CSCI-UA.0201

Computer Systems Organization

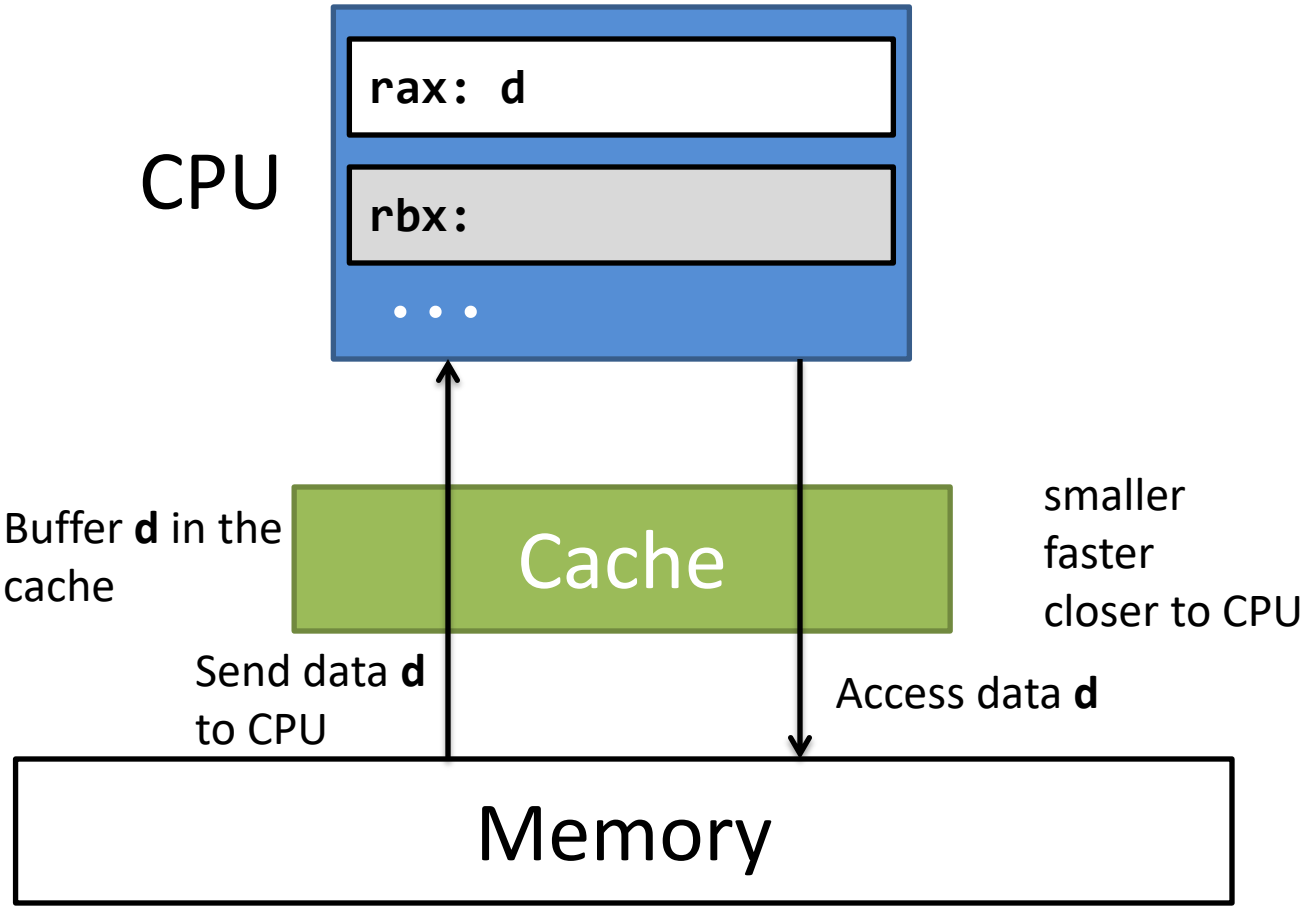
Memory Management – Caching

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

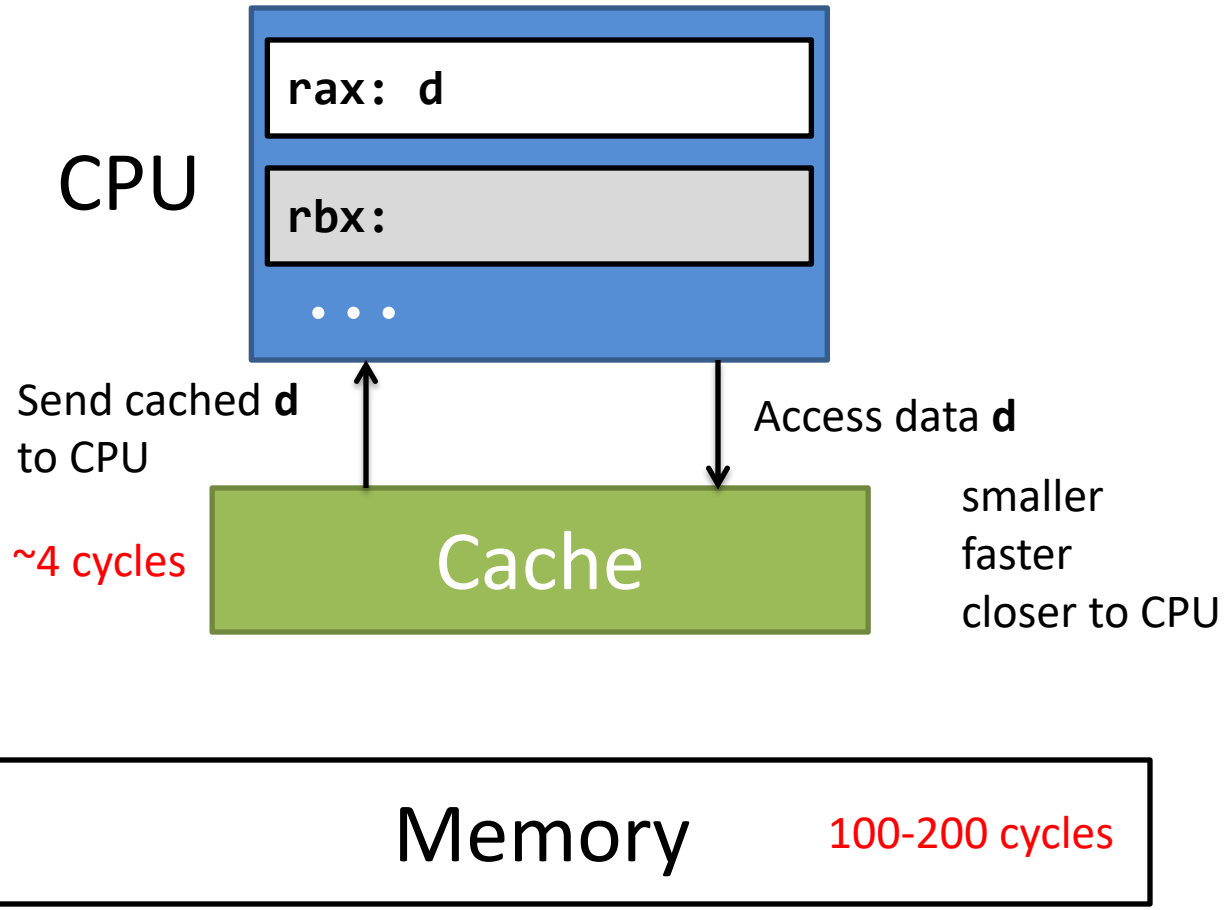
Principle of Locality

- **Temporal locality**
  - If memory location x is referenced, then x will likely be referenced again in the near future.
- **Spatial locality**
  - If memory location x is referenced, then locations near x will likely be referenced in the near future.
- **Idea**
  - Buffer recently accessed data in cache close to CPU

