GPU요리하기_하드웨어전문가의 CUDA 개발자 가이드

2강. 숨은 병목 찾기: 데이터 전송과 커널실행

- GPU 성능 최적화의 첫 걸음 -

목표:
간단한 벡터 덧셈 예제를 통해,
GPU 컴퓨팅의 숨겨진 병목인 ' 데이터 전송 시간 ' 의 중요성을 이해하고,
프로파일링 툴로 확인하는 방법을 학습한다.

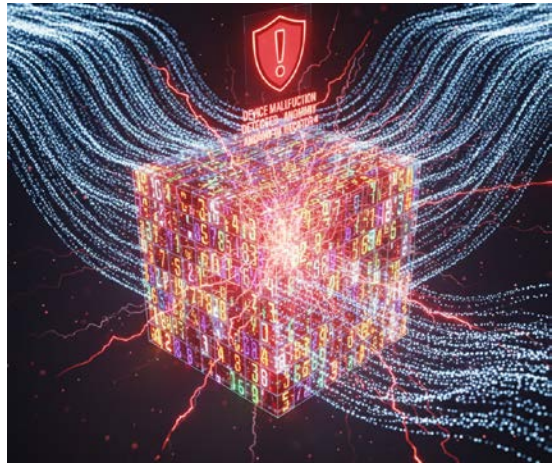
자료제공:NVIDIA

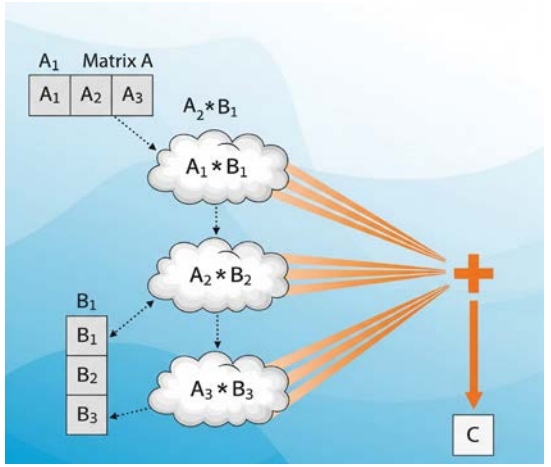
2.1 돌아보기

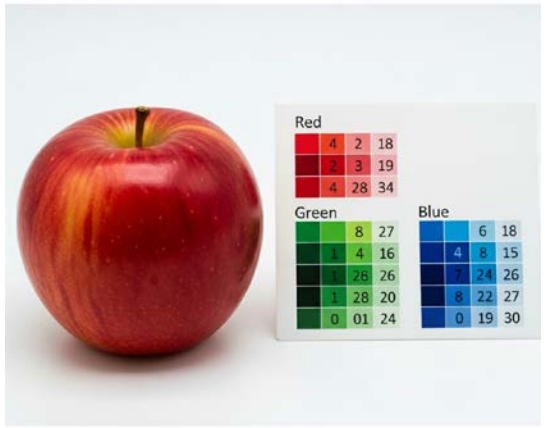
- 2차원 정적 행렬
- 3차원 동적 행렬
- 행렬 연산에 특화된 GPU

