[CSE4152] 고급소프트웨어 실습 I
Week 14

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

Turing TU102 GPU

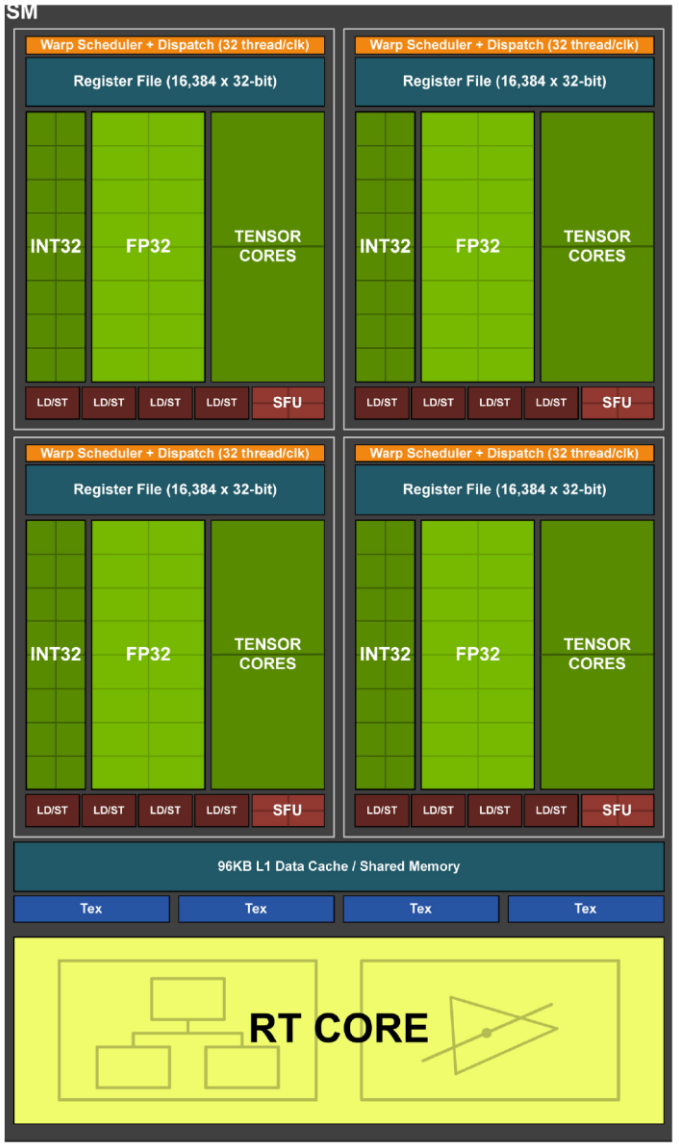
NVIDIA Turing GPU Architecture: Graphics Reinvented (2018)

- ✓ 6 **Graphics Processing Clusters**(GPCs)
- ✓ 6 **Texture Processing Clusters**(TPCs) / GPC
- ✓ → 36 TPCs
- ✓ 2 **Streaming Multiprocessors**(SMs) / TPC → **72 SMs**
- ✓ 64 **CUDA Cores** / SM → 4,608 CUDA Cores
- ✓ 1 **RT Core** / SM → 72 RT Cores
- ✓ 8 **Tensor Cores** / SM → 576 Tensor Cores
- ✓ 4 **texture units** / SM → 288 texture units
- ✓ 12 **memory controllers**
- ✓ 8 **ROP units** / MC → 96 ROP units
- ✓ 512 KB of **L2 cache** / MC → 6144 KB of L2 cache

TU102/TU104/TU106 Streaming Multiprocessor (SM)

NVIDIA Turing GPU Architecture: Graphics Reinvented (2018)