Basic Idea - Caching

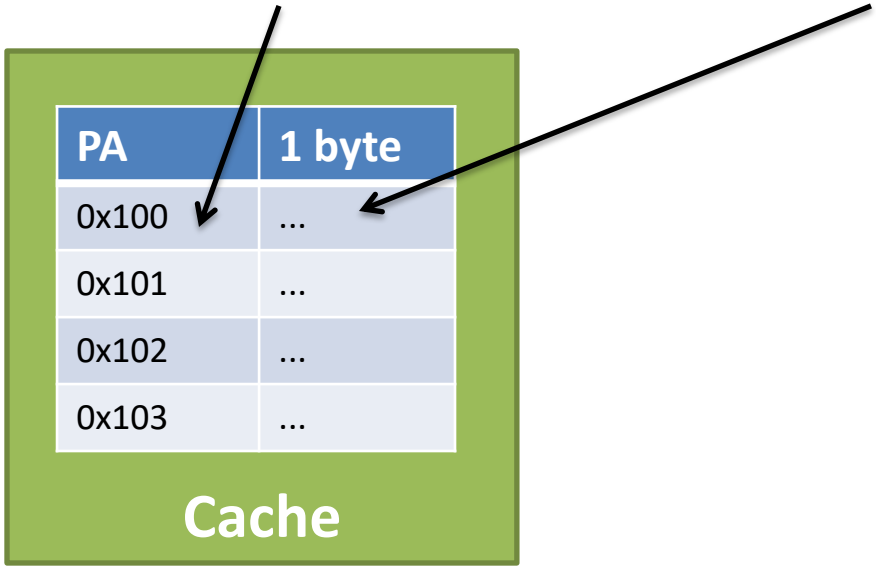


Basic Idea - Caching



Intuitive implementation

- Caching at byte granularity:
  - Search the cache for each byte accessed
    - `movq (%rax), %rbx` → checking 8 times
- High bookkeeping overhead
  - each cache entry has 8 bytes of address and 1 byte of data



Caching at Block Granularity

Solution:

- Cache one block (cache line) at a time.
- A typical cache line size is 64 bytes

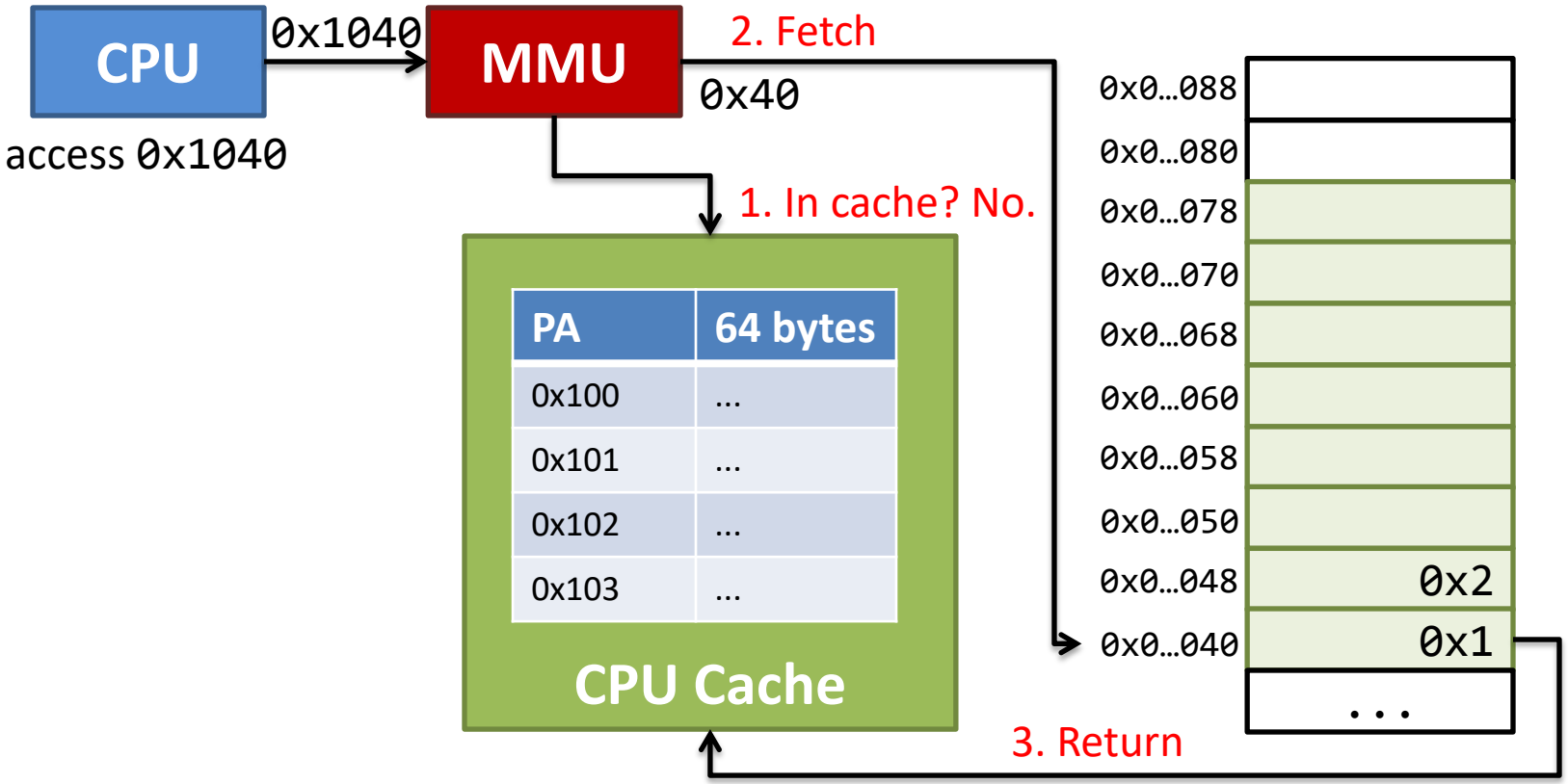
Advantage:

- Lower bookkeeping overhead
  - A cache line has 8 byte of address and 64 byte of data
- Exploits spatial locality
  - Accessing location x causes 64 bytes around x to be cached

Direct-mapped cache

Caching at block granularity

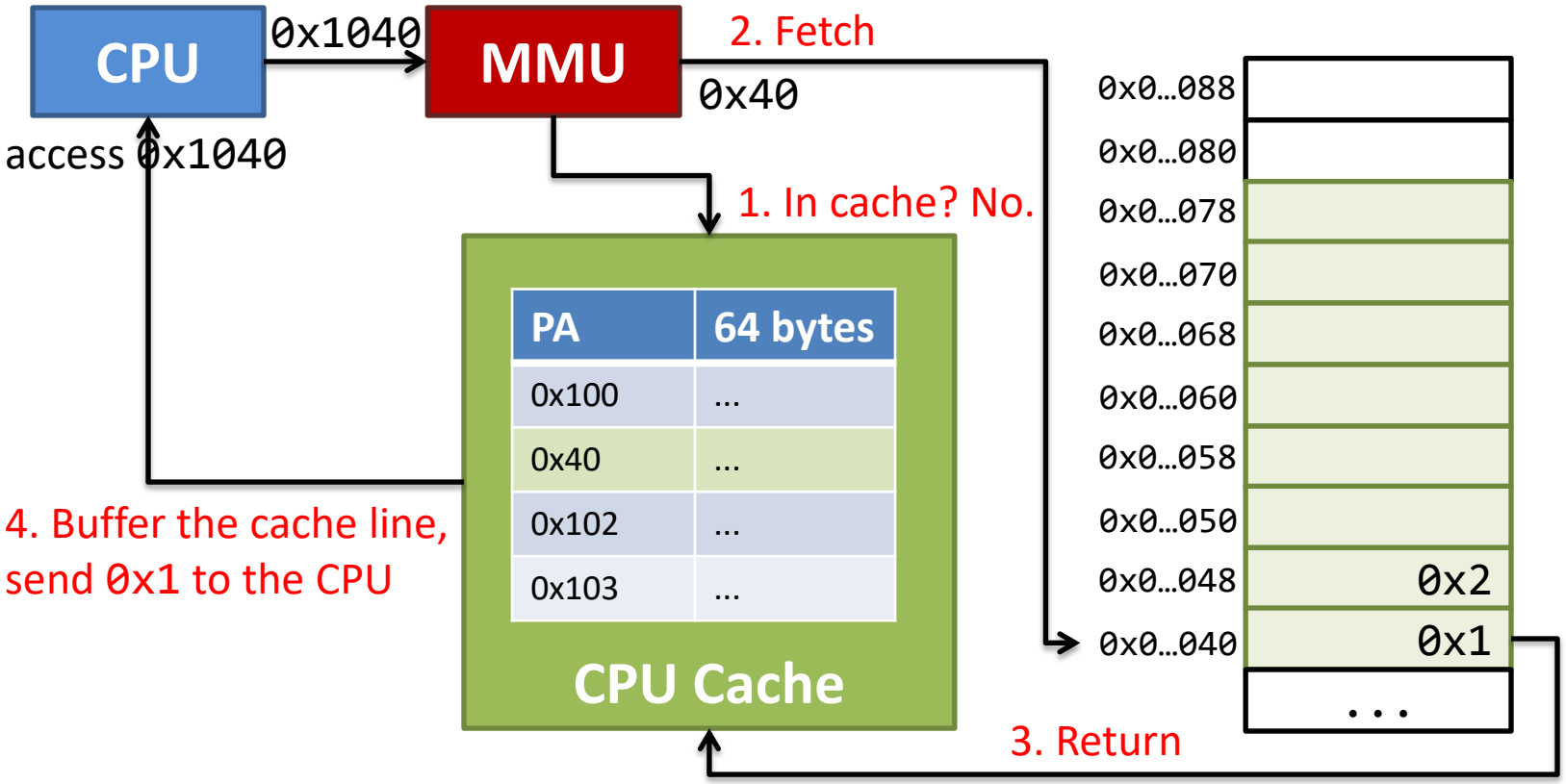
- Each cache line has 64 bytes



Direct-mapped cache

Caching at block granularity

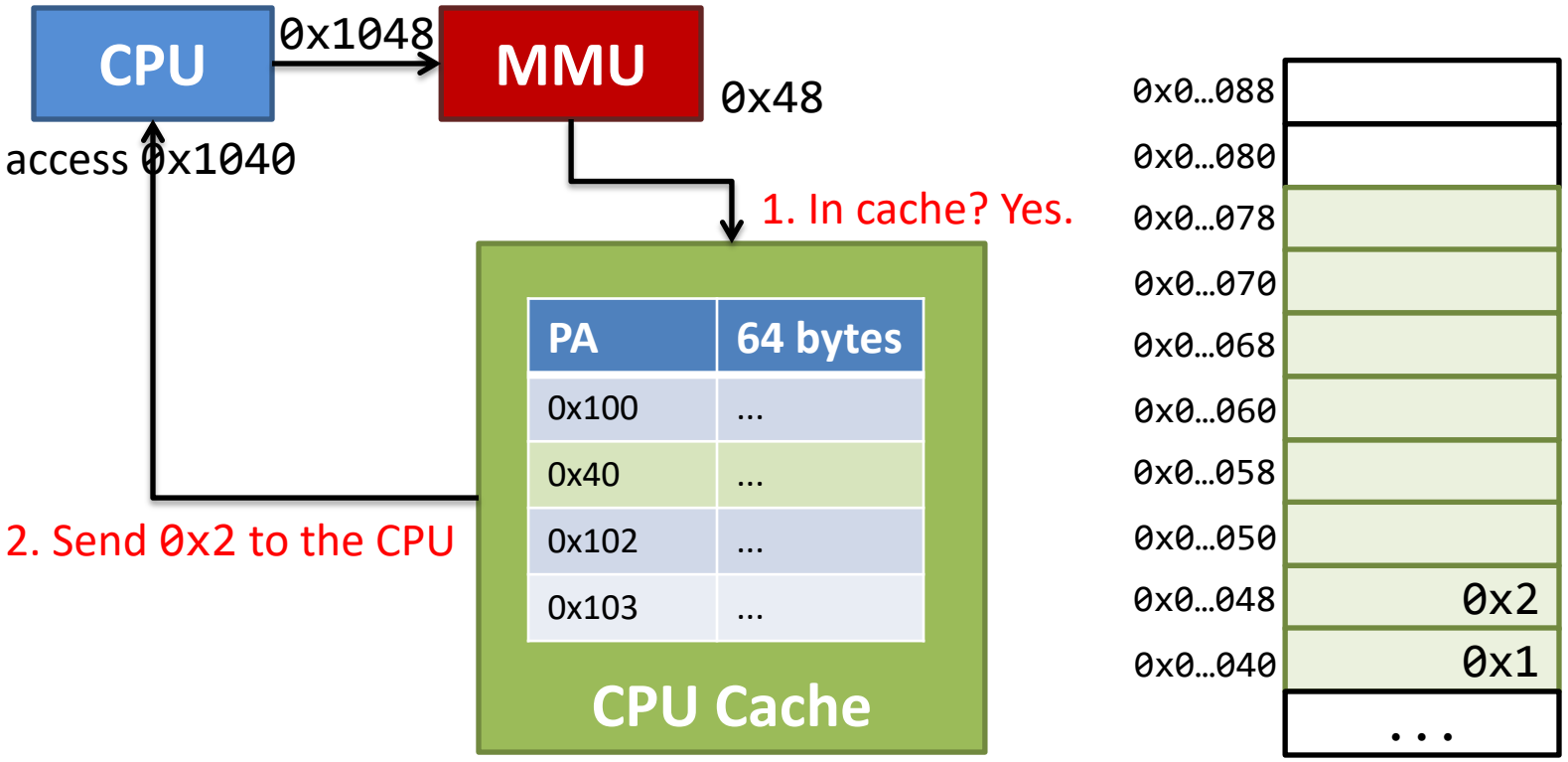
