Shared Memory

송민호

유튜브: https://youtu.be/IWJHnz-Lxjg

Memory Type

- **GPU에는 두 종류의 메모리 타입이 존재**

- On-board memory
  - Global Memory
  - 크기가 크고 latency가 높음

- On-chip memory
  - Shared Memory
  - 크기가 작고 latency가 작음
  - ➢ Global Memory보다 더 높은 bandwidth 제공

Shared Memory

- **Shared Memory는 GPU에서 key 컴포넌트 중 하나**

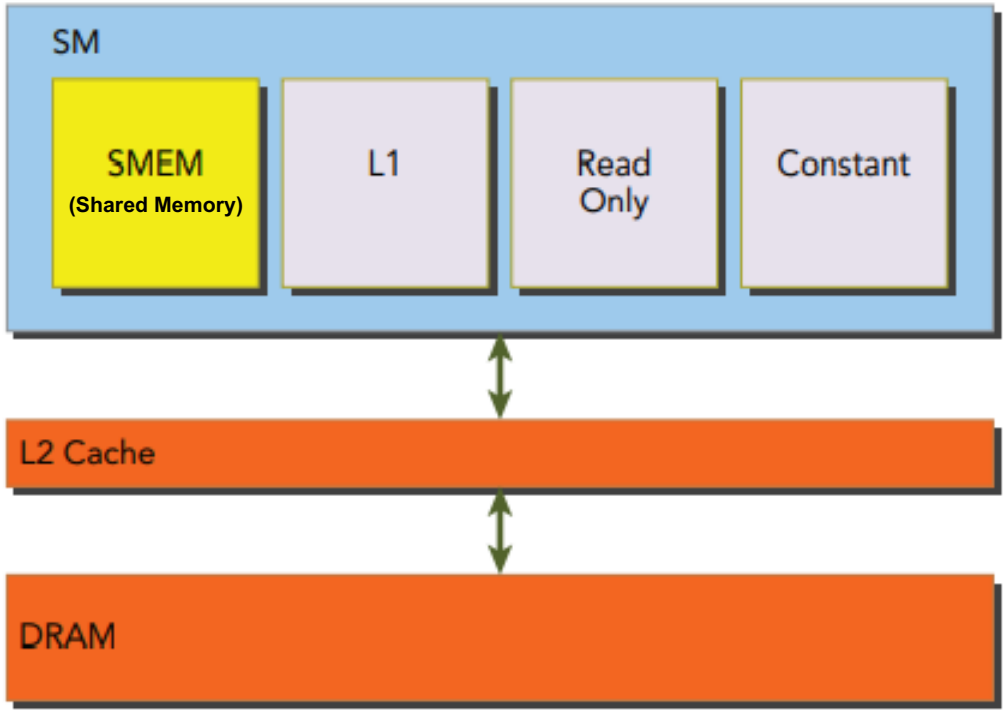
- SM(Streaming Multiprocessor)은 low-latency 메모리풀 포함
  - 메모리풀: 성능을 개선하고 메모리 단편화 문제를 개선하기 위한 방법
    - 필요 크기의 메모리를 미리 할당하고, 필요할 때마다 사용하고 반납하는 방법
  - SM에서 현재 실행 중인 스레드 블록 내의 모든 스레드가 공유
    - Shared Memory는 동일한 스레드 블록 내에서 스레드들이 협력할 수 있도록 도와줌
    - On-chip 데이터의 재사용성을 높여줌
    - Global memory bandwidth를 크게 줄일 수 있도록 해줌
  - Shared Memory의 데이터는 어플리케이션에서 명시적으로 관리되기 때문에 종종 program-managed cache라고도 함

Shared Memory

- 물리적으로 SM에 가까운 Shared Memory

- Shared Memory, L1 cache는 물리적으로 Global Memory, L2 cache보다 SM에 가까움

- Shared Memory > Global Memory
  - Latency – 20~30배 낮음
  - Bandwidth – 10배 높음



GPU의 메모리 계층

Shared Memory

- 스레드 블록들이 실행될 때, 고정된 크기의 Shared Memory가 각 스레드 블록에 할당
  - Shared Memory 주소 공간은 동일한 스레드 블록의 모든 스레드들에게 공유
  - 따라서, Shared Memory의 데이터들은 스레드 블록에서 동일한 lifetime을 가짐

