GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드

3강. 비싼 GPU, 왜 놀고 있을까?

- CUDA 성능을 잡아먹는 진짜 범인 찾기 –

목표:
진짜 범인의 정체를 밝힌다
비밀무기: 공유메모리
비밀무기로 병목현상을 돌파하자
NSIGHT가 증명하는 성능향상

3.1 범인의 정체 (CPU처럼 생각하는 개발자)

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

int main() {
    const int numElements = 1024;
    const int numIterations = 1000;
    std::vector<float> h_a(numElements, 1.0f);
    float *d_a = nullptr, *d_b = nullptr, *d_c = nullptr;
    cudaMalloc((void**)&d_a, sizeof(float) * numElements);
                ….
    for (int I = 0; I < numIterations; ++i)
    {
        cudaMemcpy(d_a, h_a.data(), sizeof(float) * numElements, cudaMemcpyHostToDevice);
        addKernel<<< (numElements + 255) / 256, 256 >>>(d_c, d_a, d_b, numElements);
        cudaMemcpy(h_c.data(), d_c, sizeof(float) * numElements, cudaMemcpyDeviceToHost);
        cudaDeviceSynchronize();
    }
    std::cout << "Execution finished." << std::endl;
    // 메모리 해제
    cudaFree(d_a);
    return 0;
}
```

병렬처리에 너무 적은 데이터

비싼 운반비 발생(PCI통신):덤프에 사과1상자

비싼 운반비 발생(PCI통신):덤프에 사과1상자

CPU-GPU 동기화로 인한 유휴 시간 발생

3.2 단계1 최적화: 통신 오버헤드 제거

```
// FOR 루프가 시작됩니다 (1000번 반복)
FOR (INT I = 0; I < 1000; I++) {

    // 1. [매번] GPU로 데이터 운반 (비싼 작업)
    // - 마치 덤프트럭을 불러 사과 1상자  싣는 것
    CUDAMEMCPY(..., CUDAMEMCPYHOSTTODEVICE);

    // 2. [매번] GPU가 계산
    // - GPU는 눈 깜짝할 사이에 계산을 끝냅니다.
    ADDKERNEL<<<...>>>(...);

    // 3. [매번] CPU로 결과물 운반 (비싼 작업)
    // - 계산이 끝난 사과 상자 '하나'를 다시 덤프트럭에 실어
보냅니다.
    CUDAMEMCPY(..., CUDAMEMCPYDEVICETOHOST);

    // 4. [매번] 운반이 끝날 때까지 모두 대기 (가장 큰 병목)
    CUDADEVICESYNCHRONIZE();
}
```

문제의 본질: GPU의 계산 속도는 매우 빠르지만, 데이터를 가져오고(CUDAMEMCPY), 작업이 끝났는지 확인하는(CUDADEVICESYNCHRONIZE) 과정은 매우 느림. 이 느린 과정을 1000번이나 반복하니, GPU는 대부분의 시간을 계산이 아닌 '대기' 상태로 보내게 됨. NSIGHT 프로파일의 ' 텅 빈 시간 ' 이 바로 이 증거

```
// 1. [단 한 번만!] 필요한 모든 데이터를 한 번에 GPU로 운반
// - 덤프트럭을 불렀을 때, 사과 1000상자를 한 번에 모두 운반

CUDAMEMCPY(..., CUDAMEMCPYHOSTTODEVICE);

// FOR 루프가 시작됩니다 (1000번 반복)
FOR (INT I = 0; I < 1000; I++) {

    // 2. [오직 계산만 반복]
    // - GPU는 외부와의 통신 없이, 이미 가져온 데이터를 가지고
    //   자신이 가장 잘하는 '계산'에만 집중합니다.
    ADDKERNEL<<<...>>>(...);
}

// 3. [단 한 번만!] 모든 계산이 끝난 것을 확인
CUDADEVICESYNCHRONIZE();

// 4. [단 한 번만!] 최종 결과물을 한 번에 CPU로 운반
// - 1000개의 계산이 모두 끝난 결과물을 덤프트럭에 실어
보냅니다.
CUDAMEMCPY(..., CUDAMEMCPYDEVICETOHOST);
```

해결책: 가장 비싼 작업인 '데이터 운반'과 '동기화'를 루프 밖으로 이동. 코드의 위치를 몇 줄 옮겼을 뿐이지만, 성능에 미치는 영향은 하늘과 땅 차이. GPU는 더 이상 기다리지 않고 쉴 새 없이 일하게 됨

3.3 단계2 최적화: 공유메모리