- Each cache line has 64 bytes



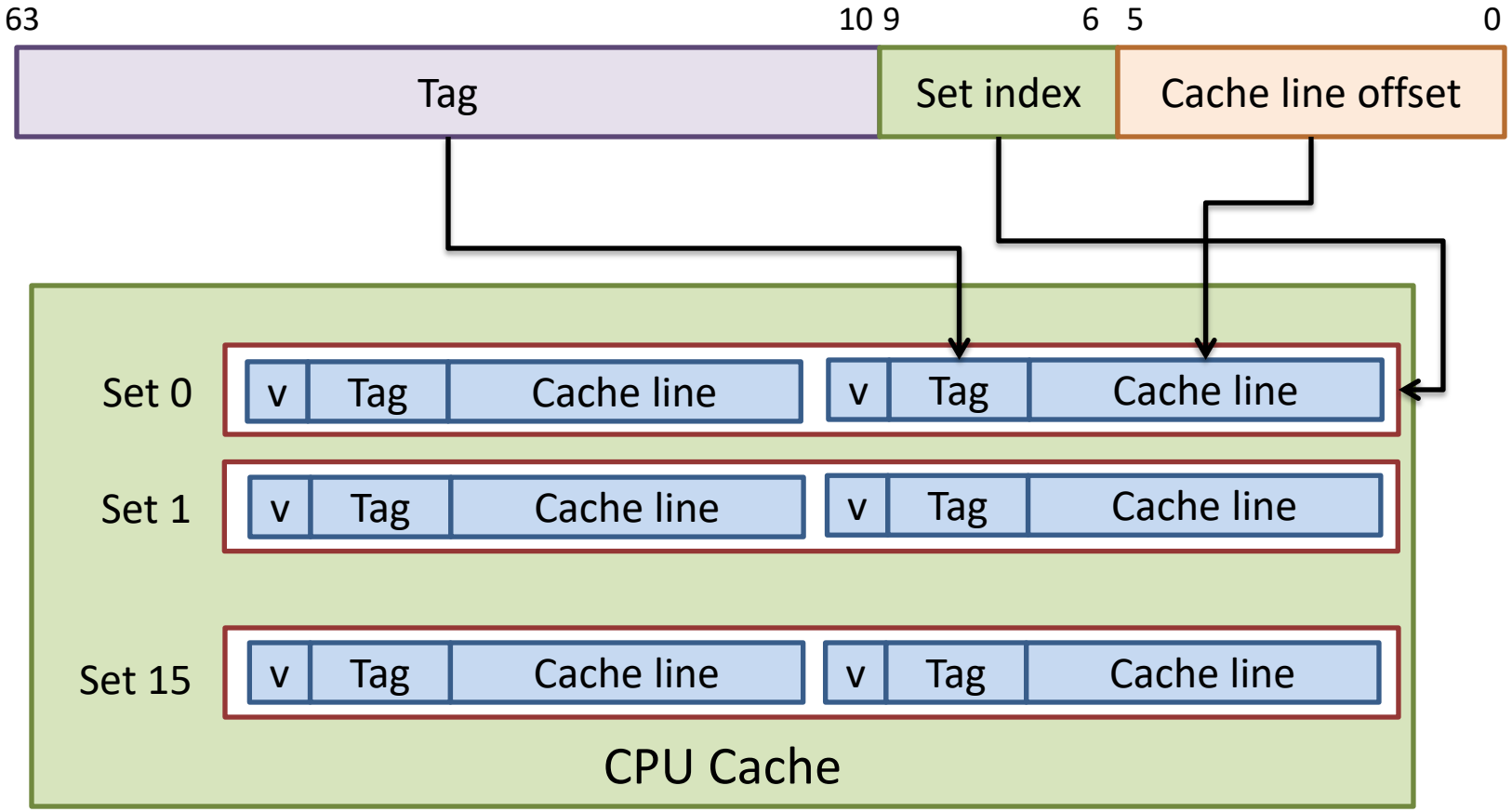
Direct-mapped cache

Caching at block granularity

- Each cache line has 64 bytes



Multi-way set associative cache



2-way set associative cache

Cache line replacement policy

- LFU (least-frequently-used)
  - Replace the line that has been referenced the fewest times over some past time window