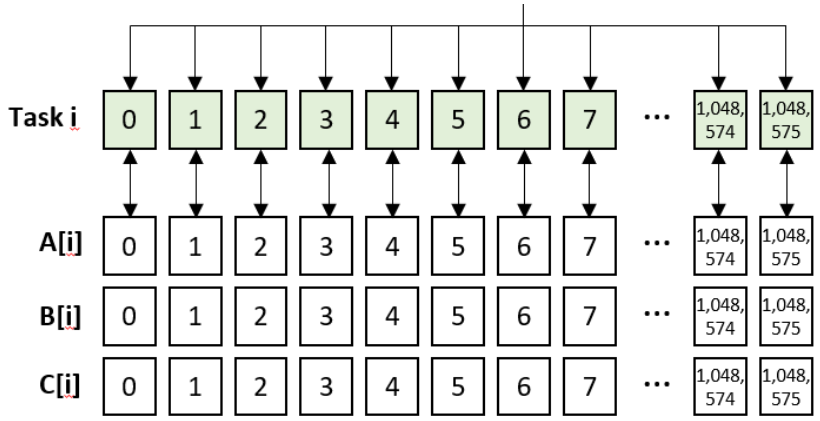
- **Each SM includes**
  - 64 FP32 Cores / 64 INT32 Cores / 2 FP64 Cores
    - Supports concurrent execution of FP32 and INT32 operations.
  - 8 mixed-precision Tensor Cores / 1 RT core
- **Each SM is partitioned into** four processing blocks **each with**,
  - 16 FP32 Cores / 16 INT32 Cores (2 cycles per warp)
  - 2 Tensor Cores
  - 1 warp scheduler
  - 1 L0 instruction cache
  - 1 **64 KB register file: 16,384 x 32-bit**
- The four blocks shares **a combined 96 KB L1 data cache/shared memory**:
  - 64 KB of graphics shader RAM + 32 KB for texture cache
  - **32 KB shared memory + 64 KB L1 cache**
  - **64 KB shared memory + 32 KB L1 cache**

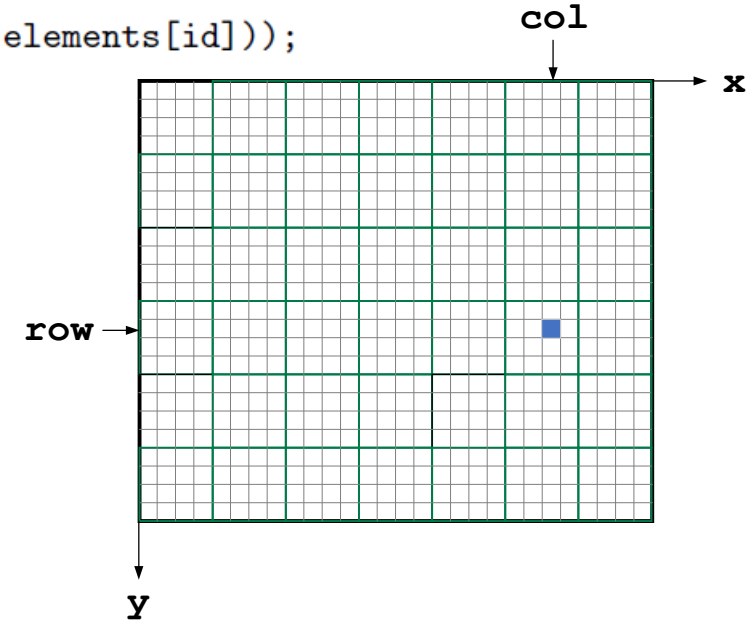


Grids, Thread Blocks, and Threads

```
for (i = 0; i < n; i++) {
  z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
}
```

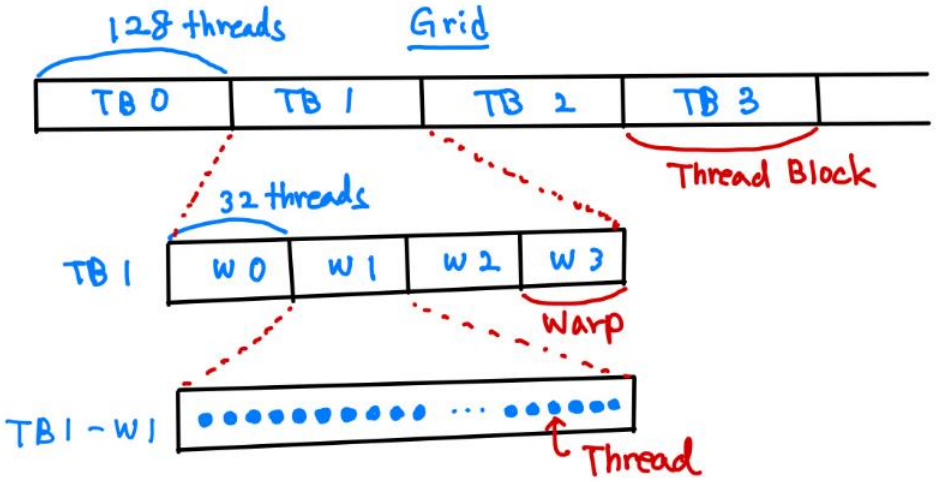
```
__global__ void CombineTwoArrraysKernel(Array A, Array B, Array C) {
  int row = blockDim.y*blockIdx.y + threadIdx.y;
  int col = blockDim.x*blockIdx.x + threadIdx.x;
  int id = gridDim.x*blockDim.x*row + col;

  C.elements[id] = 1.0f / (sin(A.elements[id])*cos(B.elements[id])
                           + cos(A.elements[id])*sin(B.elements[id]));
}
```



