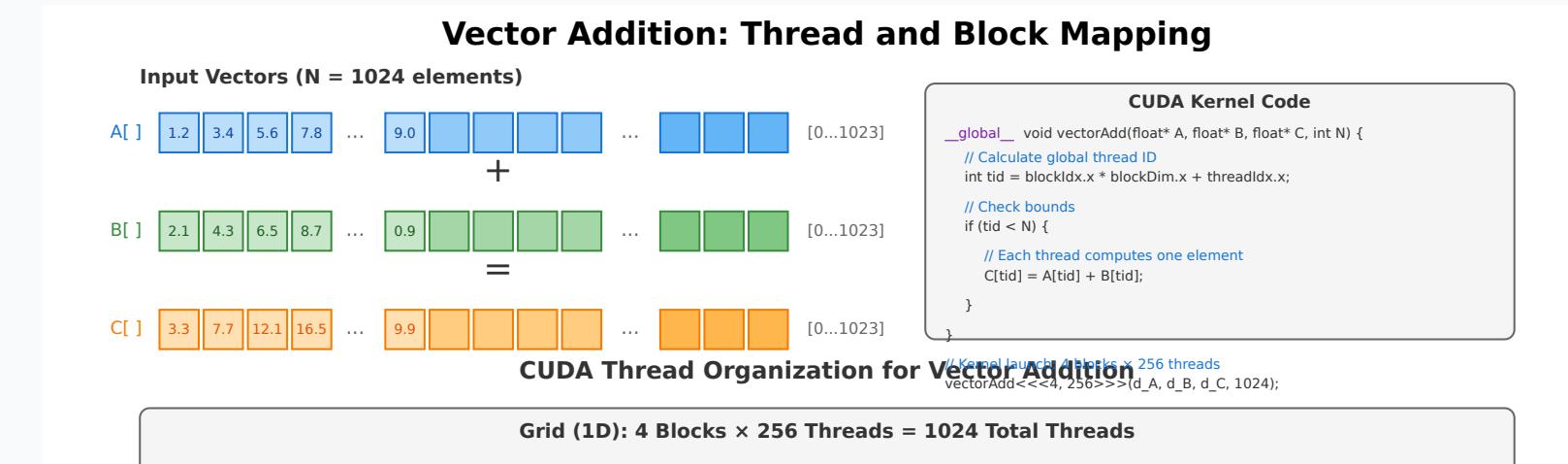
왜 CPU가 비효율적인가?

- 순차 처리: 1번에 1개씩
- 독립적 연산: 각 덧셈은 완전 독립
- 병렬화 가능: 모든 덧셈 동시 가능

CUDA의 해결책

```
// 10000 threads running concurrently
kernel<<<10000, 1000>>>(a, b, c);
// GPU time: 0.12ms (210x faster!)
```

벡터 덧셈 - 병렬화의 기본



CPII vs GPII 구현 비교

```
// CPU: 순차적 처리  
for (int i = 0; i < n; i++) {  
    c[i] = a[i] + b[i];  
}
```

선느 비교 (1처마 개 워수)

방법	시간	상대 속도	이유
CPU (1 core)	25.3ms	1x	순차 처리
CPU (8 cores)	3.2ms	8x	제한된 병렬성
GPU	0.12ms	210x	대규모 병렬

해설 차이점

- CPU: for loop로 순차 처리

벡터 덧셈 CUDA 구현

GPU 커널 함수

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

메모리 관리와 실행

함수	역할
cudaMalloc()	GPU 메모리 할당
cudaMemcpy()	CPU ↔ GPU 데이터 전송
cudaFree()	GPU 메모리 해제

```
vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, N);
```

CUDA 성능 측정 방법

GPU 시간 측정 (정확한 방법)

```
// CUDA Event로 정밀한 시간 측정
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// 시작
cudaEventRecord(start);

// 실행
myKernel<<<grid, block>>>(data, N);

// 종료
cudaEventRecord(stop);
cudaEventSynchronize(stop);

// 결과 출력 (밀리초)
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

CPU vs GPU 타이밍 비교

잘못된 방법 - CPU Timer

```
auto start = std::chrono::high_resolution_clock::now();

kernel<<<grid, block>>>();
// 문제: 비동기 실행이라 부정확!