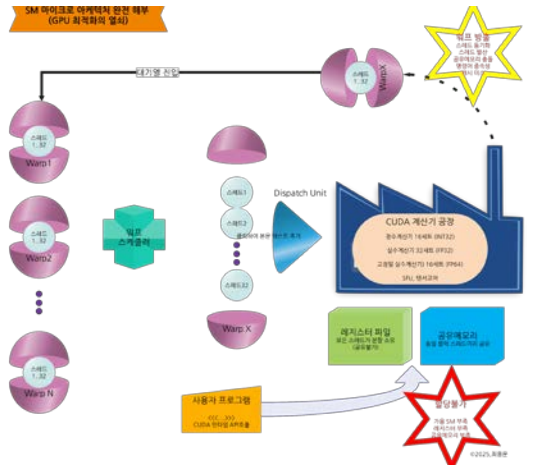
행렬은 컴퓨터가 다루는 기본 형태이며,

GPU는 행렬 병렬처리에 강력하다

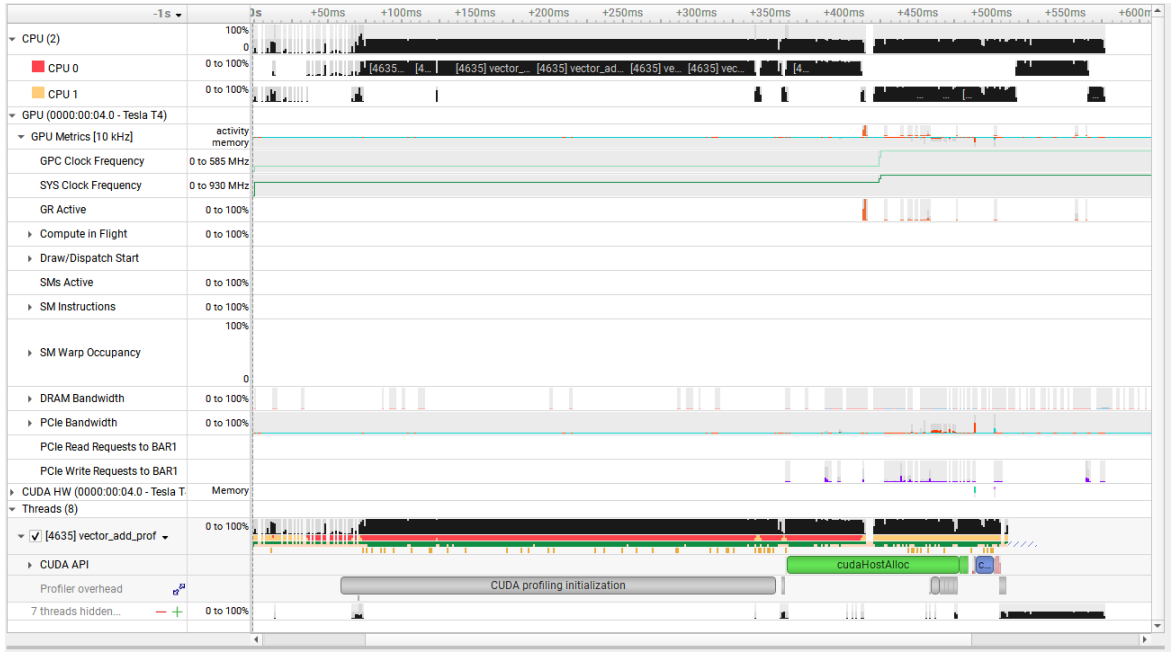








2.2 오늘의 요리: 데이터 전송과 숨겨진 병목



이 프로파일링 화면은 NVIDIA Nsight Systems을 사용하여 생성되었습니다

간단한 요리

- 벡터 덧셈 살펴보기
- 재료 (데이터)를 주방(GPU)으로 옮기는 과정 살펴보기
- 핵심: 데이터 옮기기의 중요성을 인식해야 한다!

CCTV 살펴보기 (프로파일링)

- Nsight Systems: 감시카메라 처럼 GPU에서 벡터 덧셈 실행 과정을 꼼꼼히 기록해준다
- 실행 프로파일을 살펴보면, 숨겨진 병목지점을 직접 확인할 수 있다.

2.3 CPU와 GPU (호스트와 디바이스)

호스트

CPU, 메인 메모리 (DDR)

전체 시스템을 제어

데이터의 주인

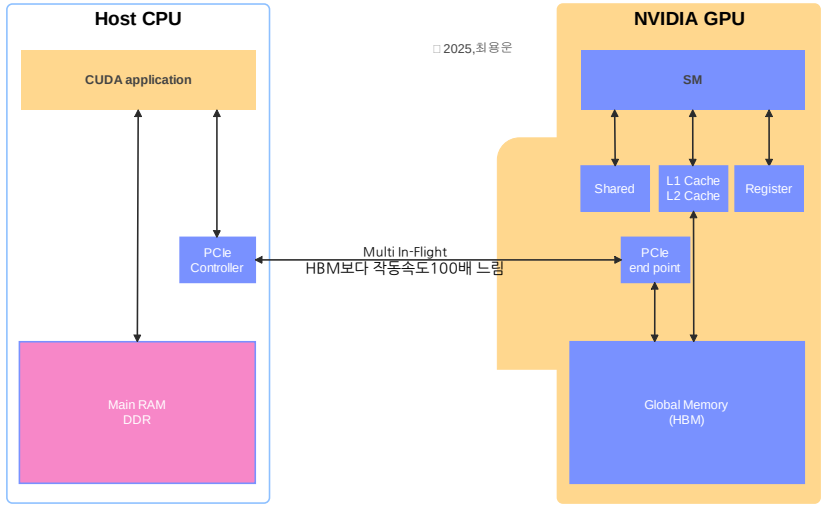
디바이스

GPU, 비디오 메모리 (VRAM, GDDR)

