GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드

4. 시간관리의 마술

4.1 비동기 요리와 스트림

GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드

전체 시리즈 소개 (bit.ly/GPU-COOK)

1강: GPU 속이 궁금해? - 주방의 모든 것 파헤치기 🍳

GPU는 CPU와 근본적으로 어떻게 다른지 알아봅니다. (안드로메다 주방장 vs 지구인 주방장)
SM, CUDA 코어, 워프 등 GPU의 핵심 부품들이 어떻게 작동하는지 하드웨어 관점에서 이해합니다.
레지스터부터 전역 메모리까지, 성능을 좌우하는 메모리 계층 구조의 비밀을 파헤칩니다.
핵심: GPU라는 주방의 구조와 도구를 완벽히 이해해야 최고의 요리가 시작됩니다.

2강: 숨은 병목 찾기 - 데이터 전송 시간을 잡아라! 🚚

GPU 연산이 아무리 빨라도 소용없는 이유, 바로 '데이터 전송'이라는 숨은 병목을 찾아냅니다.
NVIDIA Nsight Systems 프로파일러를 이용해 마치 CCTV처럼 GPU의 모든 동작을 감시하고 분석하는 법을 배웁니다.
핵심: 가장 비싼 작업은 계산이 아니라 '재료(데이터)를 주방으로 옮기는 시간'입니다.

3강: 마법의 작업대, 공유 메모리 - 최고의 요리(커널) 만들기 ✨

느려터진 전역 메모리 접근을 획기적으로 줄이는 비장의 무기, __shared__ 메모리의 정체와 사용법을 익힙니다.
'통신 오버헤드 제거'와 '공유 메모리 활용'이라는 2단계 최적화를 통해 커널 실행 속도를 극적으로 향상시킵니다.
핵심: 멀리 있는 창고(전역 메모리) 대신, 바로 앞 '마법의 작업대(공유 메모리)'를 활용해 요리 속도를 높입니다.

4강: 시간 관리의 마술 - 주방 전체를 춤추게 하라! 🧙‍♂️

요리(계산)와 재료 준비(데이터 전송)를 동시에 처리하는 비동기 프로그래밍과 CUDA 스트림의 개념을 마스터합니다.
GPU가 잠시도 쉬지 않고 100% 성능을 내도록 만들어, 전체 시스템의 처리량을 극대화하는 방법을 배웁니다.
핵심: 최고의 요리사(커널) 한 명보다, 주방 전체가 효율적으로 돌아가는 시스템이 진정한 고수입니다.

GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드

4.2 아주 자연스러운 프로그램

```
#include <iostream>
// GPU에서 수많은 스레드가 병렬로 실행할 실제 계산 로직 ( ' 요리 레시피 ' )
__global__ void vectorAdd(float *a, float *b, float *c, int n) {
    int I = blockIdx.x * blockDim.x + threadIdx.x;
    if (I < n) {
        c[i] = a[i] + b[i]; // 각 스레드는 C 벡터의 한 개 원소만 계산
    }
}
int main() {
    // --- 1. HOST 메모리,데이터 준비
    int n = 1 << 24; // 계산할 데이터 개수 (약 1,600만 개)
    size_t size = n * sizeof(float);
    float *h_a = (float *)malloc(size); // Host(CPU) 메모리에 A 배열 할당
    float *h_b = (float *)malloc(size); // Host(CPU) 메모리에 B 배열 할당
    float *h_c = (float *)malloc(size); // 결과를 담을 Host(CPU) 메모리 C 배열 할당

    // --- 2. GPU 메모리 준비 (Device)
    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size); // Device(GPU) 메모리에 A 배열 할당
    cudaMalloc(&d_b, size); // Device(GPU) 메모리에 B 배열 할당
    cudaMalloc(&d_c, size); // 결과를 담을 Device(GPU) 메모리 C 배열 할당

    // --- 3. 데이터 전송 (CPU → GPU) ---
    // Host의 h_a 내용을 Device의 d_a로 복사
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);

    // --- 4. GPU에서 계산 실행 ---
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    // vectorAdd 함수를 GPU에서 실행하도록 명령
    vectorAdd<<<numBlocks, blockSize>>>(d_a, d_b, d_c, n);

    // --- 5. 결과 전송 (GPU → CPU) ---
    // Device의 d_c 내용을 Host의 h_c로 복사
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // --- 6. 메모리 정리 ---
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    free(h_a); free(h_b); free(h_c);

    return 0;
}
```

