PDC CUDA 스터디
- Warp Level CUDA and Atomics -

2020-12-13

Constant Park (박상수)

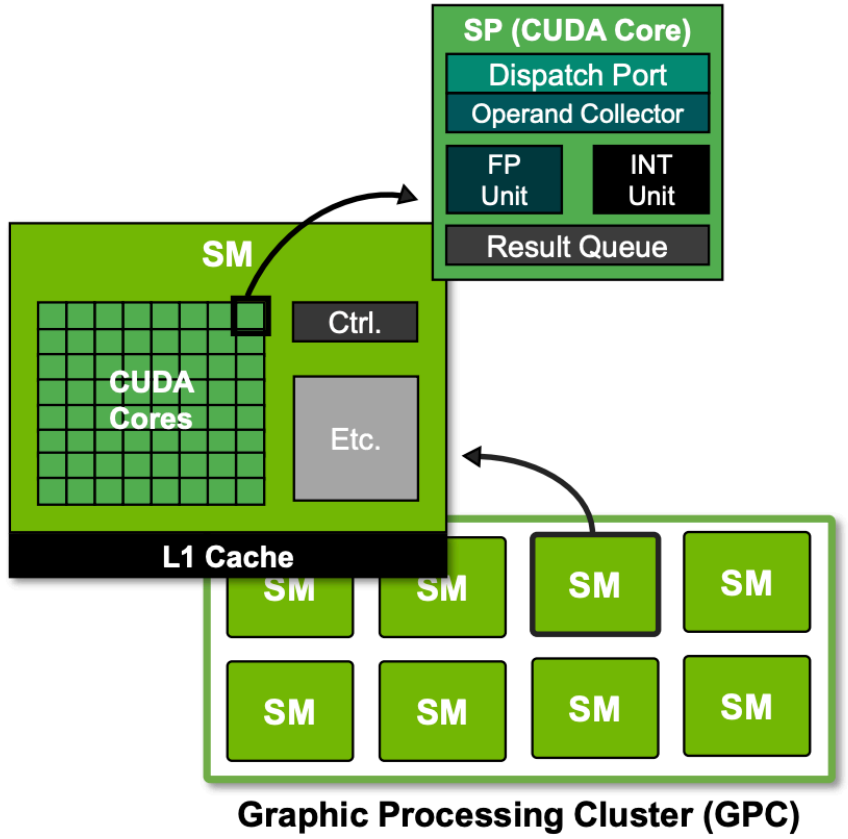
Parallel Development Community (PDC)

CUDA 하드웨어 구조 (NVIDIA's GPU)

- CUDA Core (Streaming Processor, SP)
  - GPU구조에서 가장 기본적인 연산 단위
  - 멀티코어 CPU에서 하나의 코어에 해당

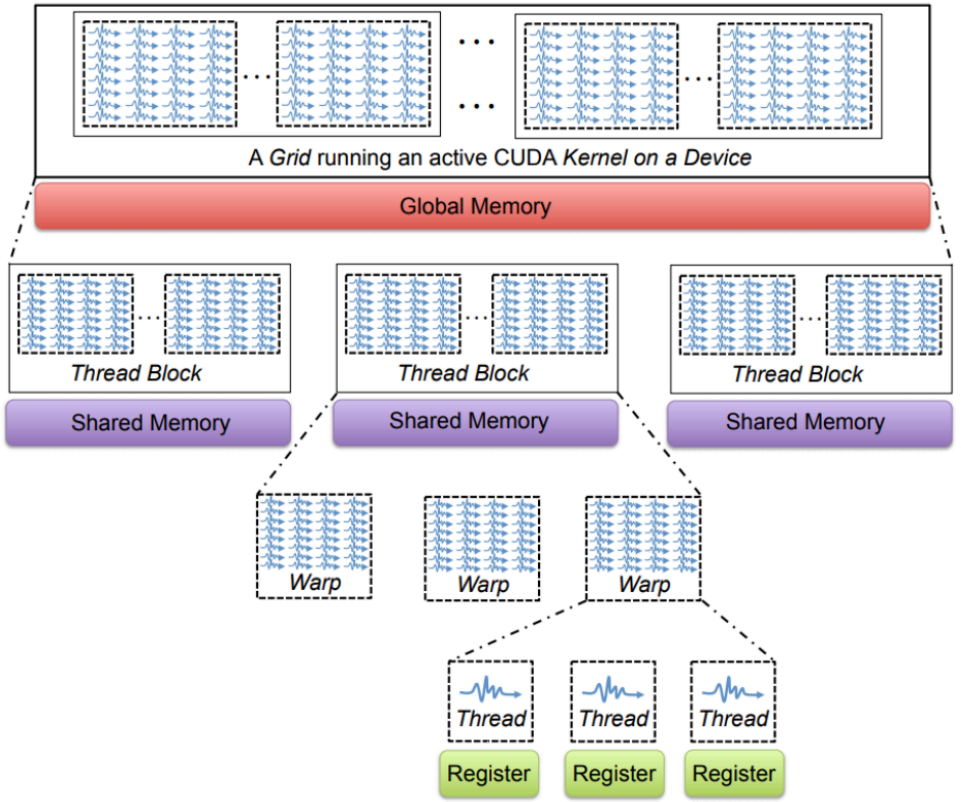
- SM (Streaming Multiprocessor)
  - 여러 개의 SP와 L1$ 포함
  - SP 여러 개는 하나의 컨트롤 로직을 공유