```
// 1단계 개선을 마친 커널 코드
__GLOBAL__ VOID KERNEL_STEP1(...)
{
    INT IDX = /* ... 인덱스 계산 ... */;

    IF (IDX < N) {
        // [남아있는 문제점]
        // 모든 스레드가 여전히 칩(SM) 바깥의
        // '느린' GLOBAL MEMORY(DRAM)에 직접 접근
        //
        // 비유: 재료는 모두 주방에 들어왔지만,
        // 요리사가 재료 하나하나를 저 멀리 있는
        // 중앙 창고에서 직접 가져오는 상황.
        C[IDX] = A[IDX] + B[IDX];
    }
}
```

```
__GLOBAL__ VOID KERNEL_FINAL(...)
{
    // 1. SM 내부에 초고속 '작업대'(__SHARED__ MEMORY) 선언
    __SHARED__ FLOAT S_A[TILE_SIZE];
    __SHARED__ FLOAT S_B[TILE_SIZE];

    INT IDX = /* ... 글로벌 인덱스 계산 ... */;
    INT TID = THREADIDX.X; // 블록 내 로컬 인덱스

    // 2. [협업] 블록 내 모든 스레드가 힘을 합쳐
    // 필요한 데이터를 GLOBAL MEMORY에서 '작업대'로 한 번에
    S_A[TID] = A[IDX];
    S_B[TID] = B[IDX];

    // 3. [동기화] 모든 동료가 재료를 가져올 때까지 대기
    __SYNCTHREADS();

    // 4. [초고속 연산] 이제 칩 외부로 나갈 필요 없이,
    // 바로 옆에 있는 초고속 '작업대'의 데이터를 사용한다!
    C[IDX] = S_A[TID] + S_B[TID];
}
```

변수 c의 위치는?

전역메모리!

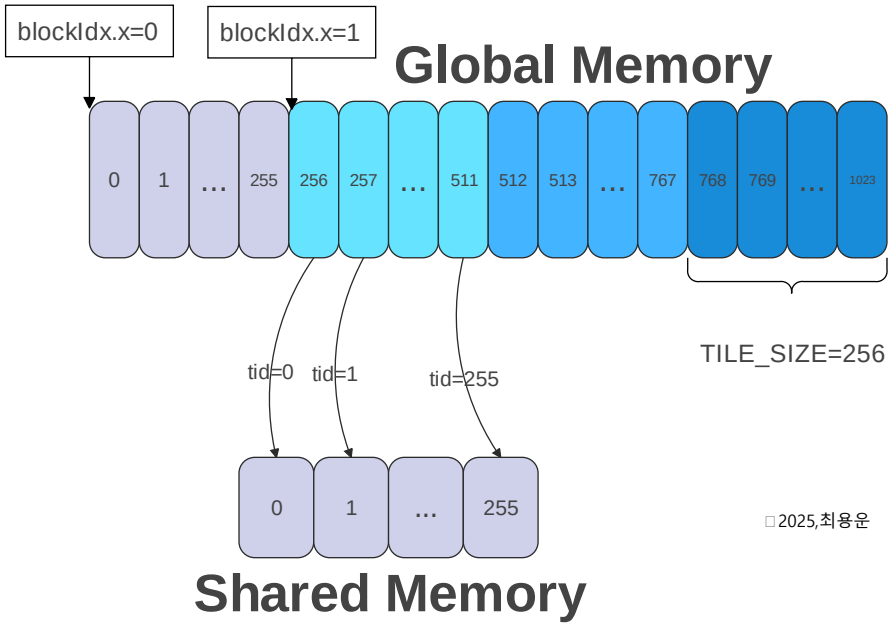
3.4 심화학습:공유 메모리 실전 코딩

타일링 (TILE_SIZE)

거대문제를 한꺼번에 처리하는 대신 스레드 블록 단위로 잘라서 처리
TILE_SIZE가 바로 타일의 크기, 스레드 블록크기와 동일하게 설정

2개의 인덱스를 사용

Idx: 글로벌 인덱스, 전체 데이터에서 나의 절대적 위치 (전역메모리)
Tid: 로컬 인덱스, 스레드 블록 내에서 상대적 위치 (공유메모리)