4.3 아주 비효율적인 실행 - 동기식

4.HOST CPU: 75%이상
놀고있는 중

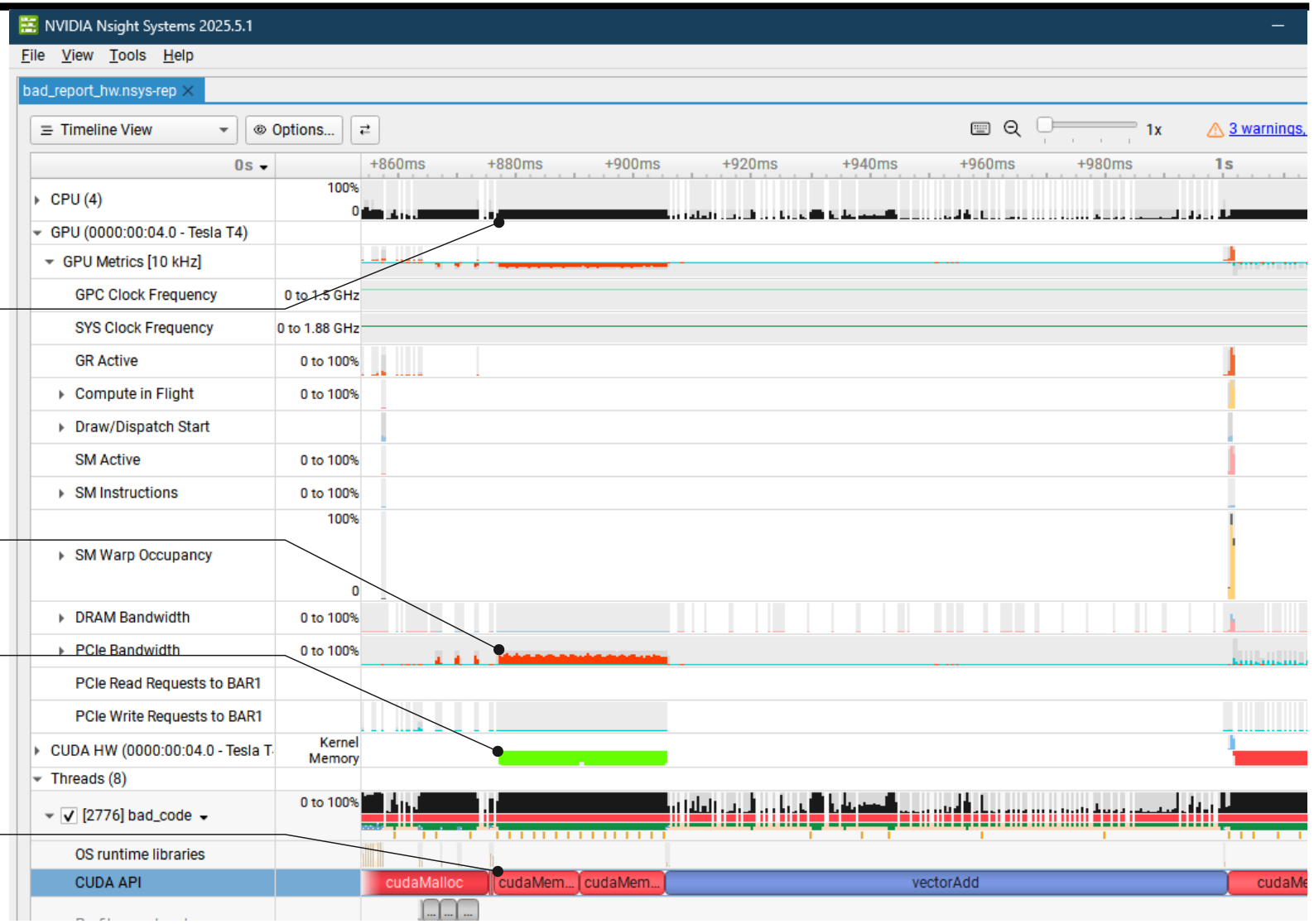
**PCI통신버스 여유있고,
CUDA HW또한 여유있는
상황인데 왜 CPU는 더 많은
데이터를 GPU에게
전송하지 못하는가?**
(Pinned Memory)

3.HOST-GPU통신버스
사용량 역시 60%이상
놀고있음

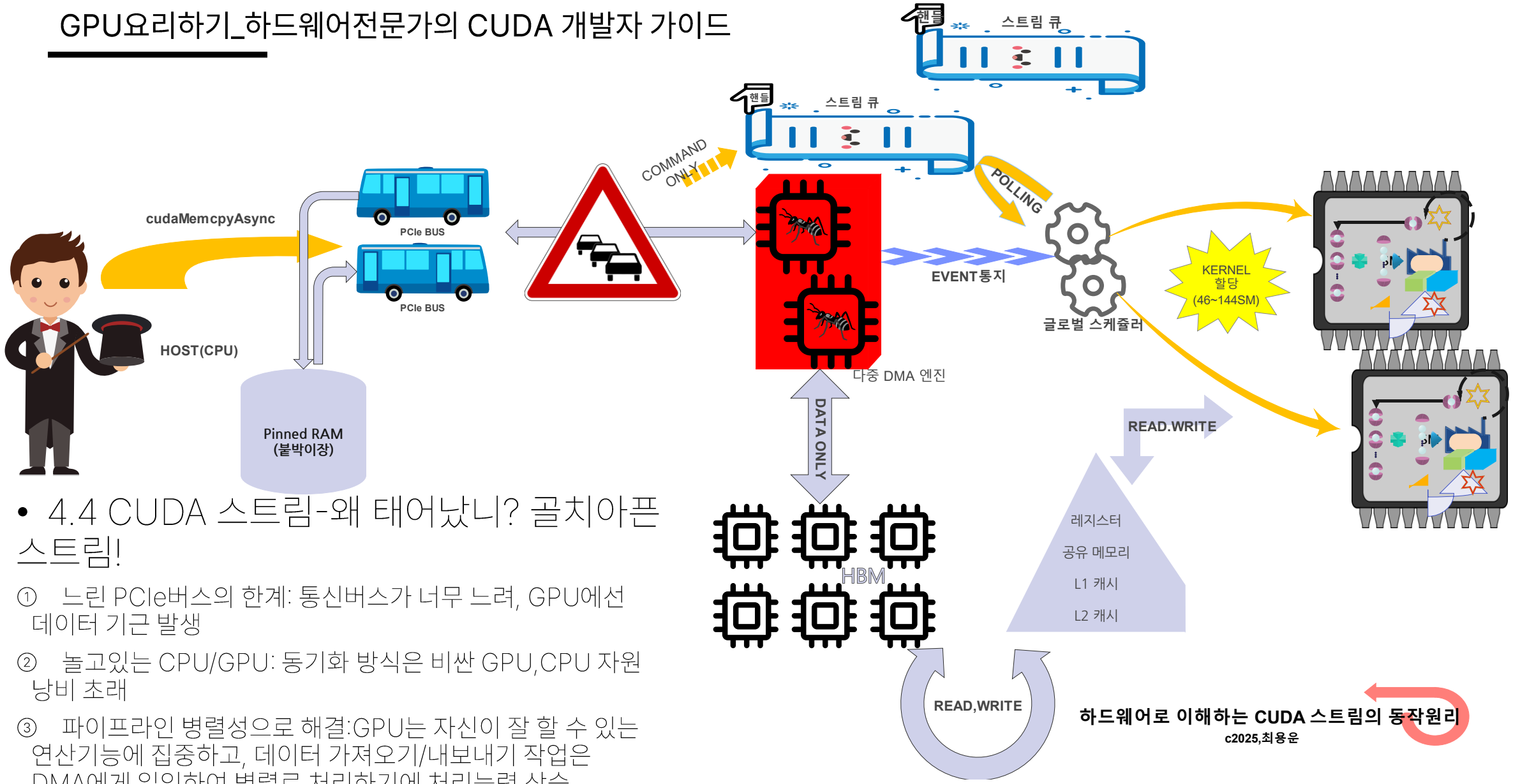
2.HOST에서 도착하는
데이터 전송율 높지 않아서,
CUDA메모리 시스템
50%이상 놀고있음