병렬연산에 특화, 데이터를 가지고 연산하는 일꾼

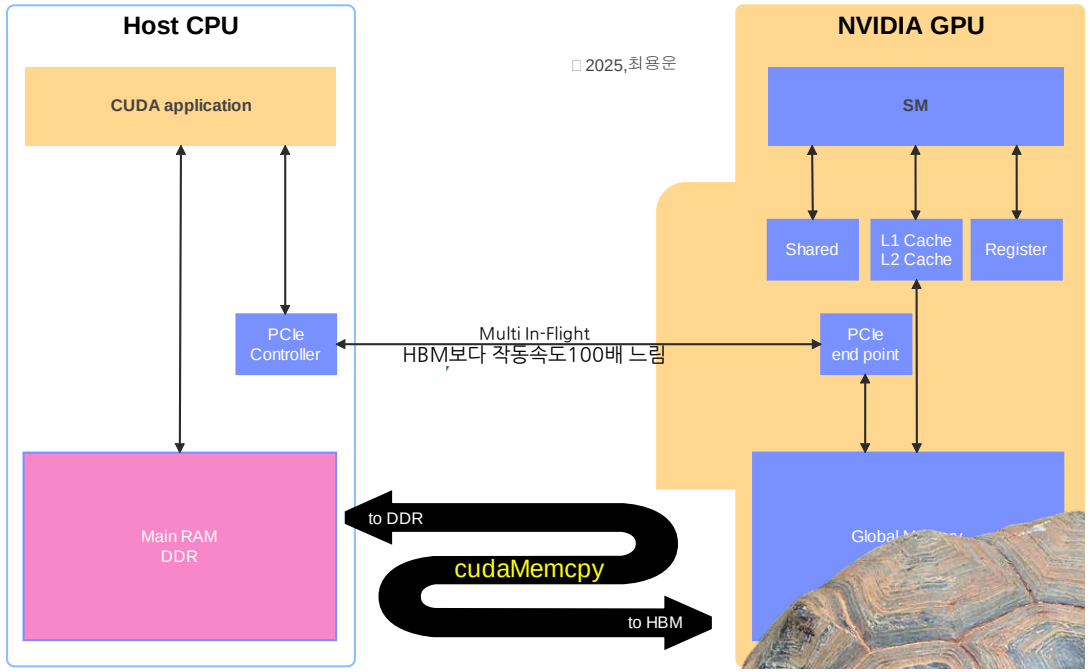
핵심!

호스트와 디바이스는 서로 다른 메모리 공간을 사용한다



이 프로파일링 화면은 NVIDIA Nsight Systems을 사용하여 생성되었습니다

2.4 데이터 전송의 이해 cudaMemcpy=주방으로 재료 옮기기



cudaMemcpy

- 호스트 메모리 → 디바이스 메모리로 복사
- 디바이스 메모리 → 호스트 메모리로 복사

왜 데이터를 전송?

- CPU와 GPU는 직접 메모리 공유 불가 (연결 안됨)
- GPU 연산에 필요한 데이터는 사전에 GPU 메모리로 복사 필요

데이터 전송 비용 발생

- 느린 PCIe 버스 통신 (HBM보다 100배 느림)
- 최적화 방안이 필요: GPU 연산 실행 + 백그라운드 호스트 전송

2.5 커널 실행

“병렬함수명<<<gridDim, blockDim>>>(,,,)”

HOST가 GPU에게 작업을 지시한다

프로그램머가 지시한 구조 정보(gridDim, blockDim)