- Shared Memory 액세스는 warp(32개 스레드) 단위로 실행
  - 액세스를 위한 warp에서 요청은 한 번의 transaction으로 처리되는 것이 이상적
  - 최악의 경우, 32개의 transaction에서 순차적으로 수행

- Shared Memory는 device 병렬화를 제안하는 중요한 리소스
  - SM에 현재 존재하는 스레드 블록들 사이에서 분할되기 때문
  - 커널의 더 많은 Shared Memory를 사용한다면 동시에 활성화 되는 스레드 블록은 더 적어짐

Shared Memory 할당

- **정적, 동적 할당 가능**

- 변수 선언

```
__shared__
```

- 크기를 알 수 없는 경우 extern 사용 가능
  - 1차원의 사이즈가 결정되지 않은 int 배열 선언

```
extern __shared__ int tile[];
```

- 커널을 시작할 때 원하는 크기를 byte 단위로 지정
  - 3번째 파라미터에 지정하여 동적 할당 가능
  - 동적 할당하는 경우 1차원의 배열만 선언할 수 있음

```
kernel<<<grid, block, size * sizeof(int)>>>(...)
```

Memory Banks

- **높은 메모리 bandwidth를 달성하기 위함**
  - Banks라는 32개의 동일한 사이즈의 메모리 모듈로 나뉘며 동시 엑세스 가능
  - Bank가 32개인 이유는 하나의 warp에 32개의 스레드가 있기 때문

- Shared Memory는 1차원 주소 공간
  - GPU의 연산량에 따라서 Shared Memory의 주소는 다른 패턴으로 다른 bank에 매핑

  - 한 번의 memory transaction으로 처리되는 경우
    - Warp에 의한 Shared Memory 명령이 bank당 둘 이상의 메모리에 엑세스하지 않을 때

  - 여러 번의 memory transaction으로 처리되는 경우
    - Warp에 의한 Shared Memory 명령이 bank당 둘 이상의 메모리에 엑세스할 때

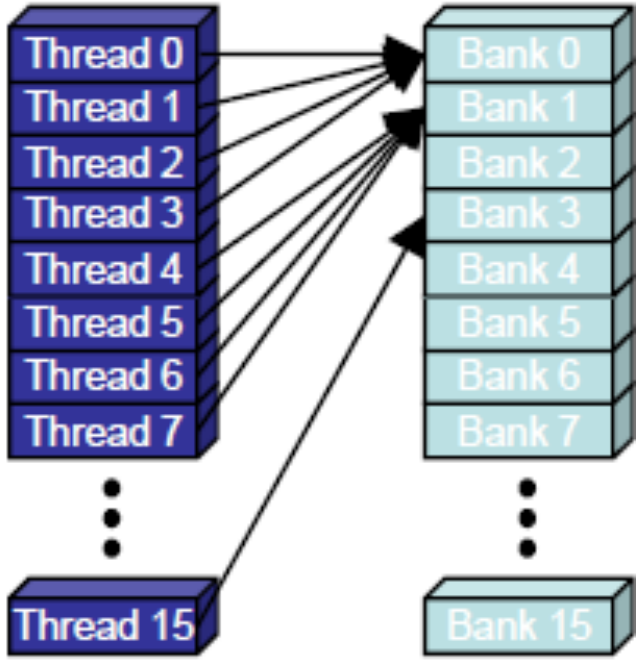
    ➢ 메모리 bandwidth 활용 감소

Bank Conflict

- **Shared Memory의 여러 주소들에 대한 요청이 동일한 bank에 발생**
  - 요청 반복

  - 하드웨어는 bank conflict가 발생한 요청을 conflict가 발생하지 않는 여러 transaction으로 필요한만큼 분할
  - 분할된 memory transaction의 비율만큼 bandwidth 감소

  - Warp에 의해 Shared Memory에 대한 요청이 있을 때 3가지 상황의 접근 방법이 발생함
    - Parallel access
    - Serial access
    - Broadcast access