1.인간의 일처리 방식 처럼,
한번에 하나씩
순서적으로 API호출
(동기적처리)

**cudaMemcpy동작을
동시에 진행할 수 없을까?**
(auto copy by DMA)



GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드



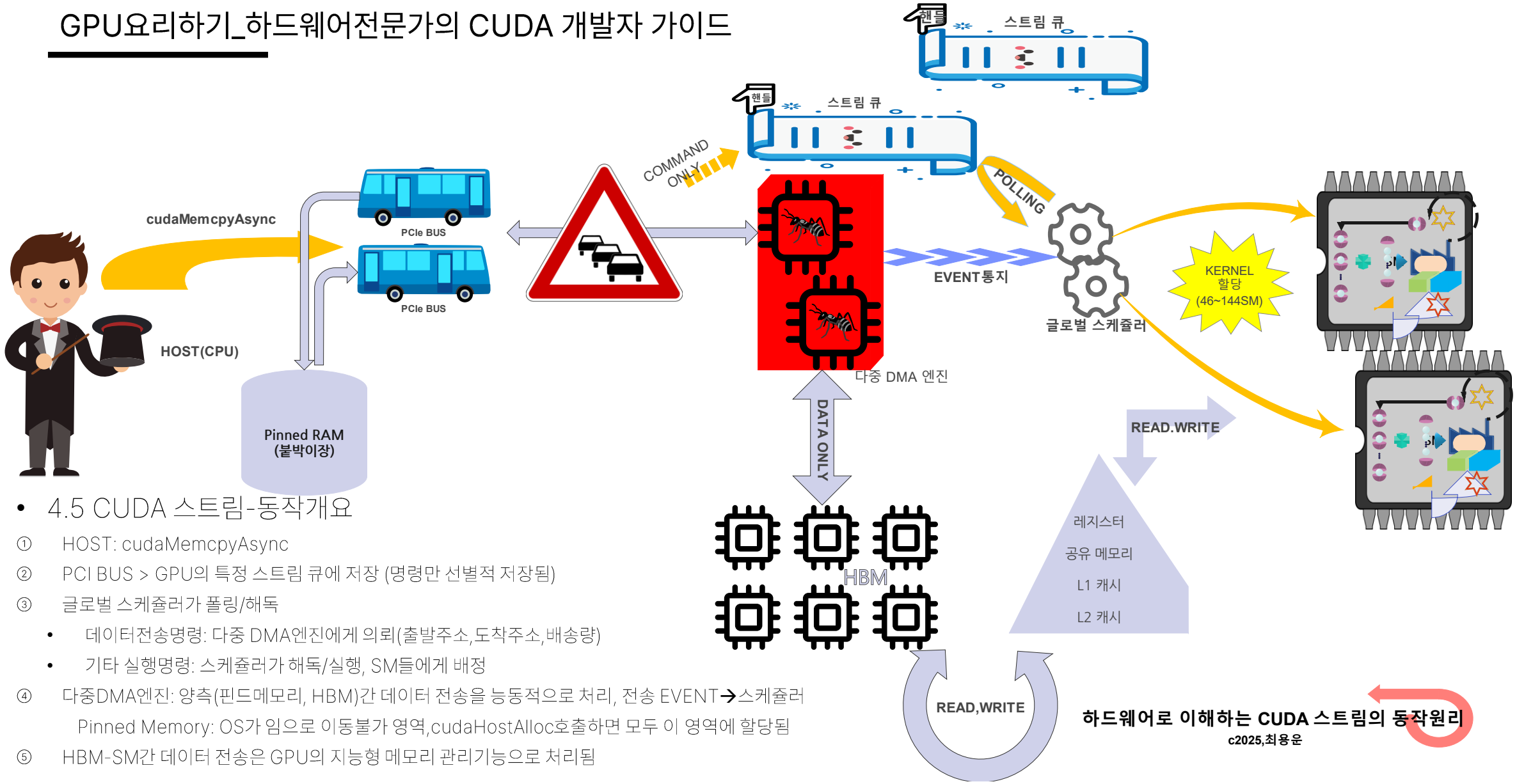
- 4.4 CUDA 스트림-왜 태어났니? 골치아픈 스트림!