- GPC (Graphic Processing Cluster)
  - SM의 그룹



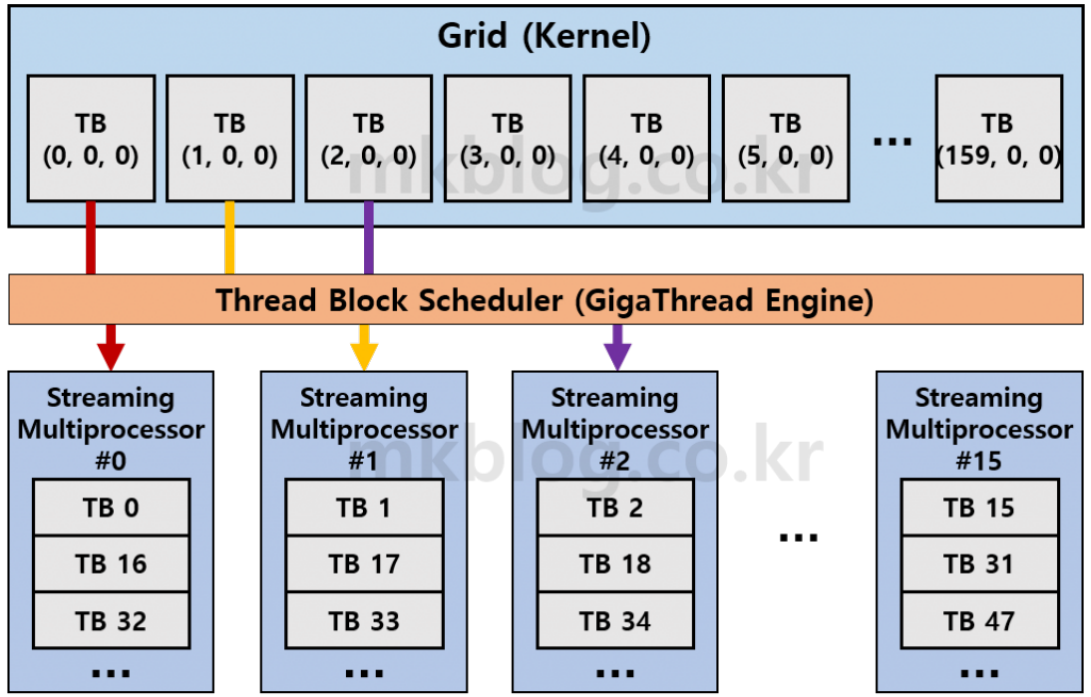
CUDA 프로그래밍 모델

- Thread
  - 멀티코어에서 Thread와 동일한 개념
  - 하나의 Thread는 하나의 SP에 동작
- Thread Block
  - 여러 개의 Thread가 묶인 단위
  - Warp 단위로 나눠져 SM에서 동작
- Warp
  - 하드웨어에서 Thread를 처리하는 단위
  - 보통 32개의 스레드가 묶여서 처리
    (AMD는 64개의 work-item, Wavefront)



Thread Block Scheduling

- Thread Block Scheduling[1]
  - Kernel이 시작된 이후 시작
  - Scheduler는 SM 성능을 고려하여 TB 할당
    -실행 가능한 Thread, Register 개수
    - Scratchpad Memory 크기
  - Thread는 하나 이상의 Warp에 매핑