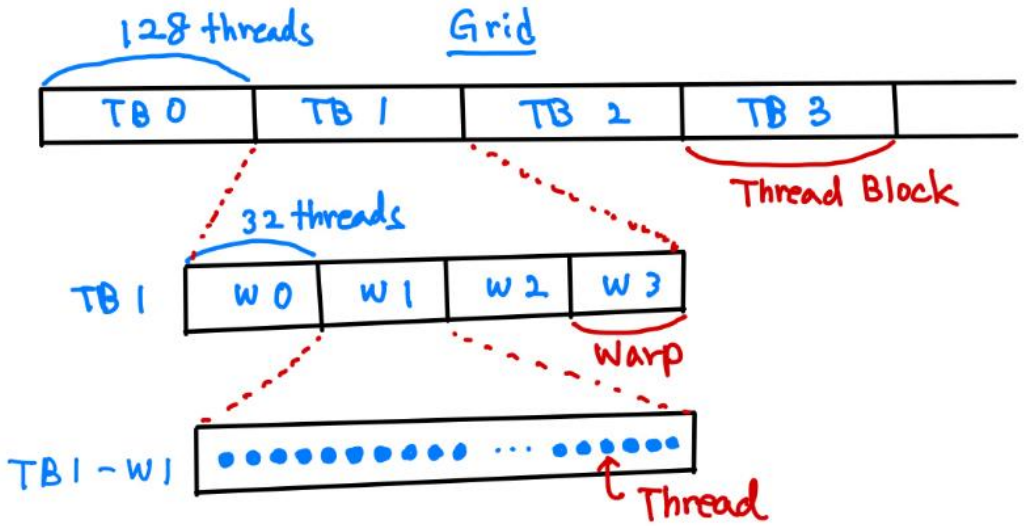


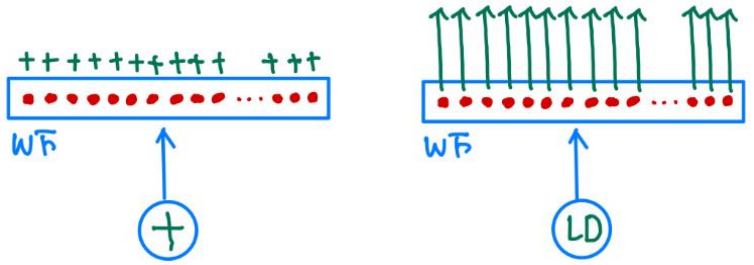
Thread Processing on GPU

- **Grid and thread block:** 전체 grid의 thread들은 동일한 크기의 thread block으로 분할됨.
- **Thread Block Scheduling: GPU scheduler**는 전체 thread block pool의 thread block들을 순차적으로 streaming multiprocessor(SM)에 배분하여 처리토록 함.
  - SM의 제한된 resource로 인하여, 한 순간에 한 SM이 처리할 수 있는 thread block의 개수에는 제한이 걸림.
  - 프로그래머는 한 thread block이 어떤 순서대로 시작해서 어떤 순서로 끝날 지 예측할 수 없음 → 모든 thread block의 처리가 끝난 시점이 kernel 수행의 종료 시점임.