① 느린 PCIe버스의 한계: 통신버스가 너무 느려, GPU에선 데이터 기근 발생

② 놀고있는 CPU/GPU: 동기화 방식은 비싼 GPU,CPU 자원 낭비 초래

③ 파이프라인 병렬성으로 해결:GPU는 자신이 잘 할 수 있는 연산기능에 집중하고, 데이터 가져오기/내보내기 작업은 DMA에게 일임하여 병렬로 처리하기에 처리능력 상승

GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드



- 4.5 CUDA 스트림-동작개요

① HOST: cudaMemcpyAsync

② PCI BUS > GPU의 특정 스트림 큐에 저장 (명령만 선별적 저장됨)

③ 글로벌 스케쥴러가 폴링/해독
- 데이터전송명령: 다중 DMA엔진에게 의뢰(출발주소,도착주소,배송량)
- 기타 실행명령: 스케쥴러가 해독/실행, SM들에게 배정

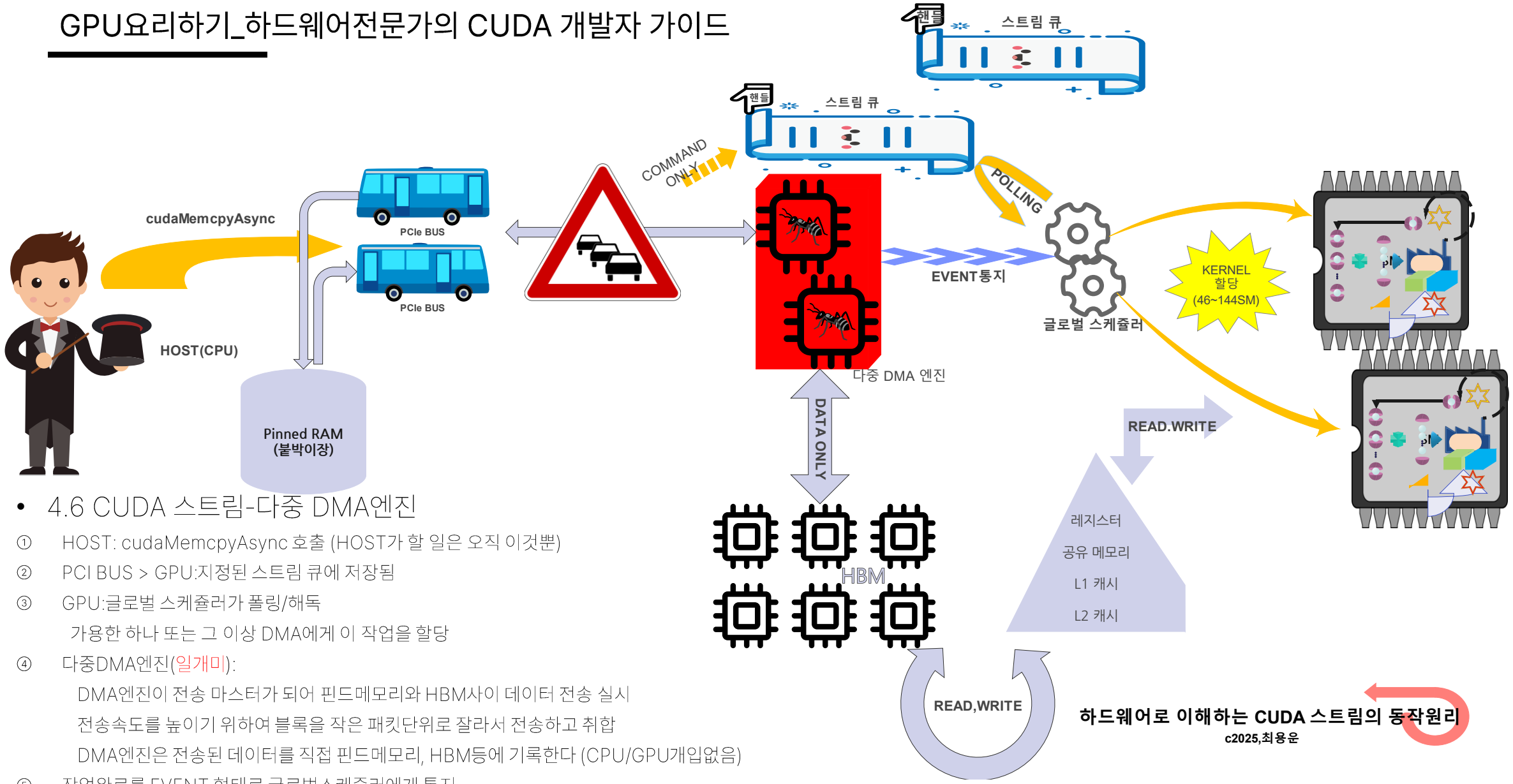
④ 다중DMA엔진: 양측(핀드메모리, HBM)간 데이터 전송을 능동적으로 처리, 전송 EVENT→스케쥴러
Pinned Memory: OS가 임으로 이동불가 영역,cudaHostAlloc호출하면 모두 이 영역에 할당됨

⑤ HBM-SM간 데이터 전송은 GPU의 지능형 메모리 관리기능으로 처리됨

하드웨어로 이해하는 CUDA 스트림의 동작원리

c2025,최용운

GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드



- 4.6 CUDA 스트림-다중 DMA엔진

① HOST: cudaMemcpyAsync 호출 (HOST가 할 일은 오직 이것뿐)