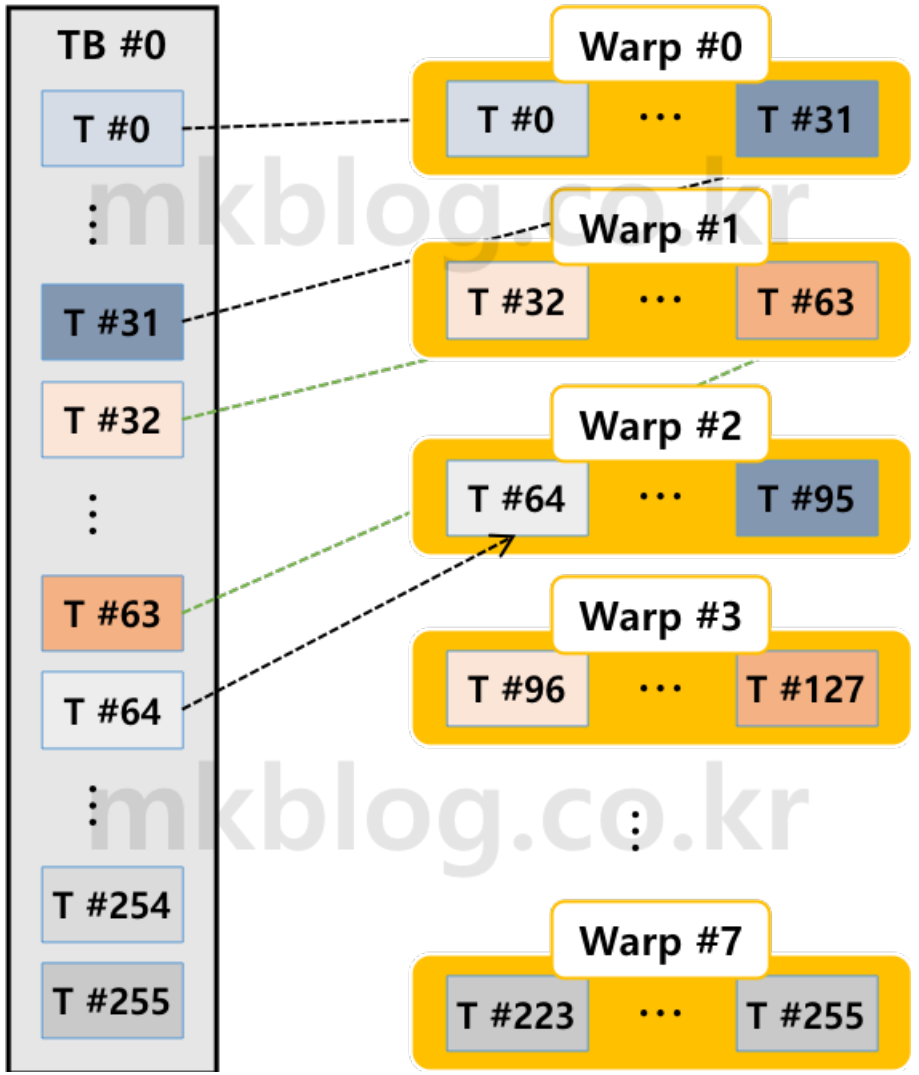


[1] https://mkblog.co.kr/2018/10/13/gpgpu-series-5-scheduling-thread-blocks/

Warp Scheduling

- GPU 명령어는 Warp 단위로 실행
  - Warp 내부에서는 SIMD (SIMT) 연산 수행
    - 하드웨어 (SIMD), Warp (SIMT)
    - 조건에 해당되는 코드가 실행된다면 ?

      (Branch Divergence)



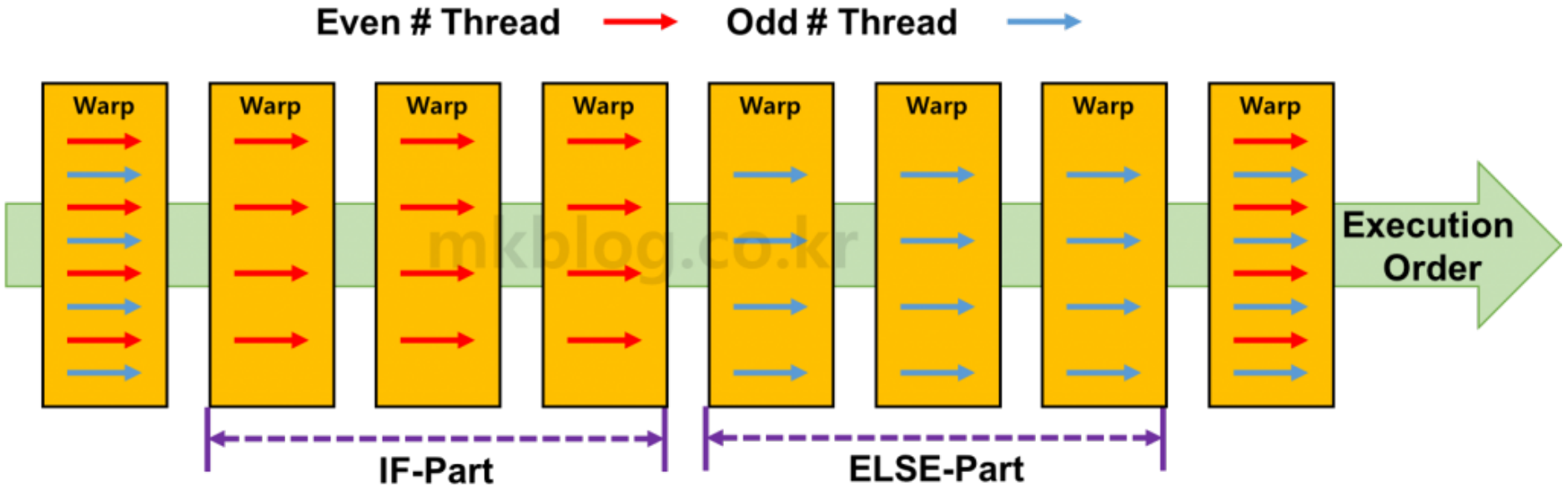
Branch Divergence #1

- GPU 명령어는 Warp 단위로 실행
  - Warp에 속한 짝수 T (IF), 홀수 (ELSE)
    - 하드웨어 (SIMD), Warp (SIMT)
    - 다수의 분기가 발생한다면, 순차적으로 실행

```
__global__ void mkTest(){
        int threadID = threadIdx.x
        if((threadIdx.x % 2) == 0){
                짝수 Thread 코드 ...
        }
        else{
                홀수 Thread 코드 ...
        }
}
```