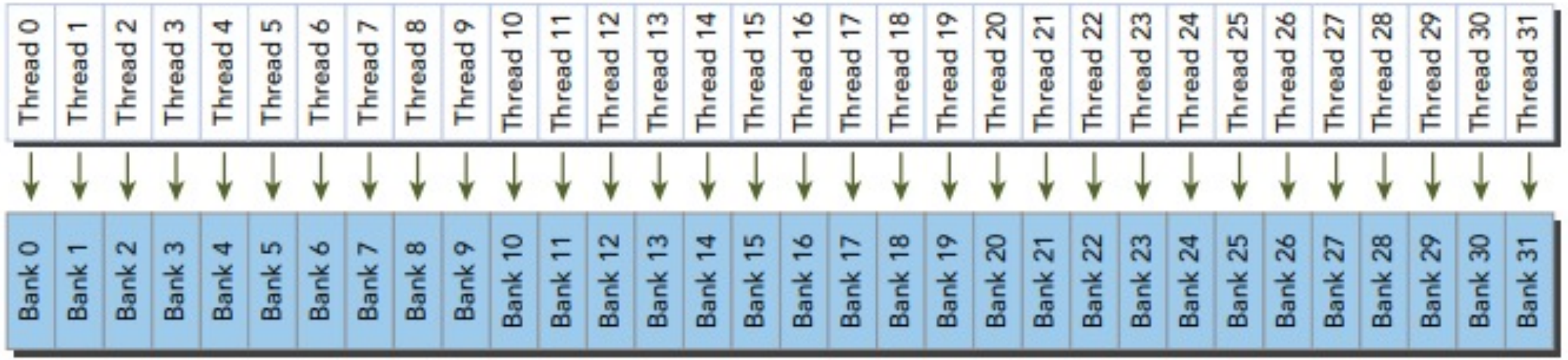
Bank Access

- Parallel access
  - 가장 일반적인 패턴
  - 전부는 아니지만 주소의 일부가 하나의 메모리 transaction으로 처리 가능
  - 모든 주소가 별도의 bank에 있을 때 conflict가 없음
- Serial access
  - 가장 최악의 패턴
  - 여러 주소가 동일한 bank에 속할 때 메모리 요청이 연속적으로 처리됨
  - Warp 내의 모든 스레드가 단일 bank에 속할 경우 32배 오래 걸림
- Broadcast access
  - 한 warp의 모든 스레드가 하나의 bank 내의 동일한 주소를 읽음
  - 한 번의 메모리 transaction이 수행되고 액세스된 word는 모든 스레드로 broadcast
  - 아주 적은 양의 bytes만 읽으므로 bandwidth 활용도가 낮음

Bank Access 패턴

- 최적의 parallel access 패턴

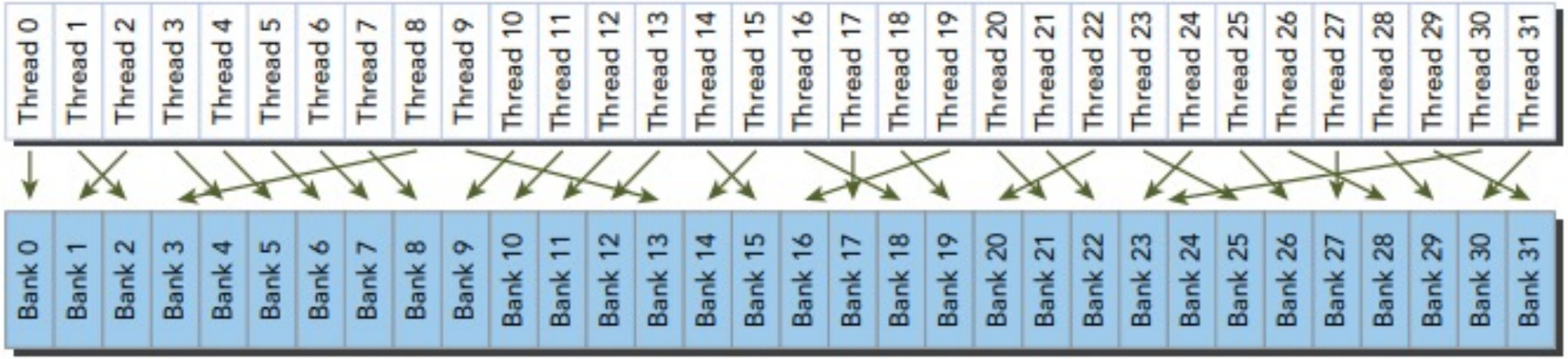
- 각 스레드가 하나의 bank에 액세스
  - Bank conflict가 발생하지 않음