```
// --- 호스트 코드 (Kernel Launch) ---
const int TILE_SIZE = 256; // 타일 크기 = 블록 크기
// 전체 데이터를 타일 크기로 나누어 블록의 수를 계산
int numBlocks = (numElements + TILE_SIZE - 1) / TILE_SIZE;
kernel<<<numBlocks, TILE_SIZE>>>(...);

// --- 디바이스 코드 (Kernel) ---
__global__ void kernel_final(...) {
__shared__ float s_data[TILE_SIZE];

    // 1. 나의 '두 가지' 주소 계산
    int idx = blockIdx.x * blockDim.x + threadIdx.x; // 글로벌 주소
    int tid = threadIdx.x; // 로컬 주소

    // 2. 데이터 로딩 (가장 중요한 부분!)
    // 만약 전체 데이터 범위를 벗어나지 않았다면,
    if (idx < numElements) {
        // 나의 글로벌 주소(idx)에서 데이터를 가져와
        // 우리 팀 공유 공간의 로컬 주소(tid)에 저장한다.
        s_data[tid] = global_data[idx];
    }
    // 3. 동기화: 모든 팀원이 로딩을 마칠 때까지 대기
    __syncthreads();
    ...
}
```

3.5 전역메모리 변수 C

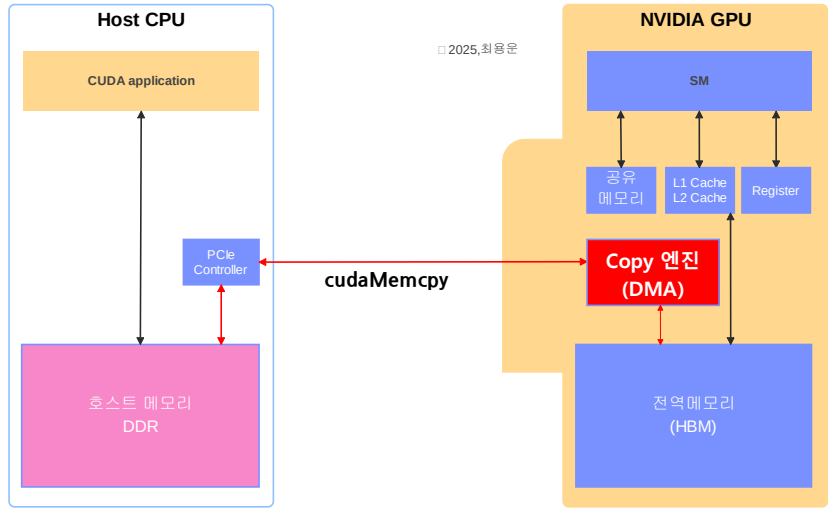
1. HOST로 통하는 유일한 관문, 전역 메모리

- GPU가 계산한 값은 결국 HOST에게 전송되어야 함
- 오직 cudaMemcpy 함수를 사용하여 Host – GPU 데이터 전달한다
- cudaMemcpy함수는 오직 전역메모리를 대상으로 작동하기때문에
- 결과값을 저장하는 변수 C는 반드시 전역메모리에 위치해야 함

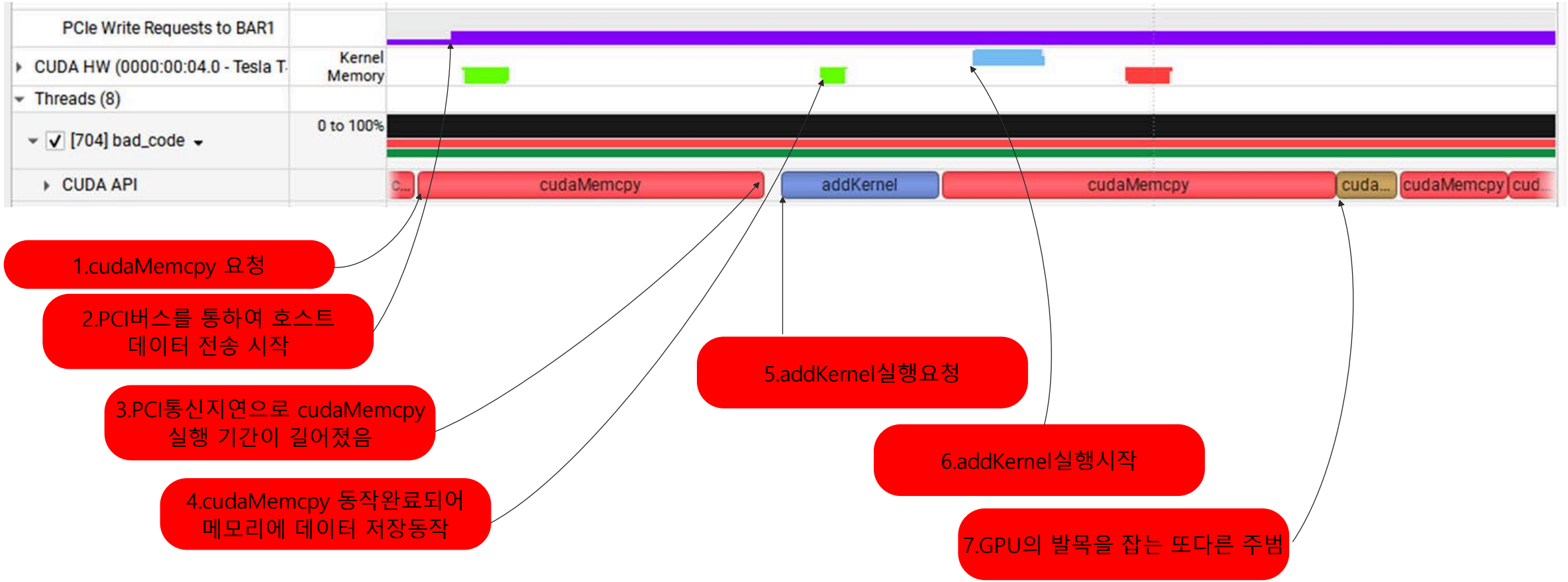
2. 공유메모리는 빠르지만, 너무 작아요!

- 공유 메모리는 수십 KB정도로 매우 작지만