Branch Divergence #2

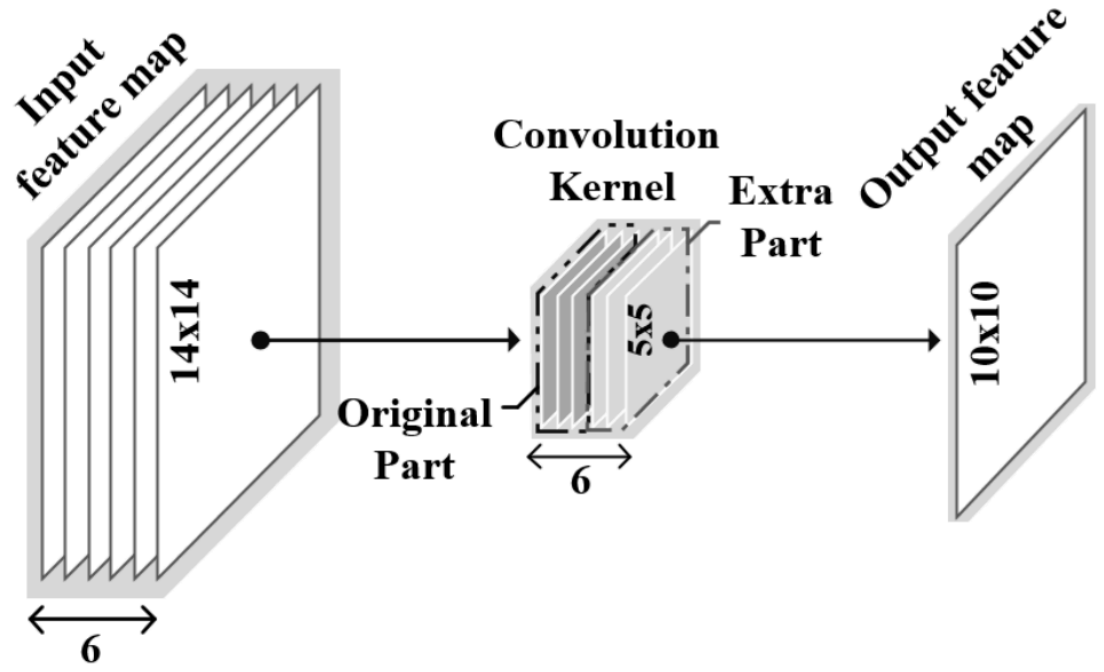
- GPU 명령어는 Warp 단위로 실행
  - Warp에 속한 짝수 T (IF), 홀수 (ELSE)
    - 하드웨어 (SIMD), Warp (SIMT)
    - 다수의 분기가 발생한다면, 순차적으로 실행
    - Ways (Branch의 개수)



Branch Divergence in CNN

- Pruning Layer
  - Convolution Layer 2 in LeNet-5
    - Feature Map의 일부분 연산에 사용 (Pruning)
    - Compare와 Branch에 해당되는 명령어 필요

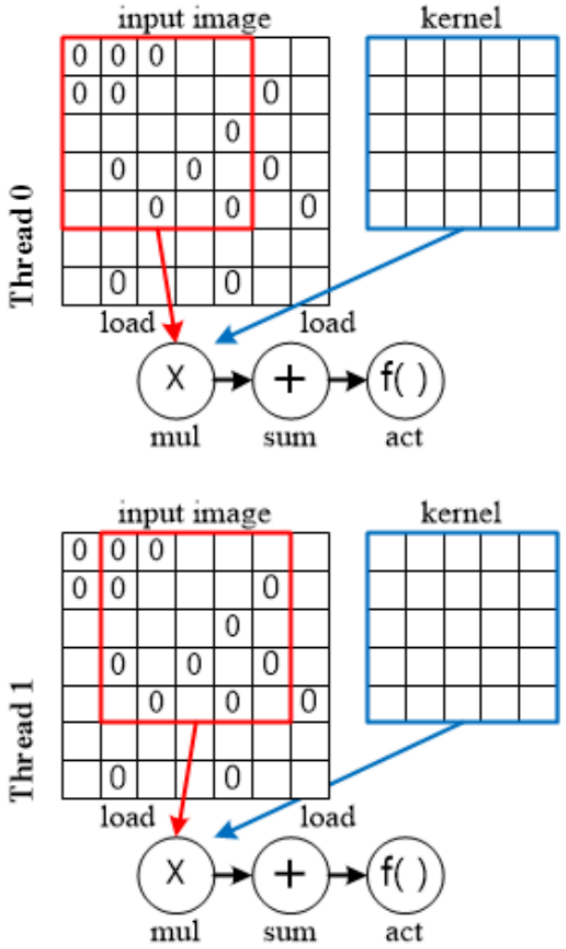
Input feature index	Output feature index															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X



[2] Modified Convolution Neural Network for Highly Effective Parallel Processing, *IEEE IRI*.