이 정보는 절대 변경 불가 (정적 계약), 프로그래머와의 약속

전역 스케쥴러는 동적으로 블럭들을 배정한다

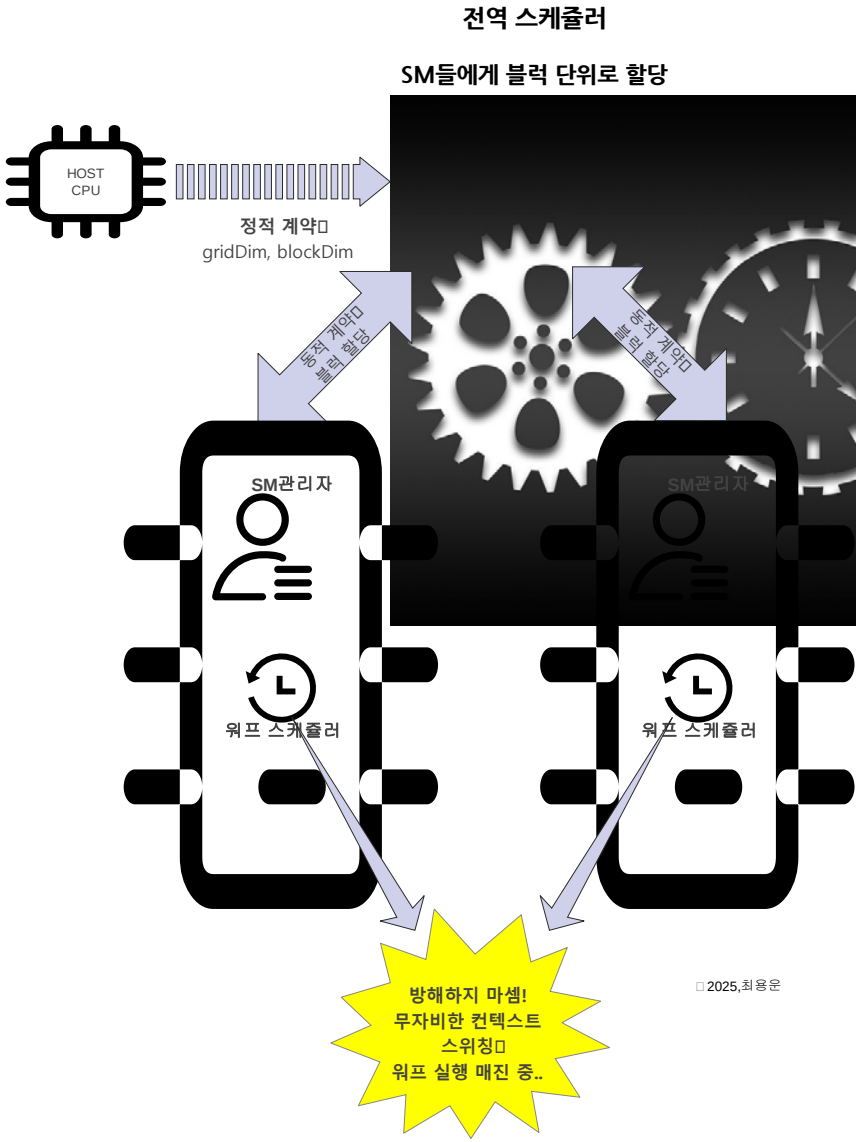
그리드 안에 있는 블럭들을 SM들에게 배정하고,

개별 SM 상황 따라 추가 배정을 실시한다

워프 스케쥴러

오직 워프(32개 스레드)들의 실행상황 감시와 스위칭 작업 매진

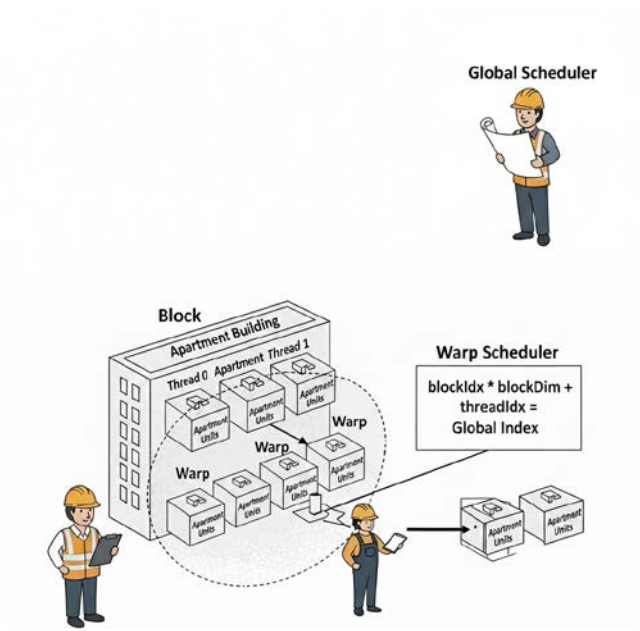
블럭단위 스케쥴링에 참견할 여유가 없음



2.5 커널 실행

논리적 스레드와 물리적 메모리의 만남

index = blockIdx.x * blockDim.x + threadIdx.x;



지역성 메모리 접근

지역성 메모리 접근? 스레드들이 인접한 메모리에 접근하는 것

메모리 접근 합칩으로 지대한 성능향상 발생한다.