- GPU가 처리한 결과값은 MB ~ GB단위로 매우 크다 (공유메모리에 저장 불가)
- 혹시, C의 매우 작은 부분만을 공유메모리에 저장?
  - 한정된 공유메모리 낭비 가능성 (인로인한 GPU 전체 작업성능 저하)

최종요리 (C) 를 개인 작업대 (공유메모리)에 쌓아둘 순 없다!



3.6 BAD_CODE (NSIGHT 프로파일)

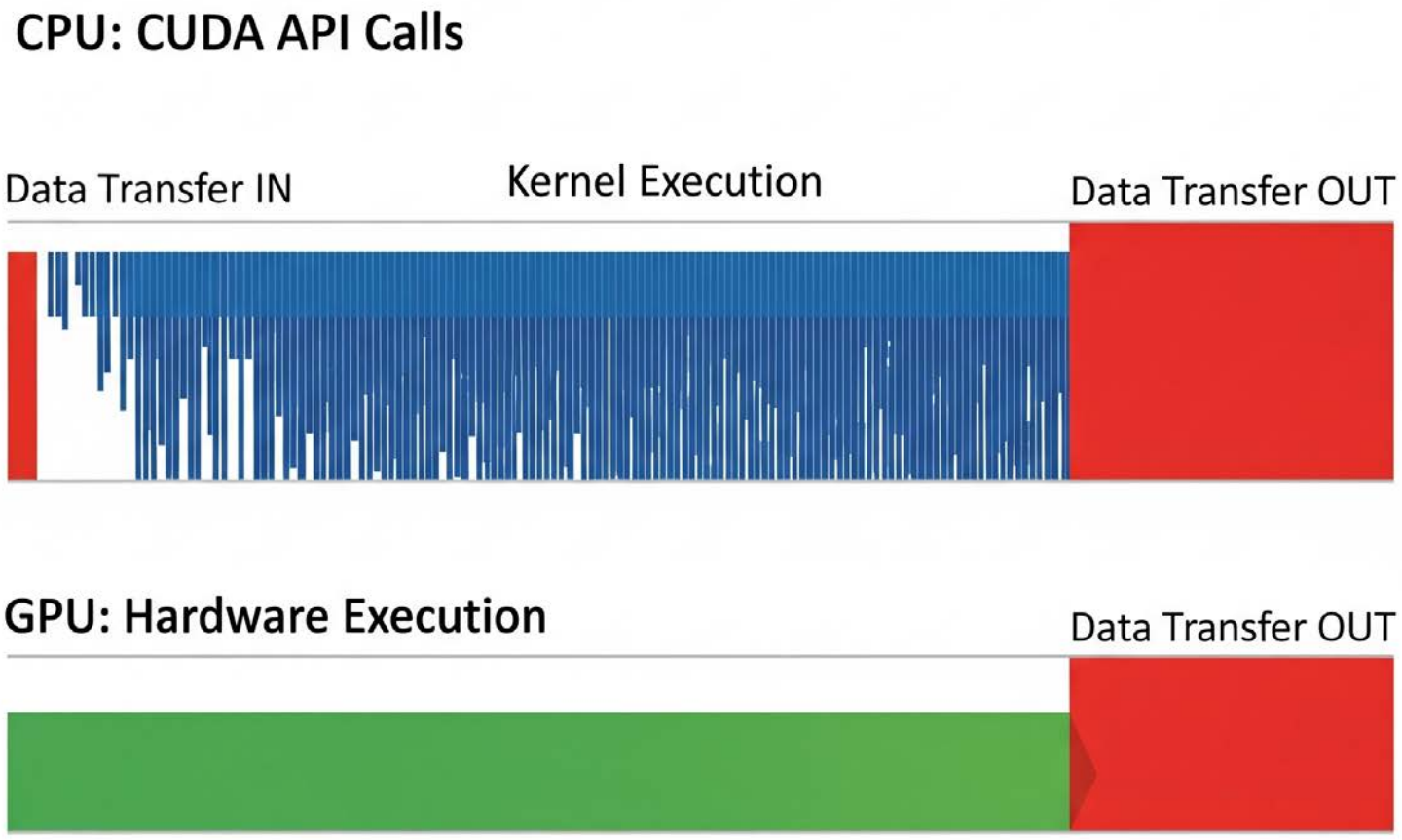


3.7 GOOD_CODE #1 (통신오버헤드 제거)

Loop 밖으로 cudaMemcpy 이동

계산 전 데이터 전송 (앞부분 빨강)
계산 후 데이터 전송 (뒤부분 빨강)

GPU 점유율 증가 (녹색)

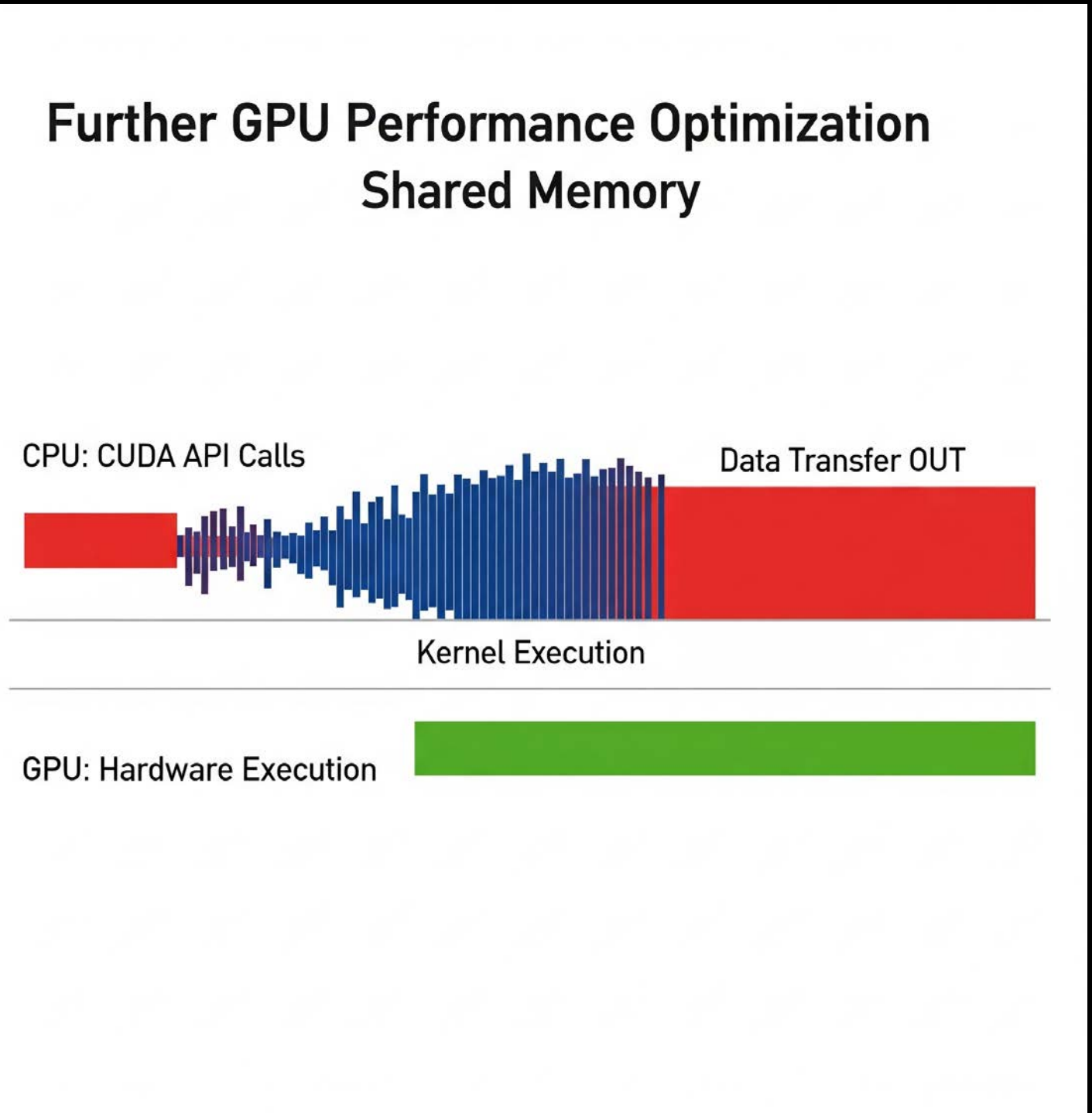


3.8 GOOD_CODE #2 (공유메모리 최적화)

__shared__ 변수명
자주 사용하는 변수 공유메모리 내재화

GPU점유율 100%유지 +
전체 계산기간(녹색) 감소함

메모리 계층구조를 활용한 최적화의 힘



3.9 부록: 전체 코드