Branch Divergence를 극복하는 방법

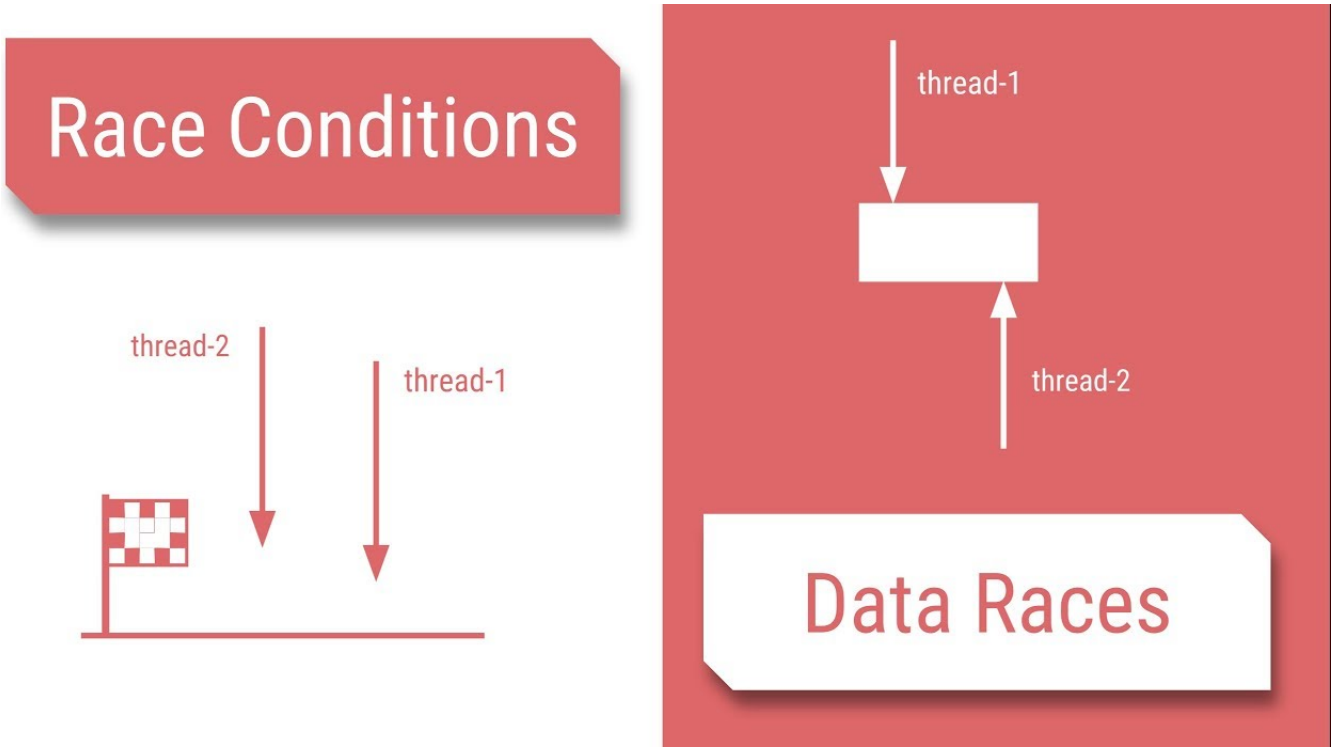
- NOP 명령어 사용
  - No Operation (NOP)를 사용
  - Branch가 발생하는 부분에 NOP 명령어 적용
  - Prediction을 사용하는 것도 있으나
    - 명령어의 숫자를 줄이는 장점, GPU Utilization 감소



[2] Implementation of MNIST classification CNN with zero-skipping.