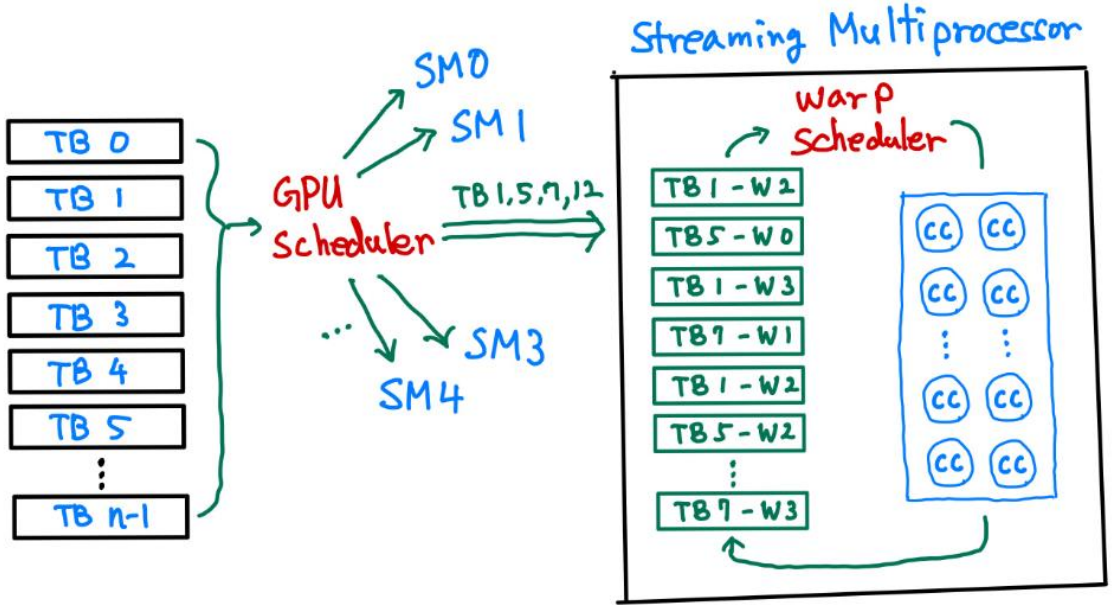


- **Thread block processing:** 특정 SM으로 배정된 **thread block**은 다시 **warp** 단위로 분할되어 처리됨.
  - ➢ **Warp:** 32 threads
  - 한 **warp 내의 thread들은 SIMT 형태로 처리**된다고 생각하면 됨.
    - 하나의 instruction이 모든 thread에 대하여 동시에 수행됨: **execute in lockstep**.
  - 한 thread block내의 warp들은 서로 독립적으로 처리되므로, 서로 간에 동기화(synchronization)가 필요한 경우 발생.





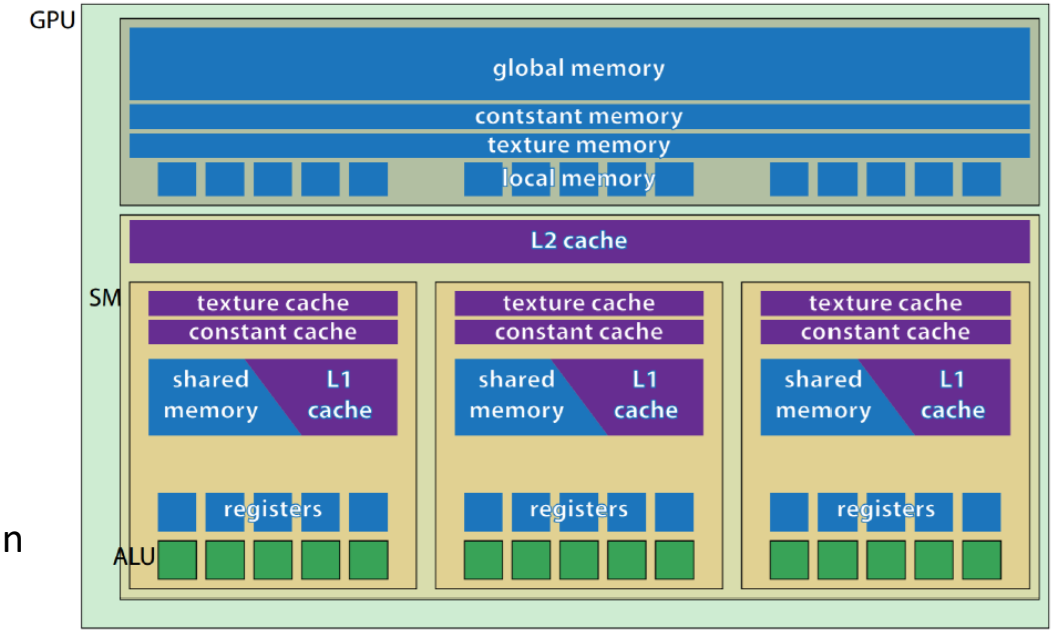
- **Warp processing: warp scheduler**는 현재 SM에 배정된 thread block들의 **warp pool**로부터 warp를 적절히 processing block에 배정하여 처리토록 함.
  - 한 processing block에 배정된 warp가 메모리 접근 등으로 인하여 대기가 발생하면 , **zero-cost context switching**을 통하여 새로운 warp가 processing block에 배정됨 ← **hiding memory latency with ALU operations**.
  - 한 thread block에서 생성된 warp들이 어떤 순서대로 시작해서 어떤 순서로 끝날지 예측할 수 없음 → 한 thread block을 구성하는 모든 warp들이 끝났을 때가 그 thread block의 처리 종료 시점임.
  - ➢ 처리가 끝난 thread block이 SM에서 빠져 나오면 대기중인 thread block이 배정됨.