auto end = std::chrono::high_resolution_clock::now();
// 실제 커널은 아직 실행 중일 수 있음
```

올바른 방법 - CUDA Event

```
cudaEventRecord(start);
kernel<<<grid, block>>>();
cudaEventRecord(stop);
cudaEventSynchronize(stop); // 대기

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

성능 메트릭 계산

주요 메트릭

```
// 1. Bandwidth (Bandwidth)
float bandwidth_GBs = (bytes_transferred / 1e9) / (time_ms / 1000);

// 2. Throughput (Throughput)
float throughput = elements_processed / (time_ms / 1000);

// 3. GFLOPS (GFLOPS)
float gflops = (floating_operations / 1e9) / (time_ms / 1000);
```

실제 예제

```
// Vector addition: N elements
size_t bytes = N * sizeof(float) * 2; // read + write
float bandwidth = bytes / (milliseconds * 1e6); // GB/s

// Matrix multiply: 2*M*N*K operations
size_t flops = 2 * M * N * K;
float gflops = flops / (milliseconds * 1e6); // GFLOPS
```

Grid-Stride Loop - 대용량 데이터 처리

문제: 데이터가 스레드보다 많을 때

상황 예시

```
// 100 million data, but...
// GTX 1070: 15 SMs × 2048 threads/SM = 30,720 max threads
// ㄷㄷㄷ ㄷㄷㄷ?
```

해결책: Grid-Stride Loop

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

    // Total thread count (stride)
    int gridStride = gridDim.x * blockDim.x;

    // Each thread processes multiple elements
    for (int i = idx; i < n; i += gridStride) {
        data[i] = data[i] * 2.0f;
    }
}
```

Grid-Stride 동작 원리

구체적인 예시 (GTX 1070)

```
// GTX 1070 hardware specs (from Chapter 16)
// - SM count: 15
// - Max blocks per SM: 32
// - Max threads per SM: 2048

// Why 30 blocks?
// 30 blocks = 15 SMs × 2 blocks/SM
// → All SMs process exactly 2 blocks each!

processArray<<<30, 256>>>(data, 100000000);
// Total threads: 30 × 256 = 7,680
```

Block이 SM에 분배되는 과정

- ① 1: Block 0-14 → SM 0-14 (SM 1)
- ② 2: Block 15-29 → SM 0-14 (SM 1)
- ⋮: ⋮ SM ⋮ ⋮ 2 Block ⋮ (⋮ ⋮ ⋮)

왜 30이 좋고 31은 나쁜가?

SM 개수와 Wave 실행



Good: 30 blocks (15 × 2)

```
Wave 1: Block 0-14 → 15 SM 00 00 (100%)
Wave 2: Block 15-29 → 15 SM 00 00 (100%)
```



Bad: 31 blocks

```
Wave 1: Block 0-14 → 15 SM 00 00 (100%)
Wave 2: Block 15-29 → 15 SM 00 00 (100%)
Wave 3: Block 30 → 1 SM 00 00 (6.7%) 0000
```

핵심 공식 (16장 SM 설명 참조)

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);
```

3. 단일 커널 호출

- 커널 호출 오버헤드 최소화
- Persistent threads 패턴

최적 Grid 크기 선택

공식

```
int numSMs;  
cudaDeviceGetAttribute(&numSMs,  
    cudaDevAttrMultiProcessorCount, 0);  
  
// Method 1: Minimum 2 blocks per SM  
int numBlocks = numSMs * 2;  
  
// Method 2: Occupancy-based  
int blockSize = 256;  
int minGridSize;  
cudaOccupancyMaxPotentialBlockSize(  
    &minGridSize, &blockSize, kernel, 0, 0);
```

GPU별 권장 설정

GPU	SMs	권장 Grid	Block	총 스레드
GTX 1070	15	30-60	256	7,680-15,360
RTX 3090	82	164-328	256	41,984-83,968
A100	108	216-432	256	55,296-110,592

함수 타입 한정자

CUDA는 함수가 어디서 호출되고 어디서 실행되는지를 명시하는 한정자를 제공합니다.