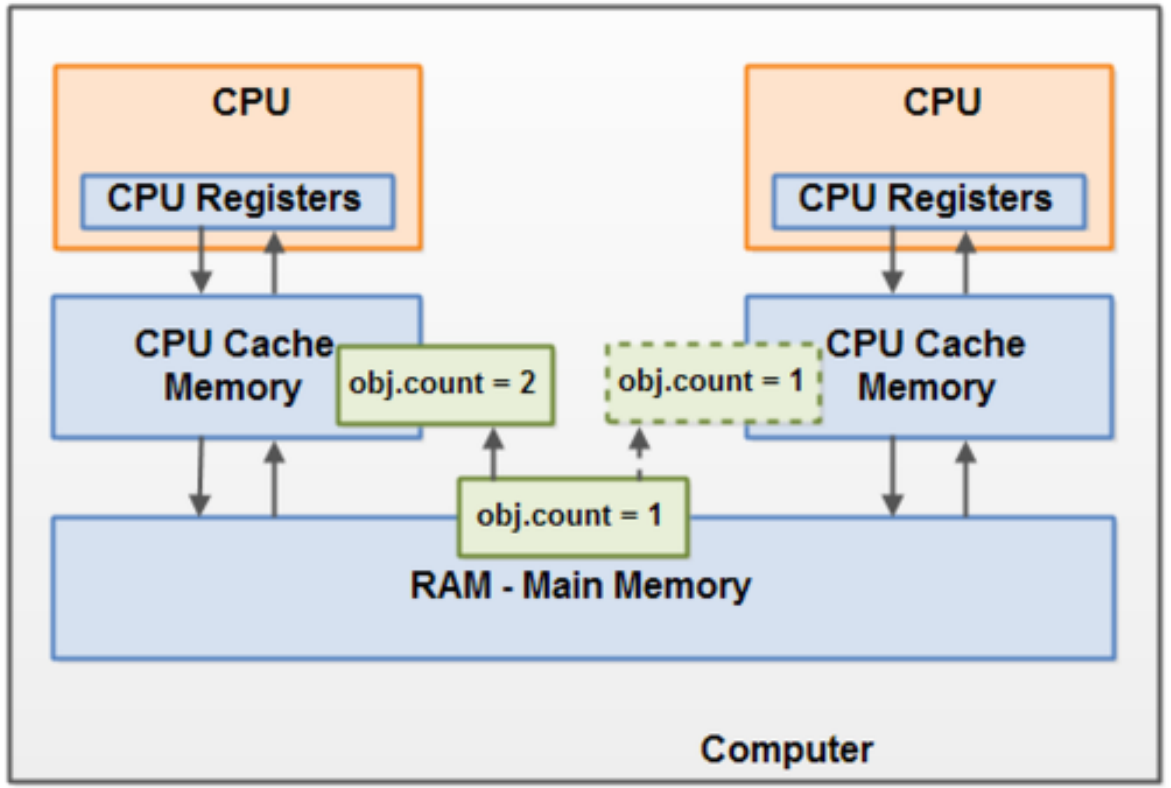
Atomic #1

- Race Condition을 해결하기 위한 방법
  - 다수의 Thread가 하나의 변수에 접근 하는 경우 (Min/Max 연산)
  - 병렬 처리를 위한 하드웨어에서 지원하는 기능 (중복된 메모리 사용)



Atomic #2

- Atomic Function 연산은 다른 Thread에 의해 방해를 받지 않아야 함
  - 다수의 Thread가 하나의 변수에 접근 하는 경우 (Min/Max 연산)
  - 임계 영역 (Critical Section): Lock이 걸려 하나의 Thread만 활동 가능 (Serialization)



Atomic #3

- CUDA에서 제공하는 Atomic 함수들
  - 여러 Thread에서 공유 메모리 변수에 동시에 읽고 쓰는 것을 방지 (성능 감소)
  - Parameter로 저장된 주소와 새로운 Value 값을 받아서 계산, 해당 주소에 Return
  - 다수의 Thread가 하나의 변수에 접근 하는 경우 (Min/Max 연산)
  - `int atomicAdd(int* address, int val)`

Name	Function
atomicAdd	old+val
atomicSub	old-val
atomicExch	swap (old.val)
atomicMin	Min (old, val)
atomicMax	Max (old, val)
atomicInc	(old >= val) ? 0 : old+1