Bank Access 패턴

- 불규칙적이고 랜덤한 액세스 패턴

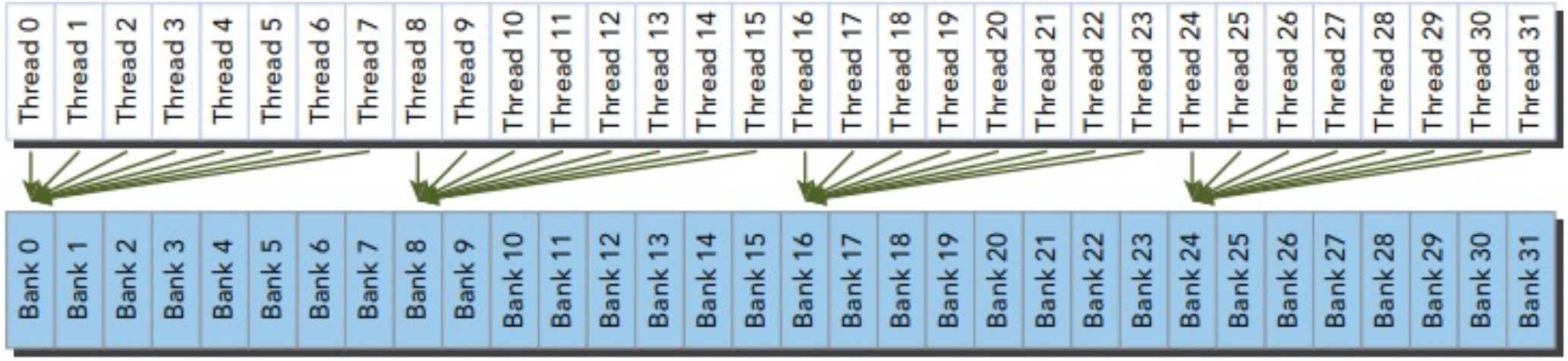
- 마찬가지로 각 스레드가 하나의 bank에 액세스
  - Bank conflict가 발생하지 않음



Bank Access 패턴

- 동일한 bank의 주소에 엑세스하는 불규칙 액세스 패턴

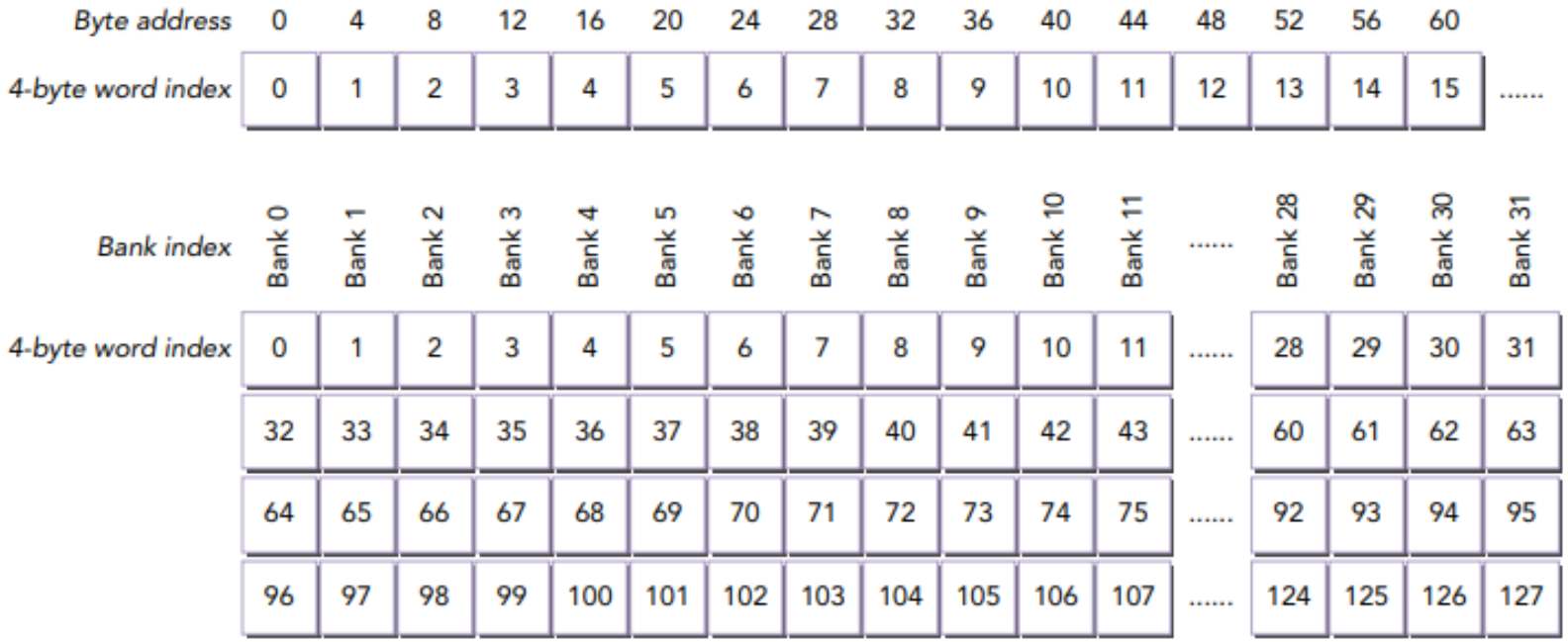
- 여러 스레드가 하나의 bank에 액세스
  - Bank conflict 발생