__global__

- Host에서 호출, Device에서 실행 (커널)
- 반환 타입은 반드시 `void`

__device__

- Device에서 호출, Device에서 실행
- 커널이나 다른 `__device__` 함수에서만 호출 가능

함수 타입 한정자 (계속)

host

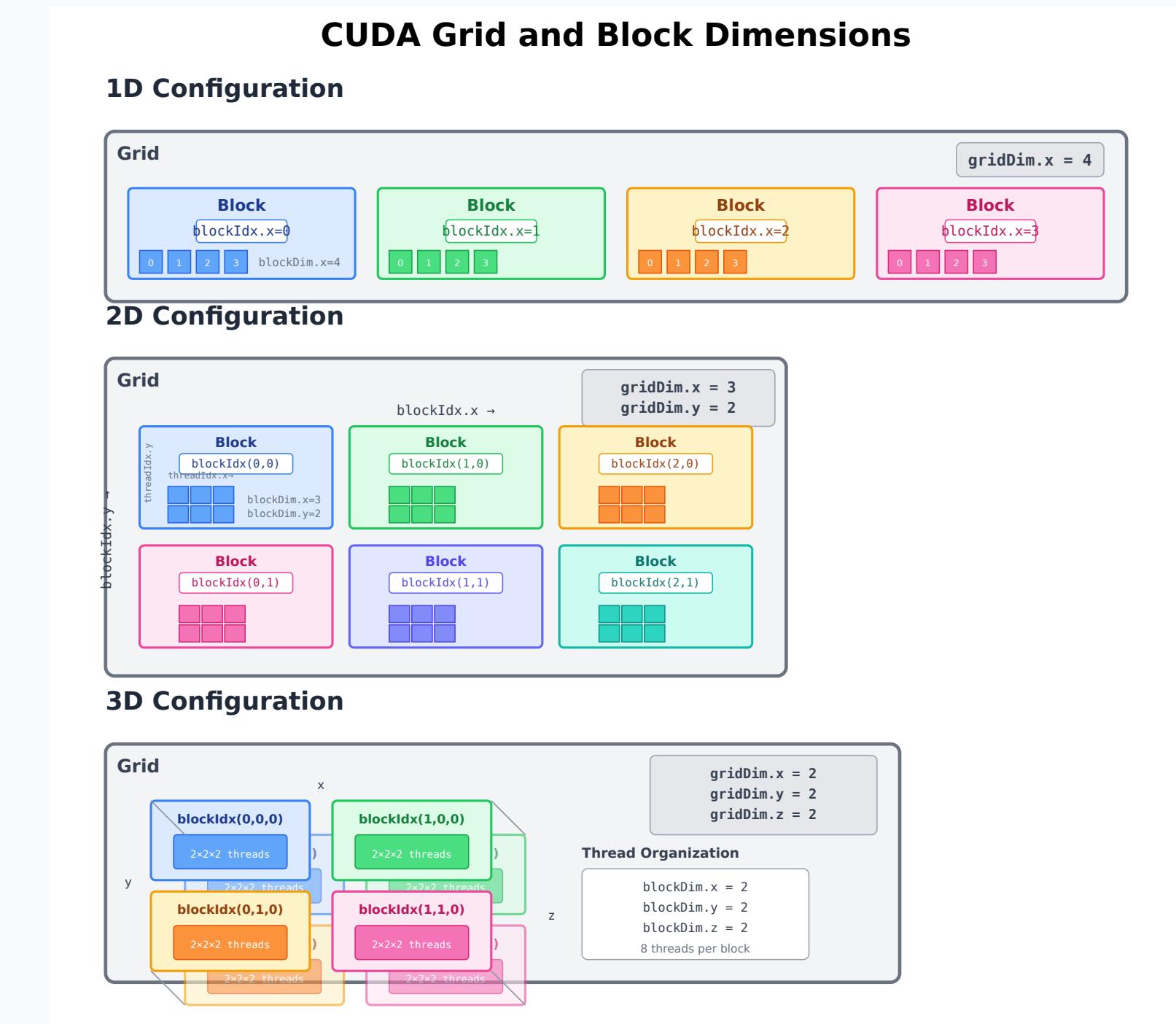
- Host에서 호출, Host에서 실행 (기본값)

host device

- Host와 Device 양쪽에서 모두 사용 가능

```
// Host-Device 데코레이터
__host__ __device__ float compute(float x, float y) {
    #ifdef __CUDA_ARCH__
        // GPU 데코레이터
        return x * y + 1.0f;
    #else
        // CPU 데코레이터
        return x * y - 1.0f;
    #endif
}
```

다차원 인덱싱



CUDA는 1D, 2D, 3D 인덱싱을 지원하여 다차원 데이터 처리를 쉽게 만듭니다.