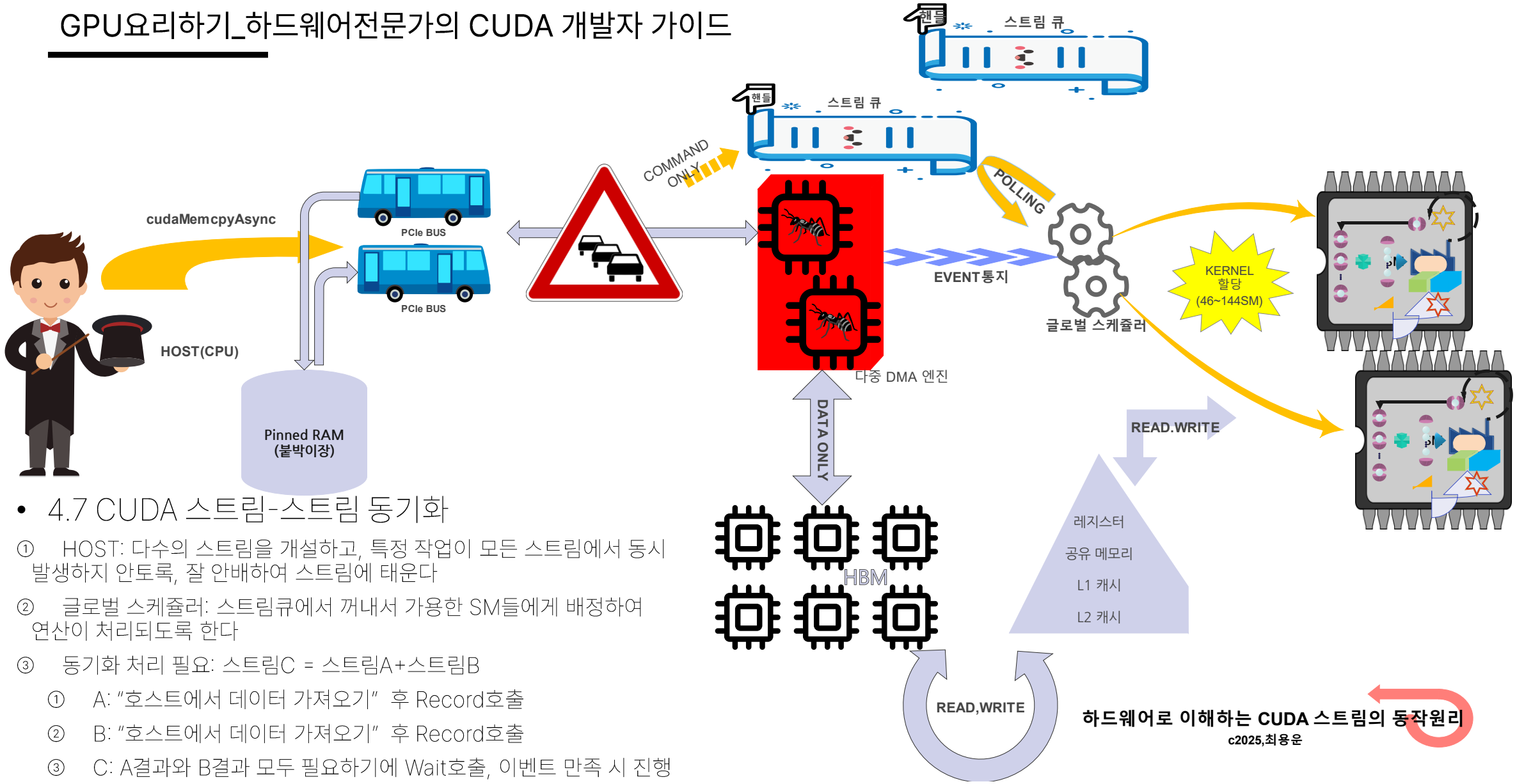
② PCI BUS > GPU:지정된 스트림 큐에 저장됨

③ GPU:글로벌 스케쥴러가 폴링/해독
가용한 하나 또는 그 이상 DMA에게 이 작업을 할당

④ 다중DMA엔진(일개미):
DMA엔진이 전송 마스터가 되어 핀드메모리와 HBM사이 데이터 전송 실시
전송속도를 높이기 위하여 블록을 작은 패킷단위로 잘라서 전송하고 취합
DMA엔진은 전송된 데이터를 직접 핀드메모리, HBM등에 기록한다 (CPU/GPU개입없음)