```
#include <iostream>
#include <cuda_runtime.h>
// CUDA API 호출 에러 체크를 위한 매크로 (좋은습관)
#define CUDA_CHECK(err) {
 cudaError_t err_ = (err);
 if (err_ != cudaSuccess) {
 std::cerr <<  " CUDA Error at  "  << __FILE__ <<  " : "  << __LINE__ \
 <<  "  -  "  << cudaGetErrorString(err_) << std::endl;
 exit(EXIT_FAILURE); \   }
 }

/**
 * @brief 공유 메모리를 사용하여 두 벡터를 더하는 CUDA 커널
 * [최적화 포인트 2] 커널 내부 최적화 (Slide 3.8에 해당)
 * - __shared__ 키워드를 사용하여 전역 메모리 접근을 최소화하고 빠른 공유 메모리를 활용합니다. */
 __global__ void addVectorWithSharedMemory(const float* a, const float* b, float* c, int n) {
 // --- 공유 메모리 최적화 4단계 프로세스 ---
 // 1단계: 선언 (Declaration)
 // 각 스레드 블록이 사용할 공유 메모리 타일(Tile)을 선언합니다.
 // blockDim.x는 블록 당 스레드의 수를 나타냅니다.
 extern __shared__ float s_data[];  // 동적 공유 메모리 할당, 정적 할당방식보다 더 유연하고 고급진 방식
 float* s_a = s_data;
 float* s_b = &s_data[blockDim.x];
```

3.9 부록: 전체 코드