2D 인덱싱 계산과 활용

2D 스레드 인덱스 계산

```
int col = threadIdx.x + blockIdx.x * blockDim.x;  
int row = threadIdx.y + blockIdx.y * blockDim.y;  
int idx = row * width + col; // 2D 인덱스
```

계산 원리

- **x축:** 열(column) 인덱스
- **y축:** 행(row) 인덱스
- **선형화:** 2D → 1D 변환

활용과 주의사항

주요 활용 분야

- 이미지/행렬 처리
- 2D 그리드 계산
- 공간 데이터 처리

필수 체크사항

```
if (col < width && row < height) {  
    // 2D 인덱스 체크  
    output[idx] = process(input[idx]);  
}
```

중요: 항상 경계 체크 필요!

실행 구성 최적화

적절한 블록과 그리드 크기를 선택하는 것은 성능에 큰 영향을 미칩니다.

최적화 핵심 요소

그리드 크기 계산

```
int gridSize = (n + blockSize - 1) / blockSize;
```

Occupancy 기반 최적화

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(  
    &maxActiveBlocks, kernel, blockSize, 0);
```

실행 구성 최적화 (계속)

고려사항

- 블록 크기: 32의 배수 (warp 크기)
- SM당 활성 블록: GPU 리소스 활용
- 메모리 접근 패턴: Coalesced access

일반적인 블록 크기

- 1D: 128, 256, 512
- 2D: (16,16), (32,8)
- 3D: (8,8,8)

실습 파일

- `exec_config_test.cu` : 다양한 구성 테스트
- `occupancy_calculator.cu` : 점유율 계산

3차원 스레드 구성

3D 인덱싱 계산

```
// 3D 인덱스 계산  
int x = threadIdx.x + blockIdx.x * blockDim.x;  
int y = threadIdx.y + blockIdx.y * blockDim.y;  
int z = threadIdx.z + blockIdx.z * blockDim.z;  
  
// 1D 인덱스 계산  
int width = gridDim.x * blockDim.x;  
int height = gridDim.y * blockDim.y;  
int linearIdx = z * width * height + y * width + x;
```

활용 예시

데이터 타입	차원 구성	용도
2D 이미지	(width, height, 1)	이미지 처리
3D 볼륨	(x, y, z)	CT/MRI 데이터
비디오	(width, height, frames)	프레임 병렬 처리

문제: GPU가 조용히 실패한다!

실제 사례: 조용한 실패

```
// Developer's code
cudaMalloc(&d_data, 10000000000); // Trying to allocate 10GB
kernel<<<1000000, 1024>>>(d_data); // Exceeds allowed range
cudaMemcpy(h_data, d_data, size, ...); // Failed but continues

// Result: Strange values, crashes, 3 days of debugging
```

왜 CUDA는 조용히 실패하나?

- **비동기 실행:** 에러가 나중에 발견됨
- **에러 전파 없음:** 하나 실패해도 계속 진행
- **디버깅 어려움:** 어디서 실패했는지 모름

해결책: 반드시 에러 체크!

CUDA 에러 처리 - 필수 기법

필수 에러 체크 매크로 (모든 프로젝트에 필수!)

```
#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)); \  
        } \  
    } while (0)
```

왜 이렇게 해야 하나?

```
// 예상 오류: 메모리 할당  
cudaMalloc(&d_data, size); // 예상 오류  
kernel<<<grid, block>>>(d_data); // 예상 오류  
  
// 실제 오류: 메모리 할당  
CUDA_CHECK(cudaMalloc(&d_data, size)); // 실제 오류
```

효과

- 즉시 탐지: 에러 발생 지점 파악
- 디버깅 시간 단축: 3일 → 3분
- 안정성 향상: 프로덕션 품질

커널 내부 디버깅

커널 디버깅 기법

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

    // 1. 범위 체크 (인덱스 유효!)
    if (idx >= N) return;

    // 2. 초기화 출력
    if (idx == 0) {
        printf("Kernel: N=%d, blocks=%d\n", N, gridDim.x);
    }

    // 3. assert 체크 (데이터 유효)
    assert(data != nullptr);

    data[idx] = idx * 2.0f;
}
```