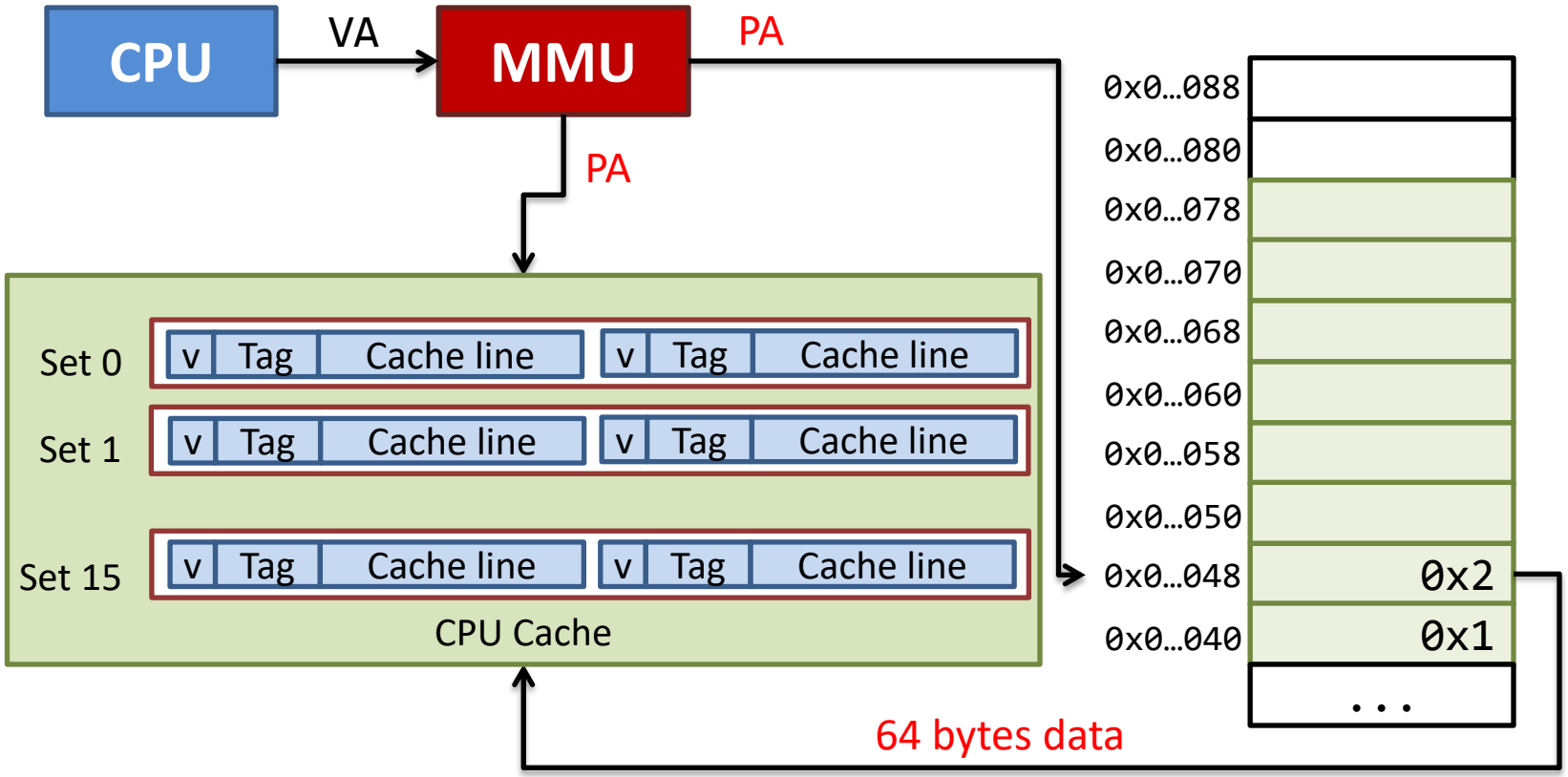
- LRU (least-recently-used)
  - Replace the line that has the furthest access in the past

These policies require additional time and hardware.

Further Optimizing Memory Access

Current design: address translation $\rightarrow$ cache access

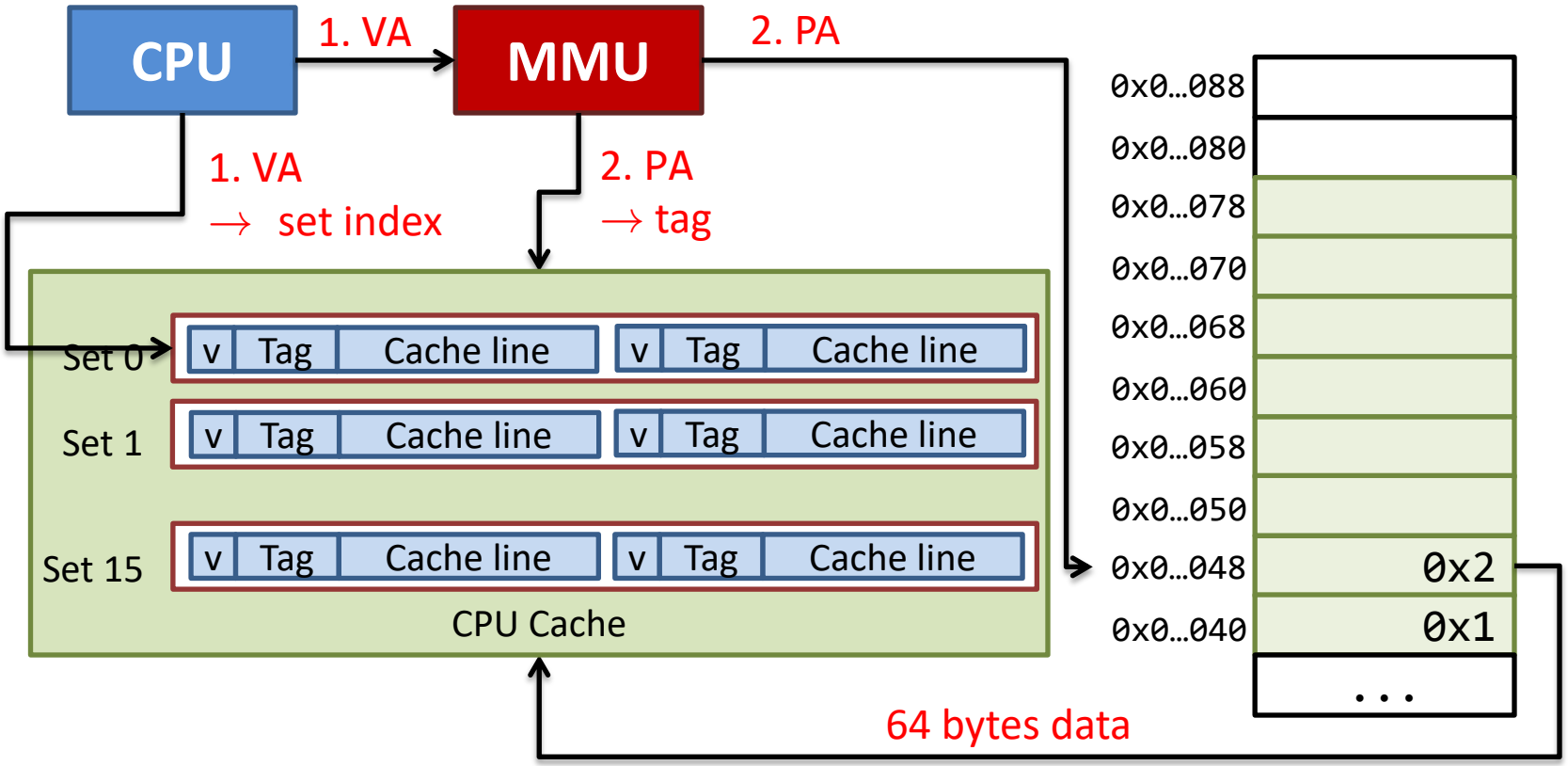
- The two steps are performed sequentially