Bank Access Mode – 32bit mode

- 연속적인 32비트 words가 연속적인 bank에 매핑
  - 각 bank는 2클럭 사이클당 32bit의 bandwidth

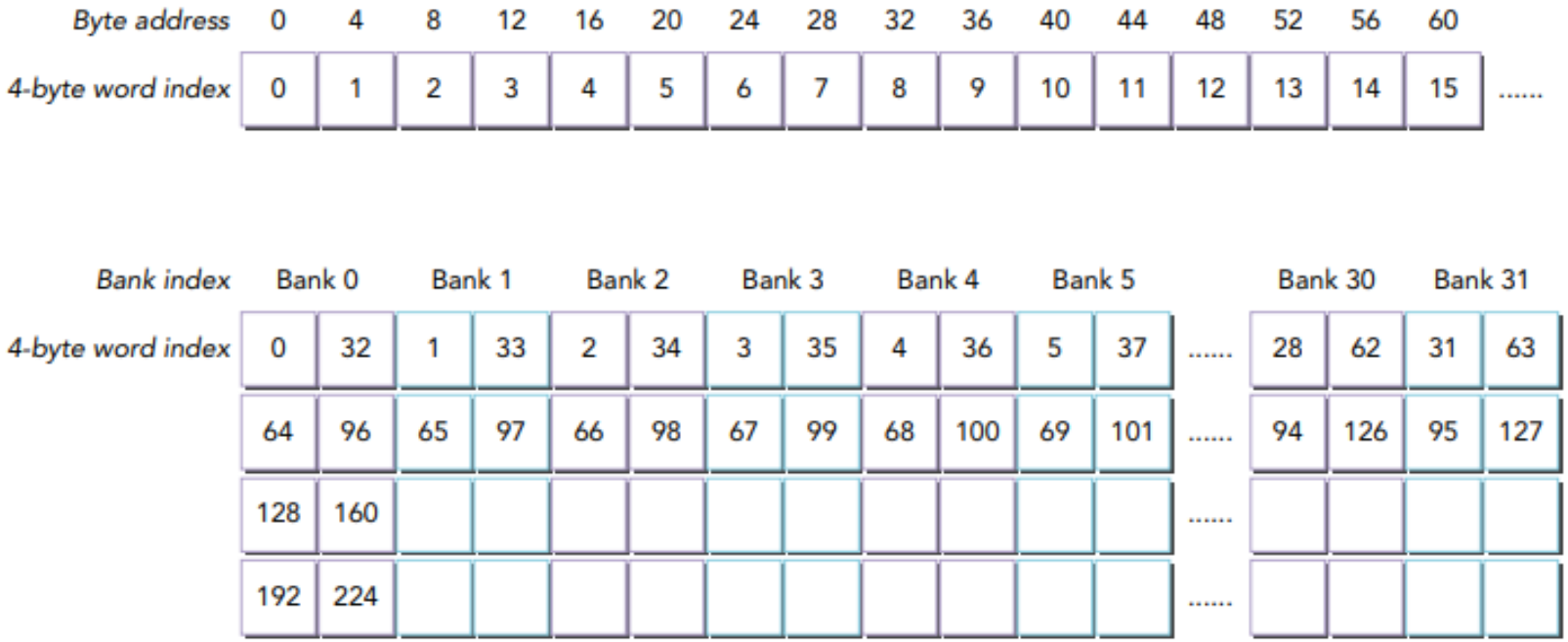
- Shared memory 주소에서 bank index의 매핑
  - Bank index = (byte address / 4bytes/bank) % 32 banks