⑤ 작업완료를 EVENT 형태로 글로벌스케쥴러에게 통지

GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드



- 4.7 CUDA 스트림-스트림 동기화

① HOST: 다수의 스트림을 개설하고, 특정 작업이 모든 스트림에서 동시 발생하지 안토록, 잘 안배하여 스트림에 태운다

② 글로벌 스케쥴러: 스트림큐에서 꺼내서 가용한 SM들에게 배정하여 연산이 처리되도록 한다

③ 동기화 처리 필요: 스트림C = 스트림A+스트림B

① A: "호스트에서 데이터 가져오기" 후 Record호출

② B: "호스트에서 데이터 가져오기" 후 Record호출

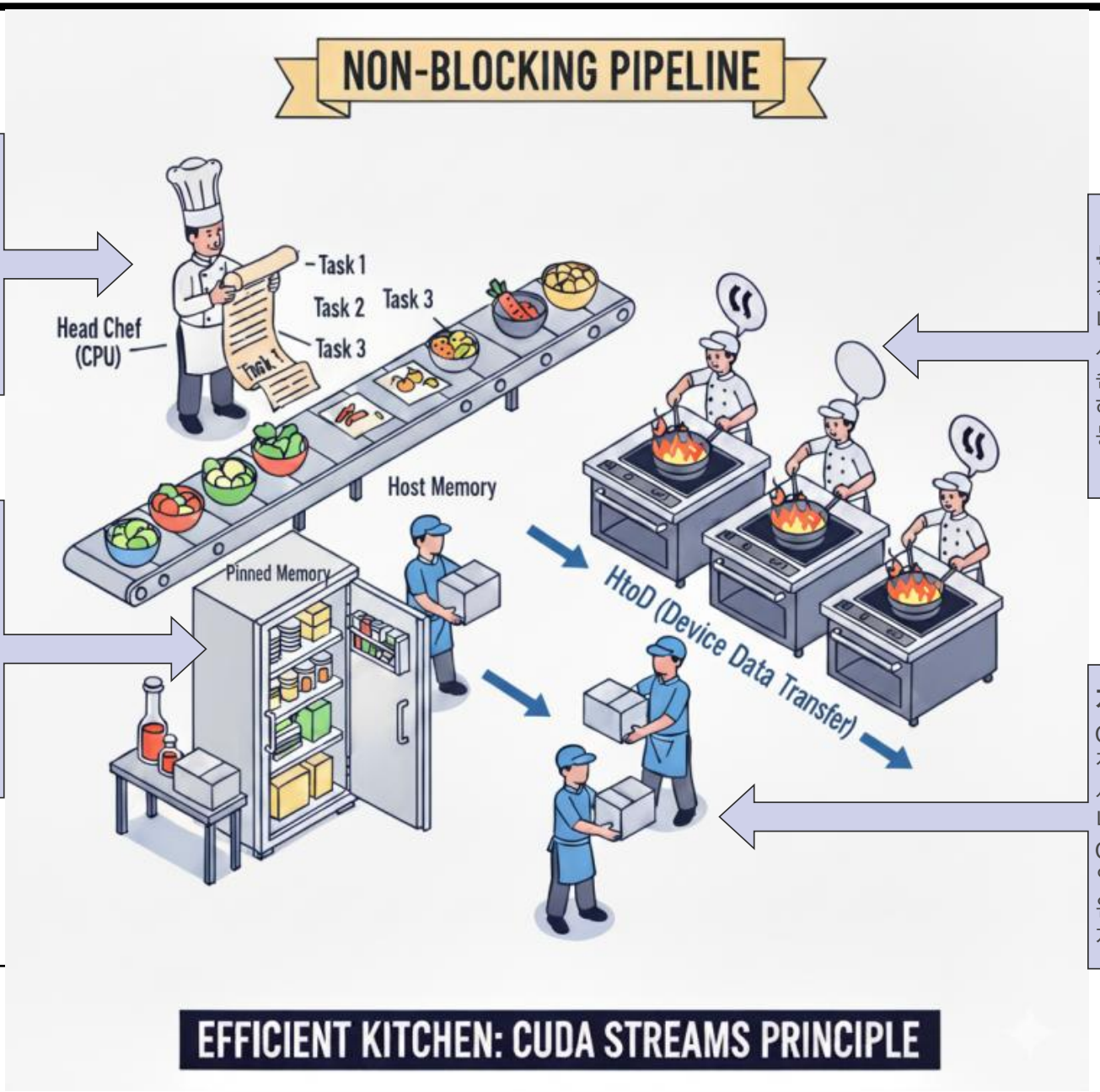
③ C: A결과와 B결과 모두 필요하기에 Wait호출, 이벤트 만족 시 진행

하드웨어로 이해하는 CUDA 스트림의 동작원리

4.8 아주 효율적인 실행 - 비동기식

총주방장(CPU)
이 큰 주문서를 처리하기 쉬운 작은 작업 지시서(chunk)로 분할하고, 지시서들을 컨베이어 벨트(for 루프)에 실어 각 요리 라인(Stream)으로 끊임없이 내려보낸다.

배달 전용 냉장고(Pinned Memory)
배달부(DMA)가 지연 없이 재료를 가져갈 수 있는 특별 구역. cudaMallocHost()로 지정하며, 일반 창고(malloc)처럼 재포장하는 과정이 없어 데이터 병목 현상 원천 차단됨.



독립된 요리 라인(CUDA Streams)
각자의 명령 큐를 가지며, 총주방장이 내린 '재료 운반'(Async Copy)과 '요리 시작'(Kernel Launch) 지시가 순서대로 쌓이면, GPU가 자원이 허락하는 한 여러 라인의 작업을 동시에 처리한다.

자율주행 배달부(DMA Engines)
CPU의 간섭 없이 데이터 운반을 전담. CPU가 "배달 시작!"(cudaMemcpyAsync) 명령만 내리면, 배달부가 운반하는 동안 CPU와 요리사(SMs)는 각자 다른 일을 동시에 진행할 수 있다. 이 완벽한 분업과 협력이 GPU의 처리량을 극대화한다.