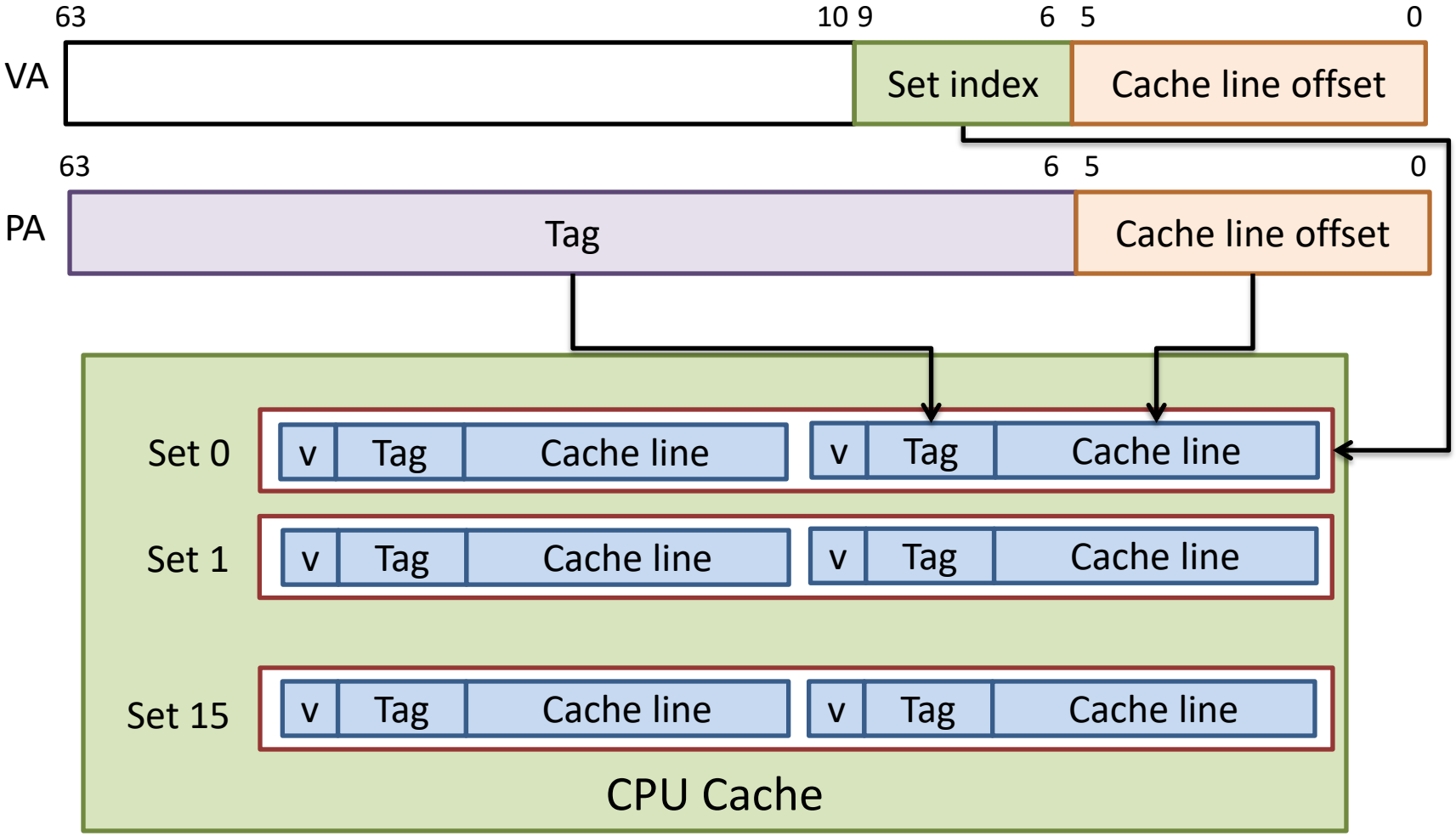
Further Optimizing Memory Access

Virtual Index and Physical Tag

- Use VA to index set, calculate the tag from PA
- Cache set lookup is done in parallel with address translation