Bank Access Mode – 64bit mode

- 연속적인 64비트 words가 연속적인 bank에 매핑
  - 각 bank는 클럭 사이클당 64bit의 bandwidth

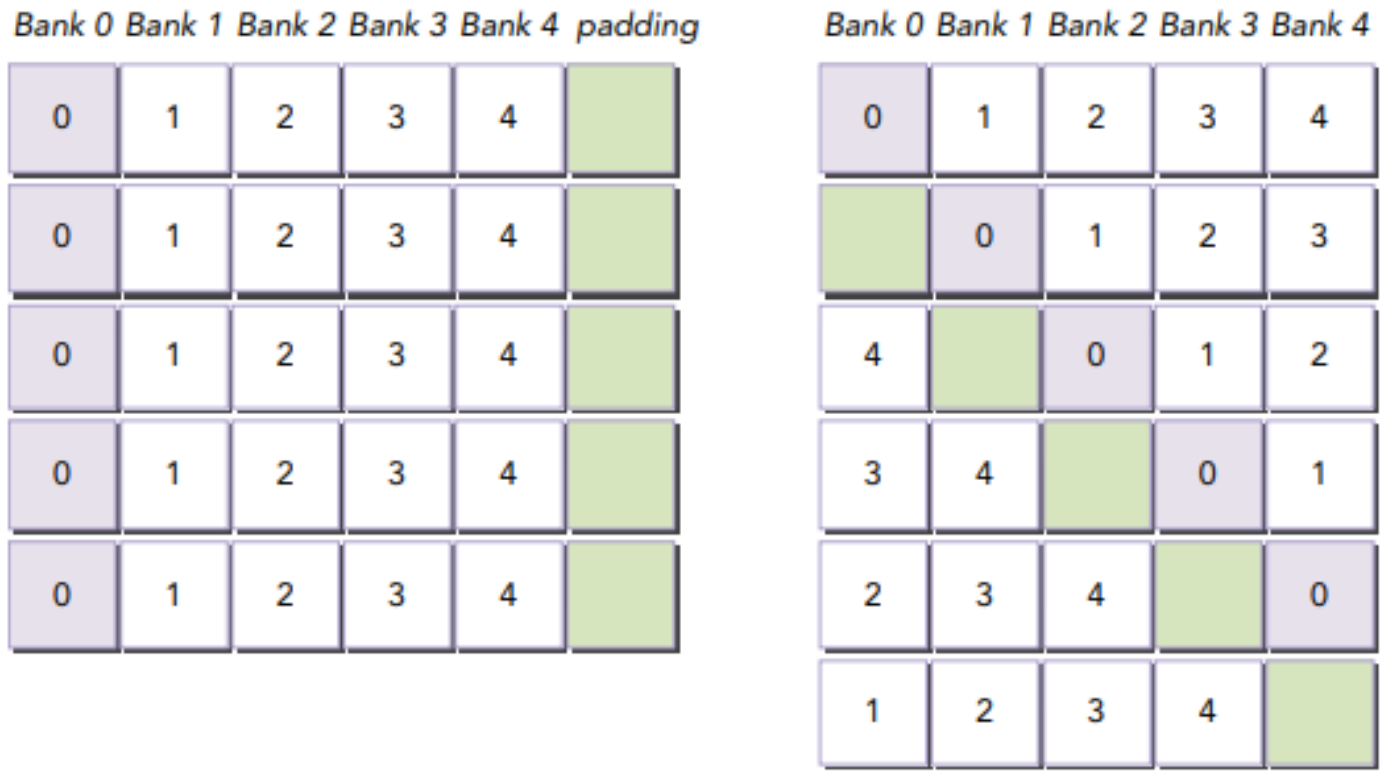
- Shared memory 주소에서 bank index의 매핑
  - Bank index = (byte address / 8bytes/bank) % 32 banks



Memory Padding

- Bank conflict를 피하는 한 가지 방법

- 마지막 원소 뒤에 하나의 padding word 추가



Q & A