4.9 비동기식 구현 -1단계

```
// main() 함수 내부 시작 부분

// 1. 보조 요리사(스트림) 숫자 결정
const int nStreams = 4;

// 2. 보조 요리사들을 고용하고 각자의 요리 라인(스트림 큐)을 만들어 줌
cudaStream_t streams[nStreams];
for (int i = 0; i < nStreams; i++) {
    cudaStreamCreate(&streams[i]);          --각 요리사에게 명령대기열이 있는 요리 스테이션 만들어주기
}                                           --각 스트림들은 병렬로 처리된다

// 3. 배달 전용 냉장고(Pinned Memory) 설치
float *h_a, *h_b, *h_c;
size_t bytes = N * sizeof(float);
cudaMallocHost(&h_a, bytes);                --일반적인 malloc과 달리, 이 공간은 Pinned Memory에 할당됨
cudaMallocHost(&h_b, bytes);                --CPU도움없이 DMA엔진이 직접 이 공간을 사용할 수 있기에,
cudaMallocHost(&h_c, bytes);                --비동기 통신으로 데이터 전송효율 극대화 달성

// ... (데이터 초기화는 잠시 생략) ...
```

4.9 비동기식 구현 -2단계

```
int chunkSize = N / nStreams;
for (int i = 0; i < nStreams; ++i) {
    int streamIdx = i;
    int offset = i * chunkSize;

cudaMemcpyAsync(d_a + offset, h_a + offset,
    chunkSize * sizeof(float), cudaMemcpyHostToDevice,
    streams[streamIdx]);

vectorAdd<<<gridSize, blockSize, 0, streams[streamIdx]>>>(
     d_a + offset, d_b + offset, d_c + offset, chunkSize);


cudaMemcpyAsync(h_c + offset, d_c + offset,
    chunkSize * sizeof(float), cudaMemcpyDeviceToHost,
    streams[streamIdx]);
}
```

--4. for 루프를 돌며 작업을 chunk 단위로 분할, 스트림에 할당
-- 현재 작업할 스트림과 데이터 위치 계산

--[명령 1] 재료 운반 지시 (Host → Device, 비동기 복사)
--"배달부! offset 위치의 재료를 streamIdx번 요리사에게 갖다줘!"

-- [명령 2] 요리 시작 지시 (커널 실행)
-- "streamIdx번 요리사! 재료 도착하면 바로 요리 시작해!"

-- [명령 3] 완성된 음식 회수 지시 (Device → Host, 비동기 복사)
-- "배달부! streamIdx번 요리사가 요리 끝내면 음식 가져와!"

4.9 비동기식 구현 -3단계

cudaDeviceSynchronize();

--5. 주방 마감: 모든 요리사(스트림)의 작업이 끝날 때까지 대기
-- "모든 요리 다 끝나고 주방 정리될 때까지 퇴근 금지!"
-- 이 함수는 Blocking함수 임. CPU는 실행을 멈추고 대기

// 모든 작업이 완료되었으므로, 이제 결과를 확인하거나 메모리를 해제할 수 있음
// ... (결과 검증 및 메모리 해제 코드) ...

4.10 아주 비효율적인 실행 – 동기식 (복습용)

4.HOST CPU: 75%이상
놀고있는 중

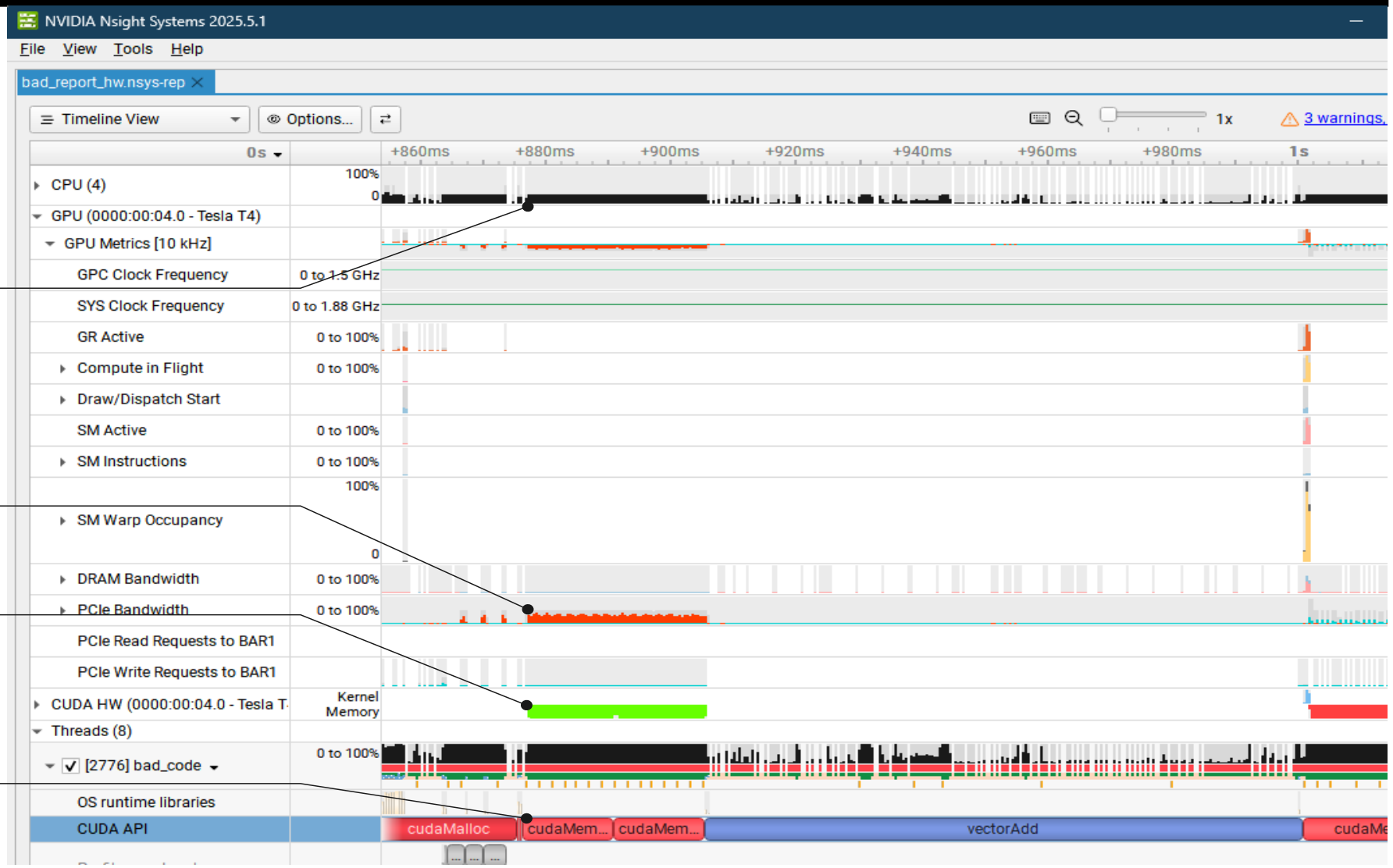
**PCI통신버스 여유있고,
CUDA HW또한 여유있는
상황인데 왜 CPU는 더 많은
데이터를 GPU에게
전송하지 못하는가?**
(Pinned Memory)