Example: NVIDIA Turing GPU
- Max 32 warps per SM
- 64 CUDA cores per SM
- 4 Processing Blocks per SM

CUDA Memory Hierarchy

- **Private local memory**: Register and local memory
  - Scope: the thread to which the register space is allocated
  - Lifetime: the thread
  - Latency: fastest
- **Shared memory**
  - Scope: all threads of the block
  - Lifetime: the block
  - Latency: a few dozens of cycles
- **Global memory**
  - Scope: all threads within the host application
  - Lifetime: the host application
  - Latency: a few hundred cycles (without caching)
- **Constant memory**
  - Scope: all threads within the host application
  - Lifetime: the host application
  - Latency: fast when all threads in a warp access the same location

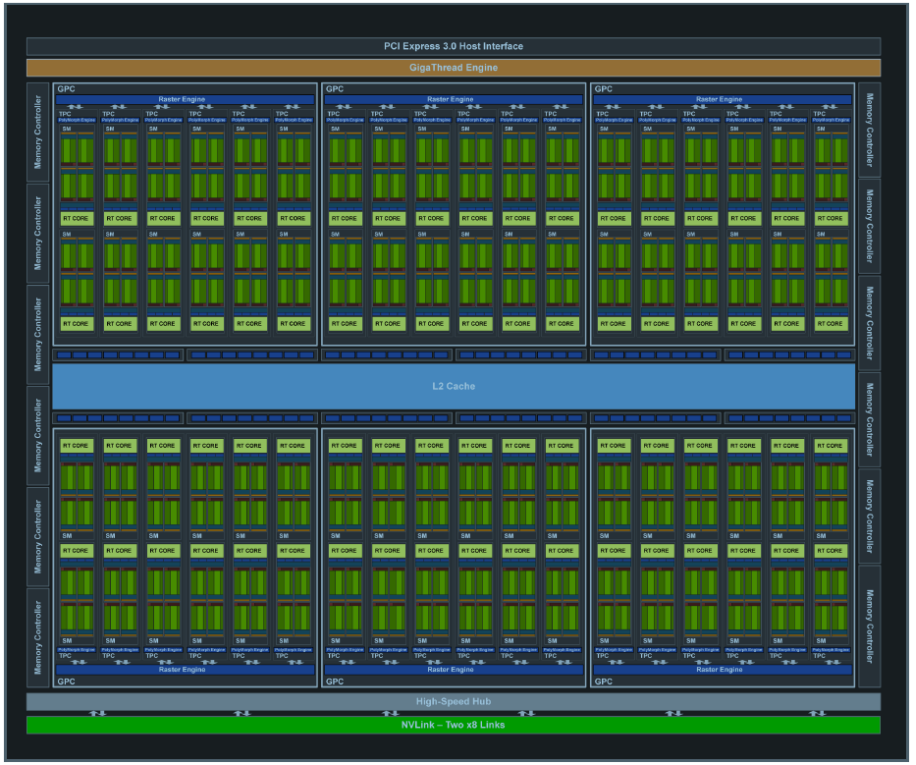


CUDA Memory Hierarchy (P. Danilewski)

Global Memory Access on NVIDIA GPU

- **Global memory** is a off-chip memory whose access **can take hundreds of cycles** ← high memory latency!
- **Global memory caching** (CUDA compute capability 3.5 or beyond)
  - Global memory accesses are cached in L2.
  - They may also be cached in the read-only data cache.
  - An opt-in caching in L1 is also allowed.