Atomic #4

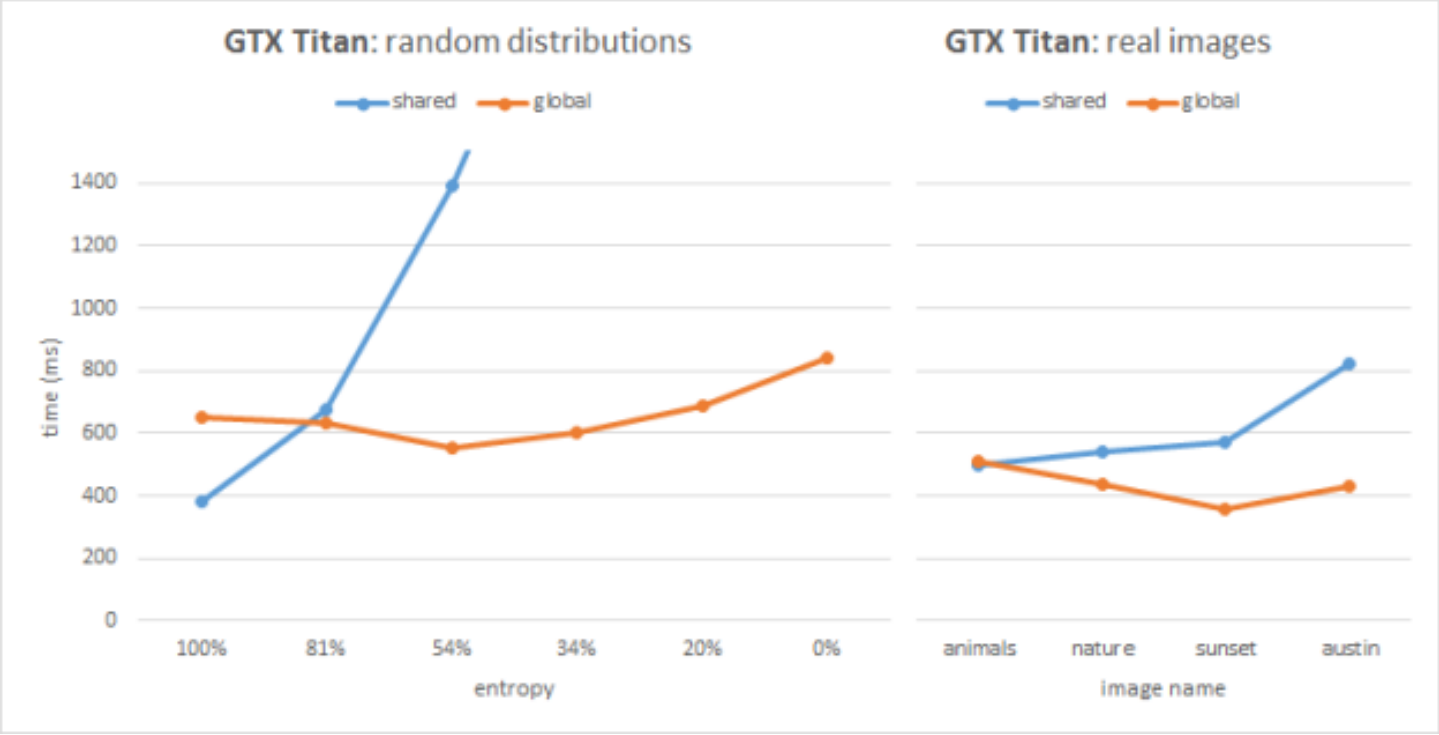
- CUDA에서의 Atomic 예제

```
__global__ void CUDA_Sum(int* dst, int * src)
{
     int idx = threadIdx.x;
     dst[0] += src[idx];// 모든 스레드가 dst[0]에 순서없이 마구 접근합니다.
     printf("src[%d]=%d, dst[0]=%d\n", idx, src[idx], dst[0]);
}


__global__ void CUDA_Atomic_Sum(int* dst, int * src)
{
     int idx = threadIdx.x;
     int out = atomicAdd(&dst[0], src[idx]); // 모든 스레드가 한줄로 서서로 차례대로 dst[0]에 접근합니다.
     //모든 스레드가 위의 작업이 끝날때까지 대기합니다.
     printf("src[%d]=%d, dst[0]=%d atomicOut=%d \n", idx, src[idx], dst[0], out);
}
```