혼동스런 스레드 인덱스!

vectorAdd <<<32,64>>> : 건물 32동, 동별 64 가구 입주

blockDim: 64세대, 프로그래머가 결정(하드웨어 최적화 관점)

gridDim: 32동, 계산값 = 전체세대수(32*64) / blockDim

blockIdx: 건축설계도의 동 번호 (0..31)

threadIdx: 건축설계도의 가구번호 (0..63)

전역 스케줄러: 총감독, 작업팀(SM)에게 특정 동(blockIdx) 작업을 지시

워프 스케줄러: 작업분배기, 32명 단위(워프)로 작업을 묶어 투입

Index = blockIdx.x * blockDim.x + threadIdx.x

스레드의 사명: “나는 목숨을 다하여 내 코드만 실행한다”, 다른 스레드? 묻지마셈!”

blockIdx, threadIdx 결정 주체는?

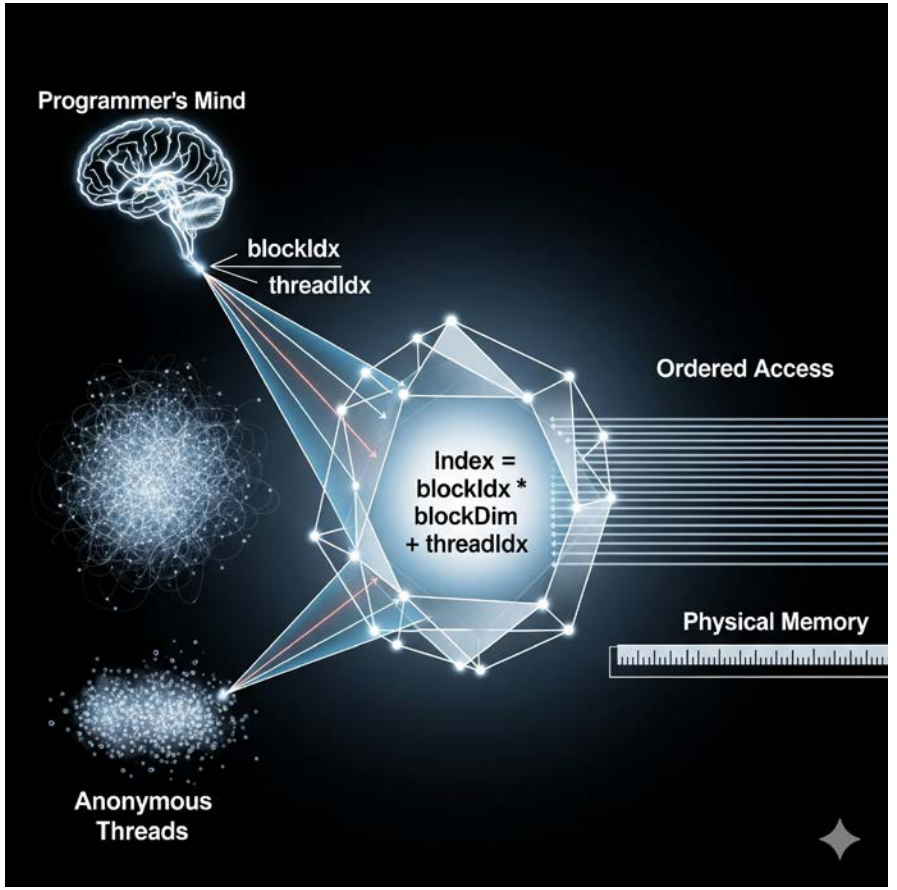
<<<32,64>> 문장따라 호출되는 쿠다 런타임 API가 결정

이 정보(blockIdx,threadIdx)는 마치 출석부에 등재된 기준정보이며, 실행 중 변경없음

2.5 커널 실행

한 줄의 공식, 3가지 철학적 의미
index = blockIdx.x * blockDim.x + threadIdx.x;