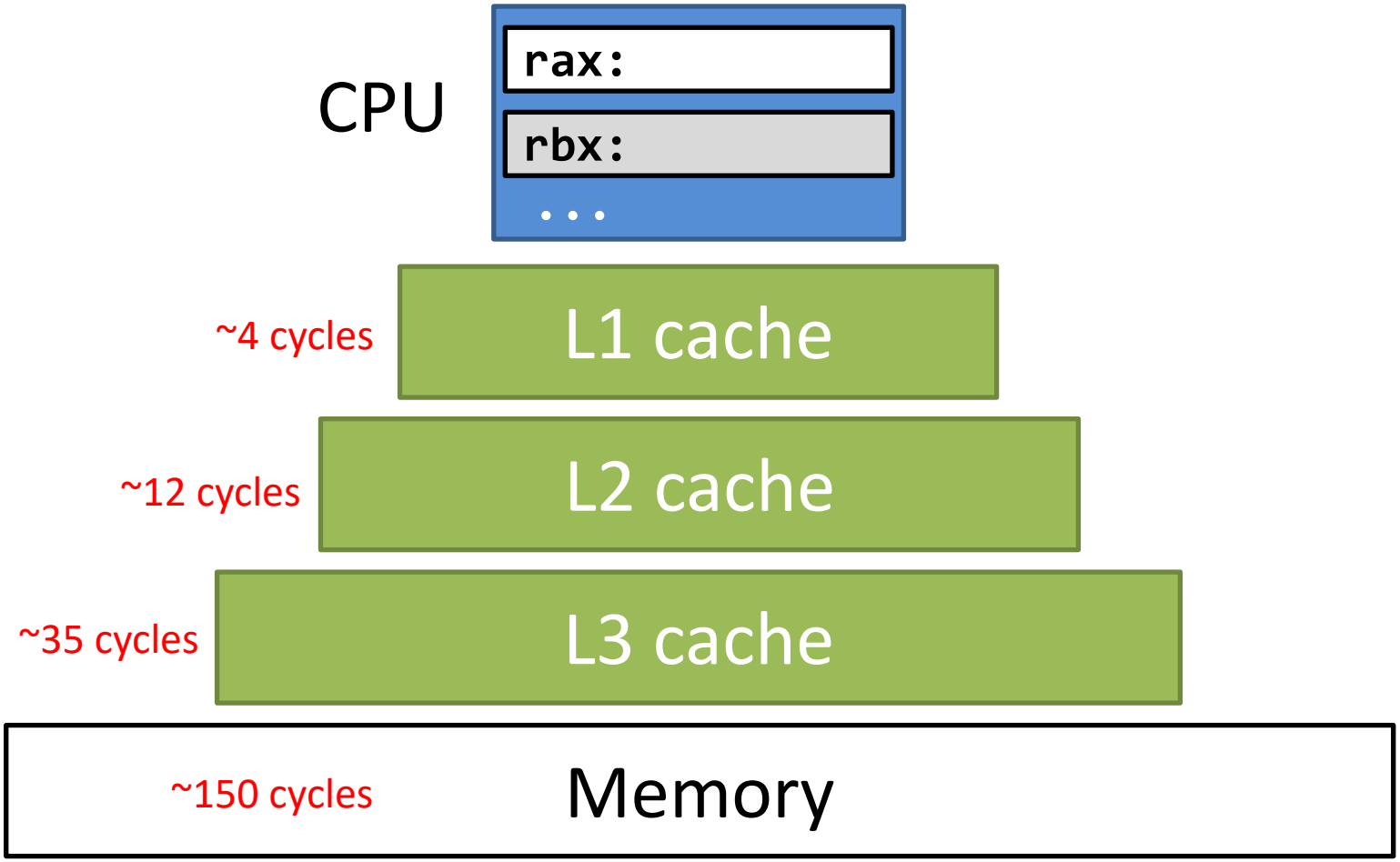


Virtual Index and Physical Tag



2-way set associative cache

Memory Hierarchy



Cache summary

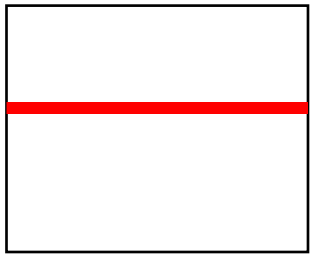
- Caching can speed up memory access
  - L1/L2/L3 cache data
  - TLB caches address translation
- Cache design:
  - Direct mapping
  - Multi-way set associative
  - Virtual index + physical tag parallelize cache access and address translation

Cache-friendly Code

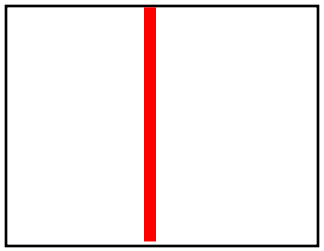
```
int64 sumarrayrows(int64** a,
                   int r, int c) {
  int i, j = 0;
  int64 sum = 0;

  for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
  return sum;
}
```

```
int64 sumarrayrows(int64** a,
                   int r, int c) {
  int i, j = 0;
  int64 sum = 0;

  for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
  return sum;
}
```

Which implementation is more cache friendly?



(i,*)



(*,j)

Cache-friendly Code

```
int64 sumarrayrows(int64** a,
                   int r, int c) {
  int i, j = 0;
  int64 sum = 0;

  for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
  return sum;
}
```

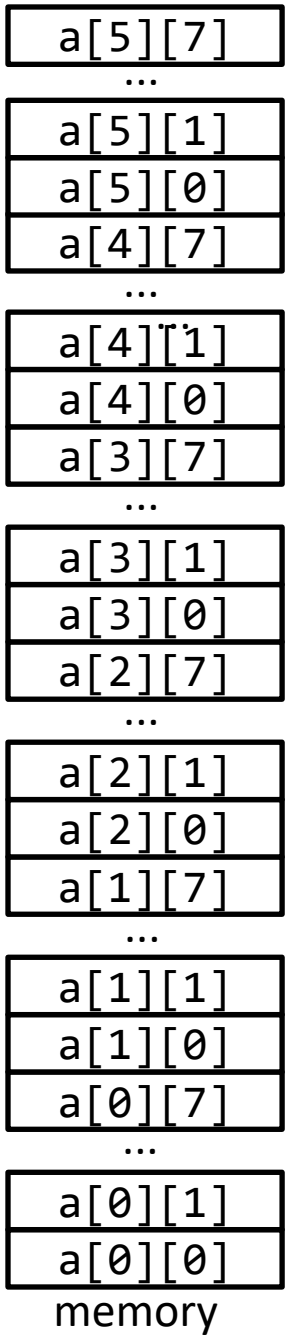
```
int64 sumarrayrows(int64** a,
                   int r, int c) {
  int i, j = 0;
  int64 sum = 0;

  for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
  return sum;
}
```

How many cache misses?

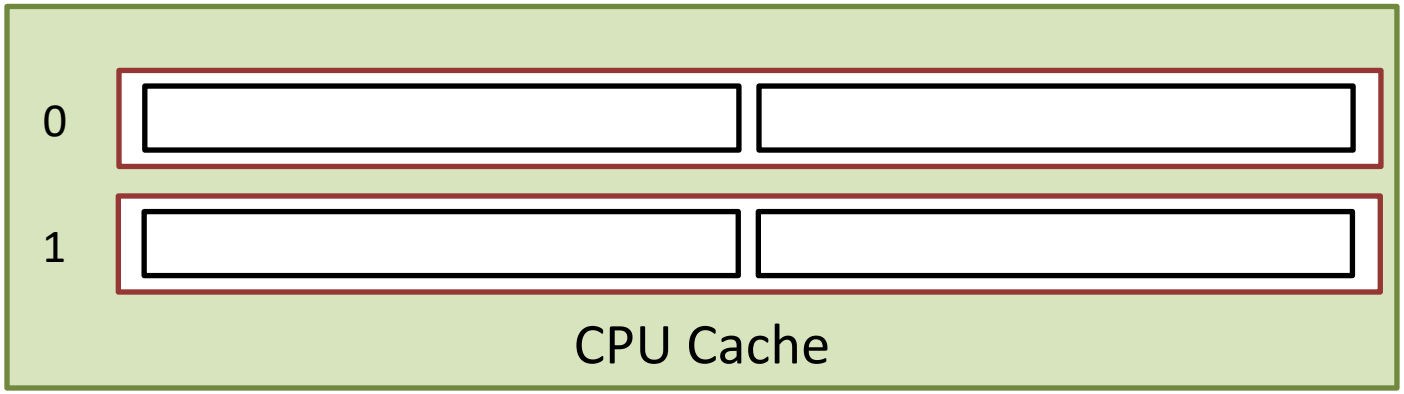
Example:

- CPU Cache: 2-way associative, 2 sets, 64 bytes cache line
- Array: `int64 a[6][8];` address of `a[0][0]` is 64-byte-aligned
- local variables: `i`, `j`, `sum` are stored in the registers