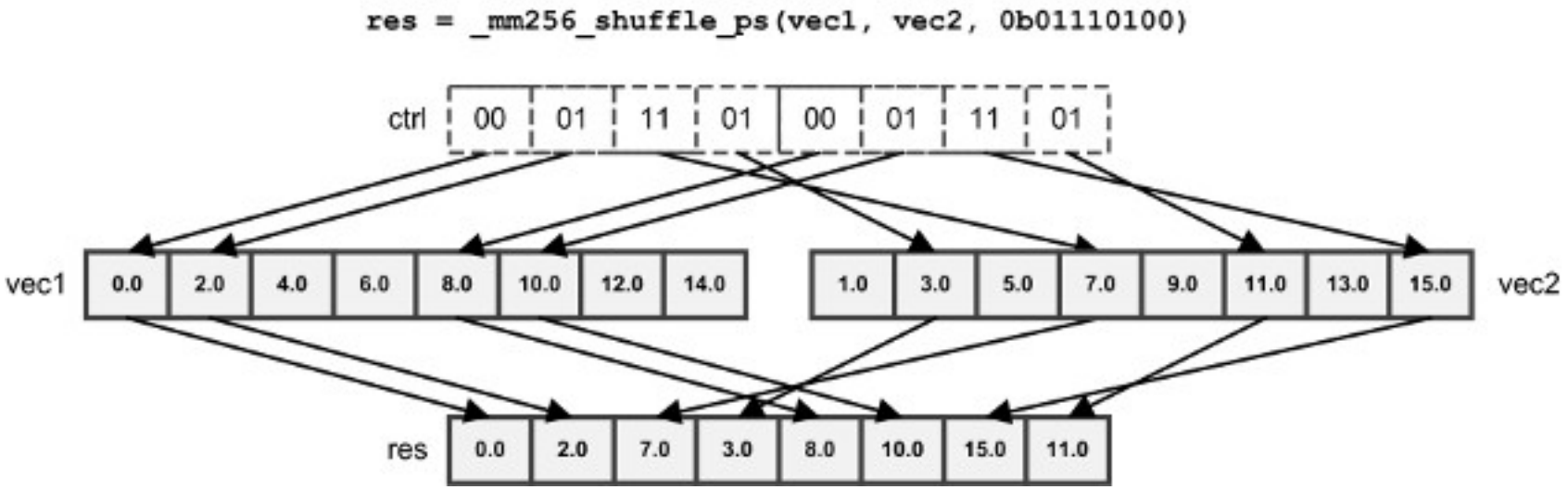
Atomic #5

- Global, Local (=Shared) Atomics
  - Atomic 연산의 성능을 높이기 위한 기능이 하드웨어에 포함
  - High Contention (Local 성능 최악) , Entropy (low = high contention)



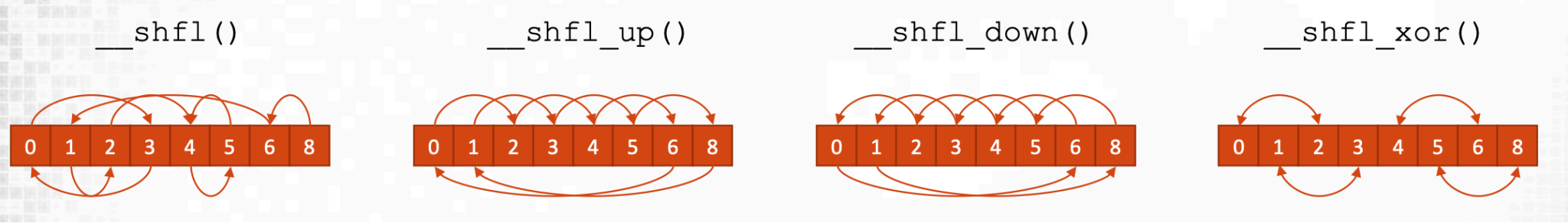
Warp Operations #1

- 동일한 Thread Block 내부에서는
  - Thread 간의 data 이동이나 비교 연산 가능
  - 하나의 Warp에서는 **Warp Shuffle (SHFL)** 사용 가능
    - 반드시 Conditional Execution에 영향 받는 일이 없어야 함
    - **Works by allowing threads to read another threads registers**



Warp Operations #2

- Shuffle Variants
  - Shuffled between any to index threads
    - `int __shfl(int var, int srcLane, int width=warpSize)`
    - `Direct copy of var in srcLane`
  - Shuffles to ^{n}th right neighbors wrapping indices (in this case n=2)
  - Shuffles to ^{n}th left neighbors wrapping indices (in this case n=2)
  - Butterfly (XOR) exchange shuffle pattern



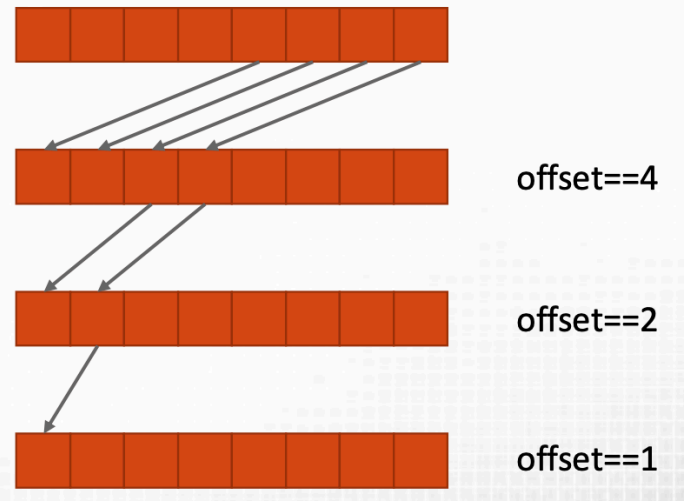
Warp Operations #3

- Shuffle Warp Sum Example (Reduction)
  - `int __shfl_down(int var, unsigned int delta, int width=warpSize)`
    - Delta is the n step used for shuffling

```
__global__ void sum_warp_kernel_shfl_down(int *a) {
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  for (int offset = WARP_SIZE / 2; offset>0; offset /= 2)
    local_sum += __shfl_down(local_sum, offset);

if (threadIdx.x%32 == 0)
printf("Warp max is %d", local_sum)
}
```



Warp Operations #4

- To test a condition across all thread in a warp
  - 모든 Warp Voting Function은 Barrier처럼 동작
  - `int all(condition), int any (condition), unsigned int ballot (condition)`

```
__global__ void voteAllKernel(unsigned int *input, unsigned int *result) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = i % WARP_SIZE;
  int vote_result = all(input[i]);

  if (j==0) result[j] = vote_result;
}
```

Warp Operations #4

- Sync Shuffle
  - Volta Architecture can interleave exaction of statements from divergent branch
  - Each thread has its own program counter to allow this

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```