1. 정체성의 선언 혼돈에서 질서로	2. 구현의 다리 논리에서 물리로	3. 추상화의 약속 복잡함에서 단순함으로
• 이 코드가 실행되기전 수 많은 스레드들은 모두 익명의 존재임. 광장에 모인 군중 • 이 한 줄 코드 실행으로 모든 스레드는 자신만의 고유한 이름표(index) 부여 받음 • 이로써 각 스레드는 “나는 누구인가” 질문에 답할 수 있음	• 프로그래머는 논리적이고 추상적인 공간(blockIdx, threadIdx)에서 문제를 설계한다 • 그러나, 데이터는 물리적이고 선형적인 공간(메모리)에 존재 • 위 공식은 추상적인(논리) 설계와 실제 데이터(물리)를 연결하는 다리 역할임 • NVIDIA GPU전반을 아우르는 데이터 중심철학을 실제 하드웨어로 구현하는 행위	• 실제 GPU하드웨어는 매우 복잡 (다수의 SM, 워프 스케줄러, 메모리 컨트롤러등) • 프로그래머는 위 공식만 사용하면 됨 (나머지 복잡성은 잊어라! 그리고 프로그래밍 문제에만 집중하시라!) • 위 공식은 개발자와 하드웨어 사이의 신성한 약속이다. 개발자님이 문제의 구조를 논리적으로 정의해준다면, 하드웨어는 그 구조를 해석하여 수백만개 작업을 효율적으로 처리하겠습니다.



태초에 하나님이 천지를 창조하시니라, 땅이 혼돈하고 공허하며 흑암이 깊음 위에 있고, 하나님이 이르시되 빛이 있으라 하시니 빛이 있었고, 빛이 하나님이 보시기에 좋았더라

2.5 커널 실행

C++

```
// GPU에서 실행될 커널 함수
__global__ void vectorAdd(float* a, float* b, float* c, int n) {
    // "이것이 바로 저희가 논의했던 그 공식입니다."
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < n) {
        // "그리고 이것이 스레드가 수행하는 핵심 작업입니다."
        c[index] = a[index] + b[index];
    }
}
```

2.6 프로파일링 삽질하기

COLAB 작업절차

1. 코드 작성 및 저장

우측 그림과 같이, 저장명령과 함께 프로그램 소스 코드를 입력하고
PLAY 삼각형을 클릭하면 저장명령이 실행되며,
현제 접속된 GPU서버의 content 디렉토리에 저장된다

```
%%writefile vector_add_profile.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib> // exit() 사용을 위해 추가

// ============================================================
// 모든 CUDA API 호출을 위한 에러 체크 매크로
#define CUDA_CHECK(err) { \
    cudaError_t err_code = err; \
    if (err_code != cudaSuccess) { \
```

2. 컴파일

!nvcc -arch=compute_80 -gencode=arch=compute_80,code=sm_80 vector_add_profile.cu -o vector_add_profile

3. 프로파일링 실행

NVIDIA 프로파일링 프로그램 (Nsight Systems) 실행파일 명: nsys

!/opt/nvidia/nsight-compute/2024.2.1/host/target-linux-x64/nsys profile --trace=cuda --gpu-metrics-device 0 -o vector_add_profile.nsys-rep ./vector_add_profile

실행파일 “nsys” 위치가 서버마다 다를 수 있음 (T4-GPU 연결 후, 위 명령으로 실행하였음)

Nsys에게 다양한 옵션을 명시하여 프로파일링 가능

프로파이링 분석결과를 별도 파일에 저장 (vector_add_profile.nsys-rep)

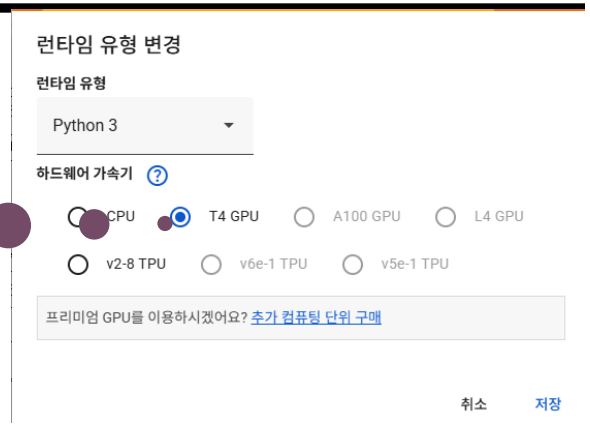
4. 프로파일링 결과파일 PC로 다운받기

우측 그림처럼 코랩에 파이선 코드 입력하고, PLAY버튼 클릭
잠시후 PC에 저장하기 대화상자가 표시되고, 위치 지정하여 SAVE

```
[ ]  from google.colab import files
     files.download('/content/vector_add_profile.nsys-rep')
```

2.6 프로파일링 삽질하기

CoLab에선 GPU/TPU 서버를 선택,연결합니다.