컴파일: `nvcc -G -g kernel.cu` (디버그 모드)

CUDA 디버깅 기법

커널 내 printf 사용

```
__global__ void debugKernel(int *data) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid == 0) { // 블록 인덱스
        printf("Block %d: data[0] = %d\n", blockIdx.x, data[0]);
    }
    if (data[tid] < 0) { // 데이터 범위
        printf("Error at index %d\n", tid);
    }
}
```

디버깅 체크리스트

단계	확인 사항	방법
시작	작은 데이터 테스트	N=32로 시작
검증	CPU 결과와 비교	병렬 구현
범위	인덱스 범위 체크	if (tid < N)
동기화	스레드 동기화	__syncthreads()

CUDA 초보자 흔한 실수 - Part 1

실수 1: 메모리 크기 누락

잘못된 코드

```
float *d_data;  
cudaMalloc(&d_data, N); // N은 float의 개수  
// N은 N bytes로 표기 (float는 4 bytes)
```

올바른 코드

```
float *d_data;  
cudaMalloc(&d_data, N * sizeof(float)); // N은 float의 개수  
// N은 float는 bytes로 표기
```

결과: Segmentation fault 또는 쓰레기 값

CUDA 초보자 흔한 실수 - Part 2

실수 2: 동기화 누락

잘못된 코드

```
kernel<<<grid, block>>>();  
cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);  
// 빠른 결과 출력
```

올바른 코드

```
kernel<<<grid, block>>>();  
cudaDeviceSynchronize(); // 동기화  
cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);
```

결과: 불완전한 결과 또는 이전 결과 복사

CUDA 초보자 흔한 실수 - Part 3

실수 3: 범위 체크 누락

잘못된 코드

```
__global__ void kernel(float *data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    data[tid] = data[tid] * 2; // tid >= N일 때!
}
```

올바른 코드

```
__global__ void kernel(float *data, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < N) { // 범위 체크
        data[tid] = data[tid] * 2;
    }
}
```

결과: Memory access violation

추가 흔한 실수들

실수 4: 잘못된 Grid/Block 크기

```
// 00: N / blockSize 0000000 00 0
int gridSize = N / blockSize; // 000 000

// 00: 00 00
int gridSize = (N + blockSize - 1) / blockSize;
```

실수 5: Shared Memory 동기화 누락

```
__shared__ float sdata[256];
sdata[tid] = input[tid];
// __syncthreads() 00!
float value = sdata[tid + 1]; // 00 0000 00 0 00 00

// 000 00
sdata[tid] = input[tid];
__syncthreads(); // 00 000 00
float value = sdata[tid + 1];
```

황금 규칙과 체크리스트

커널 시작 패턴

```
__global__ void kernel(float *data, int N) {
    // 1. Thread ID
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. 범위 체크 (tid < N)
    if (tid >= N) return;

    // 3. 초기화
    data[tid] = ...;
}
```

디버깅 체크리스트

- cudaMalloc 시 sizeof() 사용했는가?
- 커널 호출 후 에러 체크했는가?
- 범위 체크를 했는가?
- Grid 크기 계산 시 올림 처리했는가?
- Shared memory 사용 시 __syncthreads() 했는가?

Part 1 정리 - CUDA 기초 완료

학습한 내용

영역	핵심 개념	실습
병렬 컴퓨팅	CPU 한계, GPU 장점	성능 비교
GPU 아키텍처	SM, CUDA 코어, Warp	GPU 정보 조회
CUDA 모델	Host/Device, 커널	Hello CUDA
스레드 계층	Grid → Block → Thread	인덱스 계산
메모리 관리	cudaMalloc, cudaMemcpy	벡터 덧셈
에러 처리	CUDA_CHECK 매크로	디버깅 기법

핵심 용어 정리

- **커널(Kernel)**: GPU에서 병렬 실행되는 함수
- **워프(Warp)**: 32개 스레드의 실행 단위
- **SM**: Streaming Multiprocessor
- **Compute Capability**: GPU 아키텍처 버전

다음: Part 2에서 메모리 최적화와 고급 패턴 학습