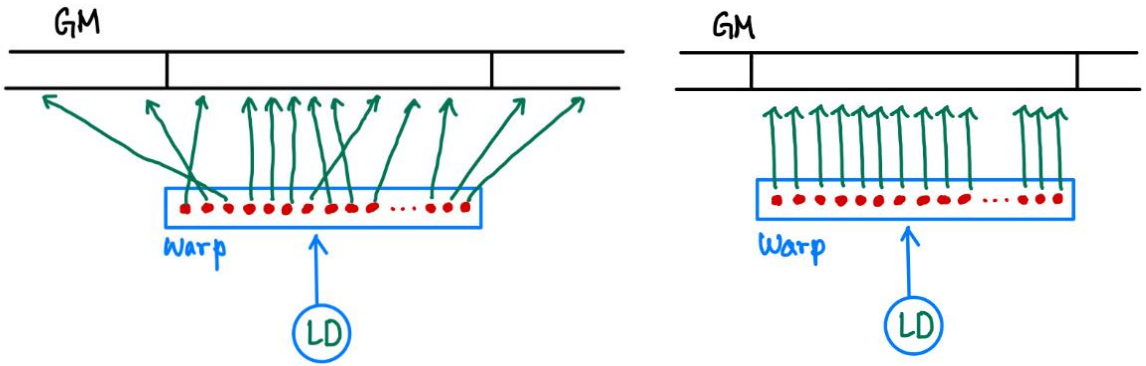
Locality of reference: spatial and temporal



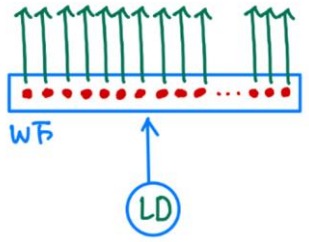
NVIDIA TURING GPU ARCHITECTURE

Coalesced memory access

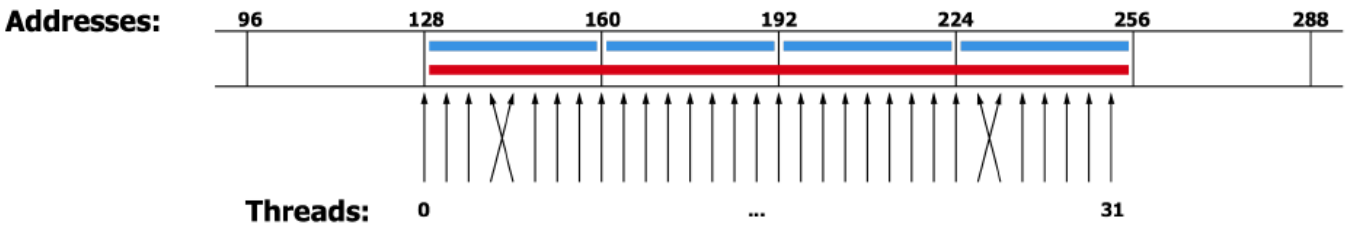
- **Cache line size**
  - A cache line is **128 bytes** and maps to a **128-byte aligned segment** in global memory.
  - Memory accesses that are cached **in both L1 and L2** are serviced with **128-byte memory transactions**.
  - Memory accesses that are cached **in L2 only** are serviced with **32-byte memory transactions**.
    - Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses



- **Global memory access**
  - Each memory request is then **broken down into cache line requests that are issued independently**.
  - A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

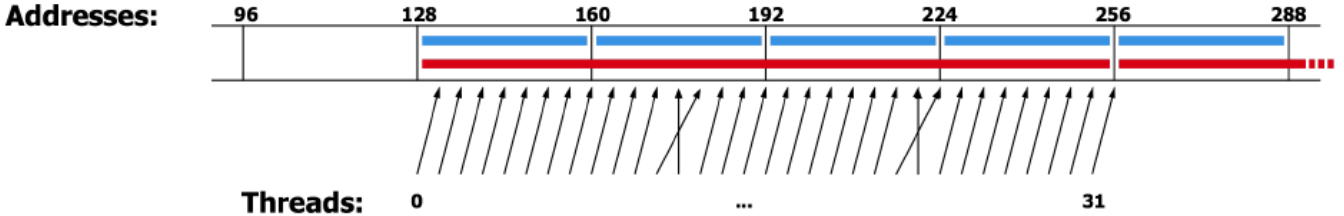


- **Example 1**
  - Aligned accesses
  - 4-byte word per thread in a warp



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

- **Example 2**
  - Mis-aligned accesses
  - 4-byte word per thread in a warp



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

강의 자료 15쪽 코드

```
__global__ void TransformAOSKernel(POINT_ELEMENT *A, int m) {
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int id = gridDim.x*blockDim.x*row + col;

  for (int j = 2; j <= m; j++) {
    float tmp = 1.0f / (float) j;
    for (int i = 0; i < ELEM_PER_POINT; i++) {
      A[id].elem[i] += tmp*A[id].elem[i];
    }
  }
}
```

```
typedef struct {
  float elem[ELEM_PER_POINT];
} POINT_ELEMENT; // for AOS
```