3.HOST-GPU통신버스
사용량 역시 60%이상
놀고있음

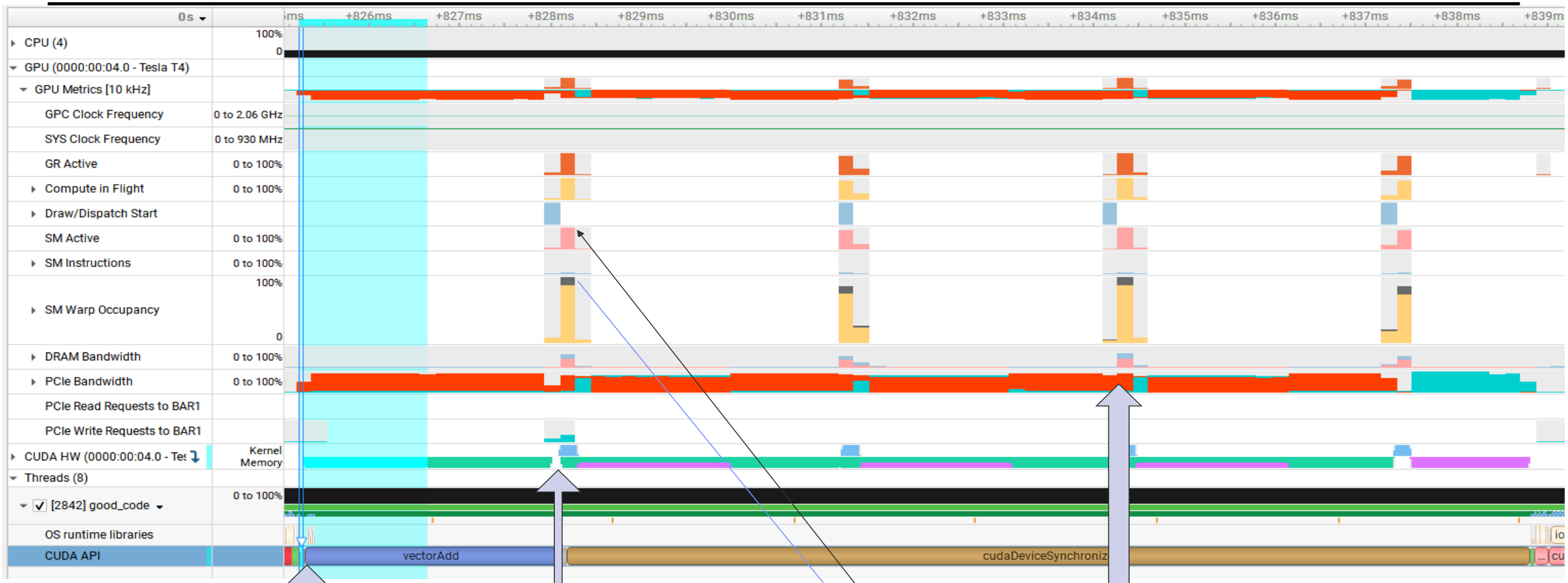
2.HOST에서 도착하는
데이터 전송율 높지 않아서,
CUDA메모리 시스템
50%이상 놀고있음

1.인간의 일처리 방식 처럼,
한번에 하나씩
순서적으로 API호출
(동기적처리)

**cudaMemcpy동작을
동시에 진행할 수 없을까?**
(auto copy by DMA)



4.11 아주 효율적인 실행 – 비동기식



CPU의 역할: Fire-and-Forget
CPU는 모든 작업 지시(커널 호출)를 순식간에 GPU에 위임하고, 자신은 긴 동기화 대기 상태로 들어갑니다. CPU가 더 이상 병목이 아님을 증명합니다.

GPU의 실제 작업: 계단식 파이프라인
GPU 하드웨어에서는 메모리 복사(HtoD/DtoH)와 커널 실행이 서로 겹쳐서 계단식으로 실행됩니다. 연산과 데이터 전송이 동시에 처리되는 완벽한 오버랩입니다.

GPU 효율: 100% 가동률
파이프라인 덕분에 SM의 활성도와 점유율(노랑)이 작업 내내 높은 수준으로 유지됩니다. GPU의 비싼 연산 자원이 낭비 없이 최대한으로 사용되고 있음을 보여줍니다.

데이터 흐름: 끊김 없는 파이프라인

PCIe 대역폭(빨강)이 bad_code처럼 중간에 끊기지 않고 꾸준히 활용됩니다. GPU가 굶주리지 않도록 데이터가 계속 공급되고 있음을 보여줍니다.