프로파일링: 실행오류 문제

1. 코드 작성 및 저장

   작업 절차 따르면 어렵지 않음

2. 실행 오류발생: “CUDA Error: the provided PTX was compiled with an unsupported toolchain”

   대책: 컴파일시, 연결된 GPU종류 따라 컴파일 명령옵션을 달리 지정해야 한다

   !nvcc –arch=compute_80 –gencode=arch=compute_80,code=sm_80 vector_add_profile.cu –o vector_add_profile

   예를들어 T4-GPU에 연결된 경우 컴파일 명령은

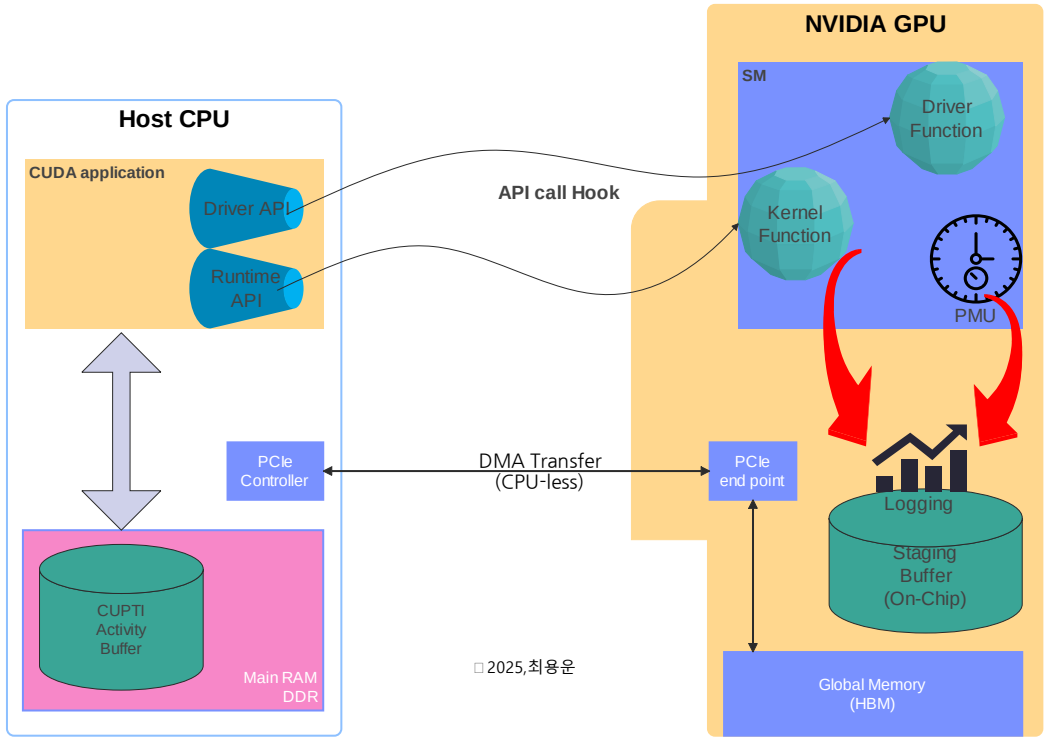
   !nvcc –arch=compute_75 –gencode=arch=compute_75,code=sm_75 vector_add_profile.cu –o vector_add_profile

3. 프로파일링 실행

   작업 절차 따르면 어렵지 않음

2.7 프로파일링

-프로파일링 작동 원리-



1. 소트프웨어 작동 모니터링 원리

- NVIDIA의 모든 런타임 라이브러리는 외부 프로그램(예: Nsight Systems)이 구독신청할 수 있는 기능을 구비하고 있음 (콜백 등록)
- 라이브러리(런타임,저수준 드라이버 API)는 작동시점에 등록된 구독자들을 호출해주고, 외부프로그램은 그 로그정보와 타임스템프를 기록한다
- NVIDIA는 이런 절차를 공식API(CUPTI: CUDA Profiling Tools Interface)로 100% 공개, 누구나 소프트웨어 작동을 모니터링 할 수 있음