```
// 2단계: 협업 로딩 (Cooperative Loading)
// 각 스레드가 전역 메모리(a, b)에서 하나의 데이터를 가져와 공유 메모리(s_a, s_b)에 저장합니다.
 // - idx: 전역 메모리 상의 전체 데이터 인덱스
 // - tid: 블록 내에서의 스레드 인덱스 (공유 메모리 인덱스로 사용)
 unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
 unsigned int tid = threadIdx.x;
 if (idx < n) {
 s_a[tid] = a[idx];
 s_b[tid] = b[idx];
 }
// 3단계: 동기화 (Synchronization) // 블록 내의 모든 스레드가 공유 메모리에 데이터 로딩을 마칠 때까지 기다립니다.
 // 이 __syncthreads()가 없으면, 일부 스레드는 데이터가 로딩되기도 전에 연산을 시작할 수 있습니다.
 __syncthreads();
 // 4단계: 연산 (Computation)
 // 이제 모든 연산은 매우 빠른 '공유 메모리'를 통해 수행됩니다.
 // 전역 메모리에 직접 접근하는 것보다 훨씬 빠릅니다.
 if (idx < n) {
 c[idx] = s_a[tid] + s_b[tid];
 }
 }
```

3.9 부록: 전체 코드

```
int main() {
 int n = 1 << 20; // 1,048,576개의 원소
 size_t bytes = n * sizeof(float);
 // Host 메모리 할당
 float* h_a = new float[n];
 float* h_b = new float[n];
 float* h_c = new float[n];
 // 데이터 초기화
 for (int i = 0; i < n; ++i) {
 h_a[i] = static_cast<float>(i);
 h_b[i] = static_cast<float>(i * 2);
 }
 // Device 메모리 할당
float *d_a, *d_b, *d_c;
 CUDA_CHECK(cudaMalloc(&d_a, bytes));
 CUDA_CHECK(cudaMalloc(&d_b, bytes));
 CUDA_CHECK(cudaMalloc(&d_c, bytes));
 // --- [최적화 포인트 1] 통신 오버헤드 제거 (Slide 3.7에 해당) ---
 // 반복문 밖에서 '한 번만' 필요한 모든 데이터를 GPU로 복사합니다.
 std::cout << "1. Copying data from Host to Device..." << std::endl;
 CUDA_CHECK(cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice));
 CUDA_CHECK(cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice));
```

3.9 부록: 전체 코드

```
// 커널 실행 설정
 int threadsPerBlock = 256;
 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
 // 공유 메모리 크기 설정 (float 2개를 각 스레드가 사용하므로)
 size_t sharedMemSize = threadsPerBlock * sizeof(float) * 2;
 std::cout << "2. Launching CUDA Kernel..." << std::endl;
 addVectorWithSharedMemory<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_a, d_b, d_c, n);
 // 커널 실행이 끝난 후, '한 번만' 결과를 Host로 다시 복사합니다.
 std::cout << "3. Copying result from Device to Host..." << std::endl;
 CUDA_CHECK(cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost));
 // 결과 검증 (생략 가능)
 // ...
 std::cout << "4. Done." << std::endl;
 // 메모리 해제
 delete[] h_a;
 delete[] h_b;
 delete[] h_c;
 CUDA_CHECK(cudaFree(d_a));
 CUDA_CHECK(cudaFree(d_b));
 CUDA_CHECK(cudaFree(d_c));
 return 0;
 }
```

3.10 핵심정리(코드 최적화)

- HOST코드(main함수)
  - Bad_code에 있던 for 루프 안의 cudaMemcpy가 완전 사라졌음
  - 대신 커널 실행 전 cudaMemcpy를 **두 번 호출하여 모든 데이터를 미리 GPU로 보냄**.
  - 커널 실행이 끝나고 **다시 cudaMemcpy를 호출하여 결과만 가져오도록** 함
- (통신 오버헤드 제거)

- Kernel 코드 (addVectorWithSharedMemery함수)
  - _shared_ 키워드를 사용하여 커널 내부에 고속 캐시처럼 동작하는 공유 메모리를 선언함
  - 위 최적화로 GPU의 전역메모리 접근 횟수를 획기적으로 줄여 (3.8 참조) 전체 계산 시간 감소