2. 하드웨어 작동 모니터링 원리

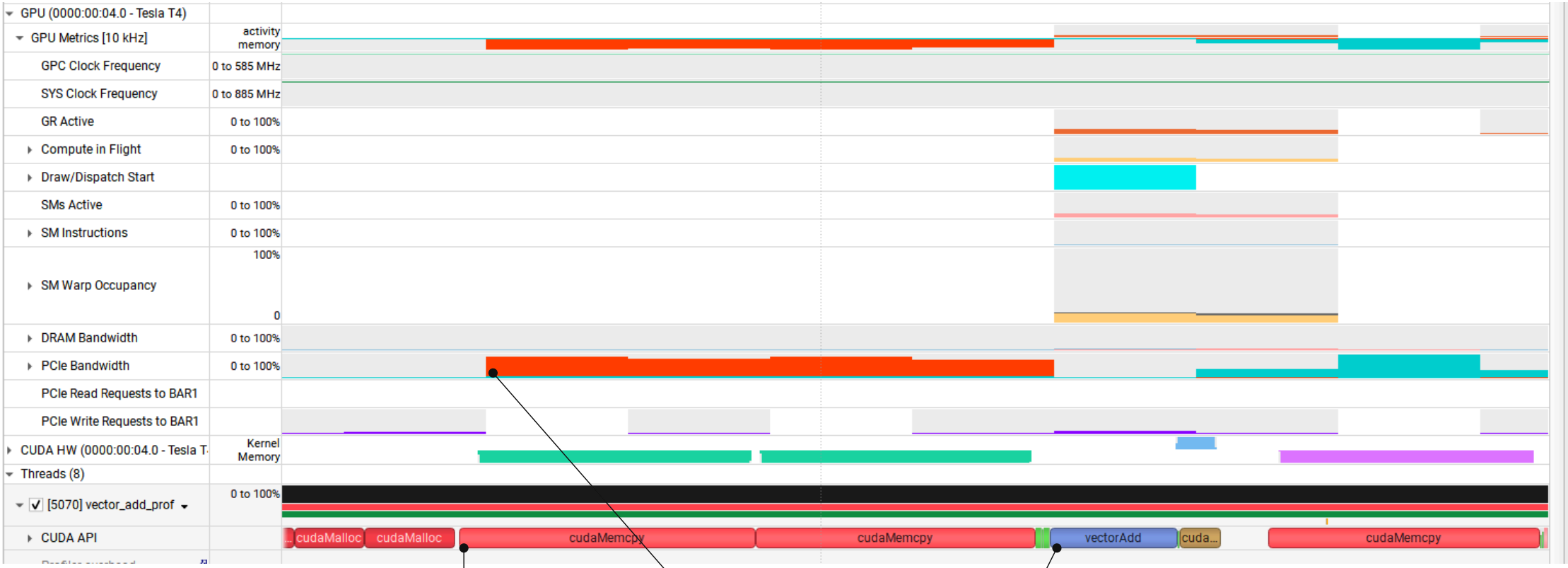
- GPU 하드웨어에서 발생하는 이벤트는 나노초 단위로 대량 발생하며,이를 상대적으로 매우 느린 콜백 호출 방식으로 CPU에게 전달하는 것은 불가능
- NVIDIA는 GPU 내부에 하드웨어 성능측정장치(PMU)가 내장해 두었으며,이는 외부 요청에따라 작동한다.
- GPU가 실행을 마치면, 외부 프로그램(Nsight)은 PMU의 기록을 조회하여 하드웨어 작동 상태를 파악한다 (CUPTI에 공개됨)
- PMU가 하드웨어 작동지표 계산에 사용한 데이터와 알고리즘은 비공개 (GPU기술유출 방지)

3. 호스트에게 정보 전달 및 표시

- 빠르게 생성되는 로그정보(소프트웨어,하드웨어)는 CPU 개입없이 DMA장치가 호스트의 특정 메모리 구역에 블럭단위로 저장한다
- 외부 프로그램(Nsight Systems등)은 이 정보를 CUPTI규약에 따라 해석하여 화면에 표시한다

2.7 프로파일링
VECTORADD

이 프로파일링 화면은 NVIDIA Nsight Systems을 사용하여 생성되었습니다



1.호스트에서 준비한 데이터를 GPU로 전송 시작

2.호스트전송 시작과 동시에 PCIe대역폭 사용한계에 도달

3.데이터준비 완료후 병렬실행 시작