Simple Example



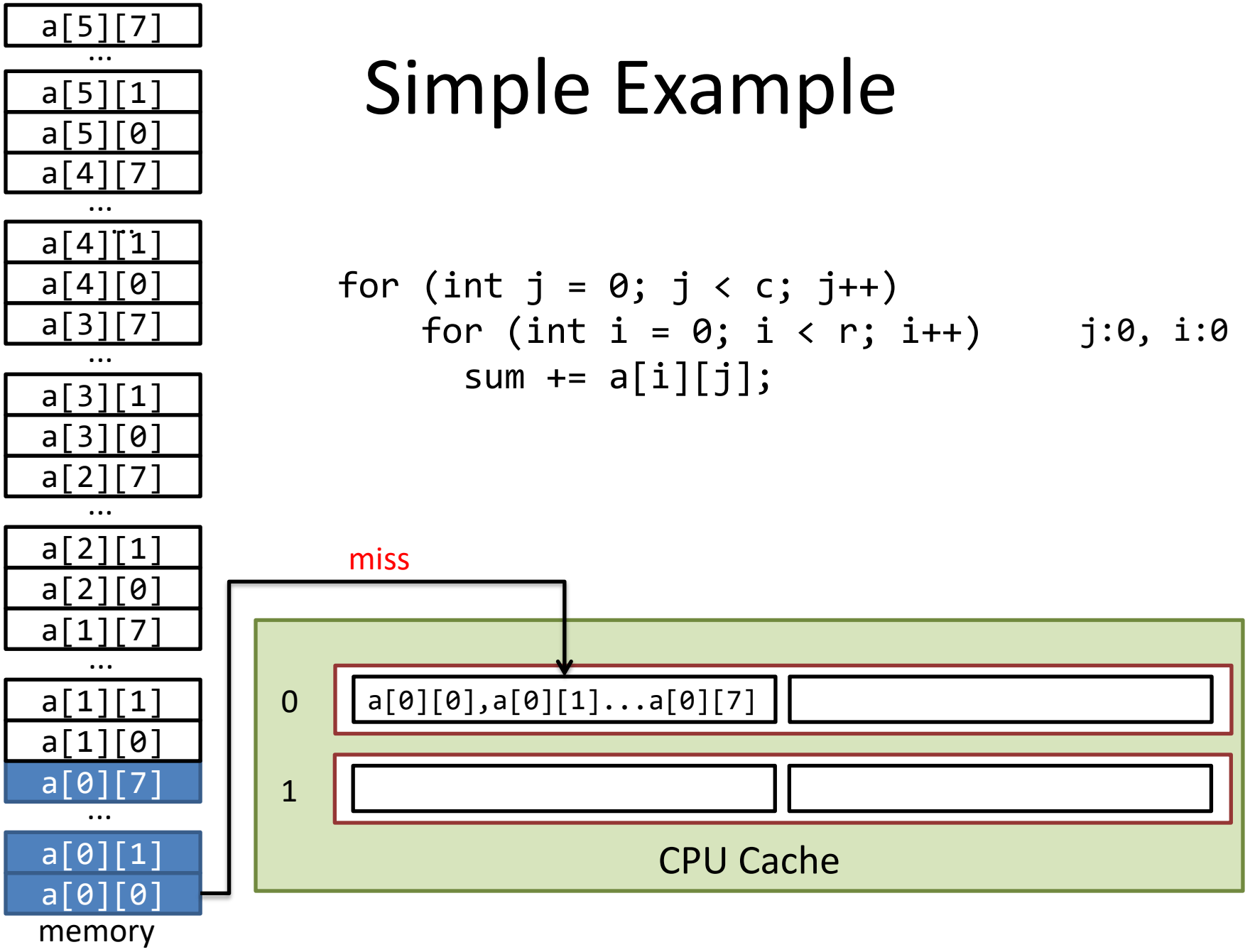

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```



Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

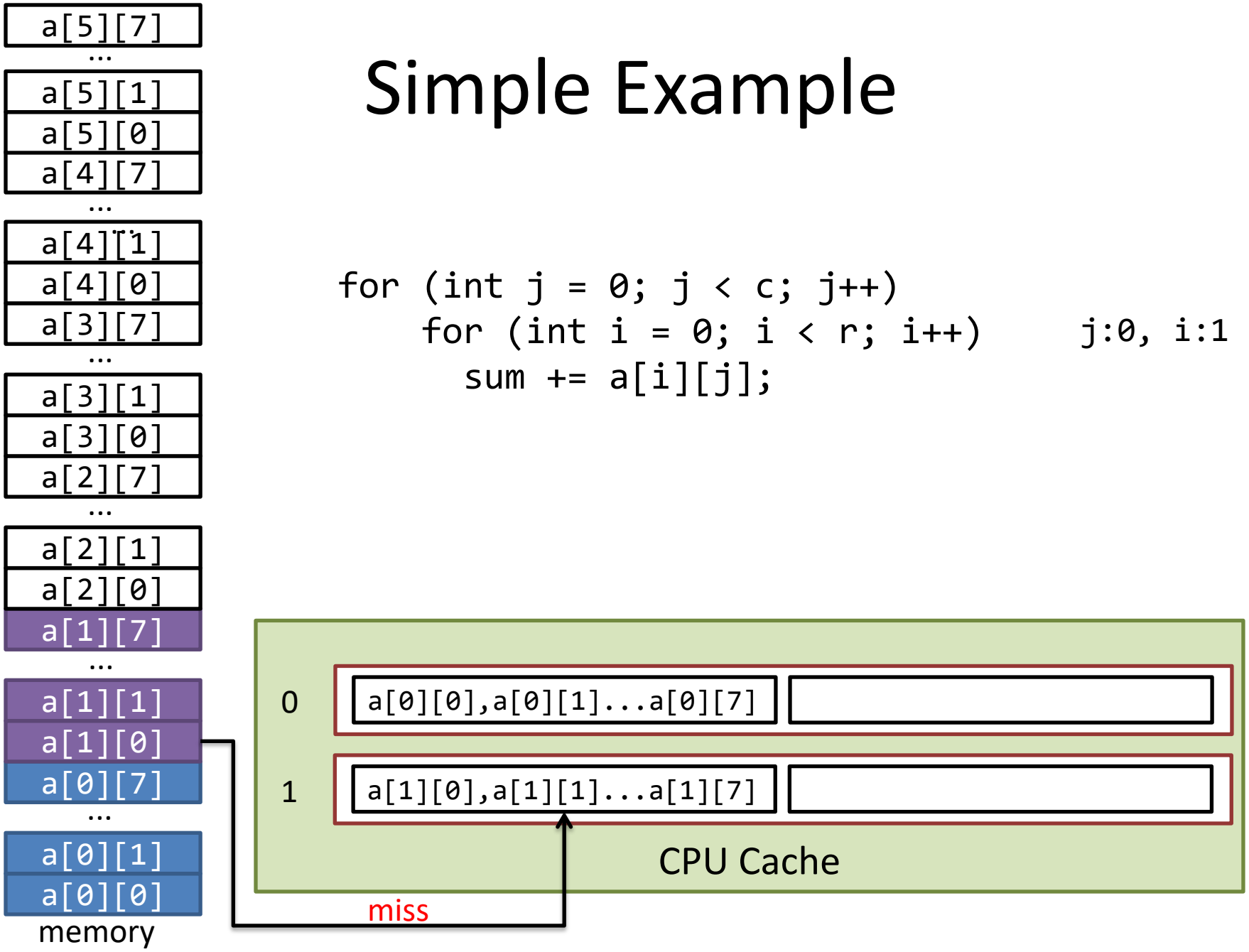
j:0, i:0



Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

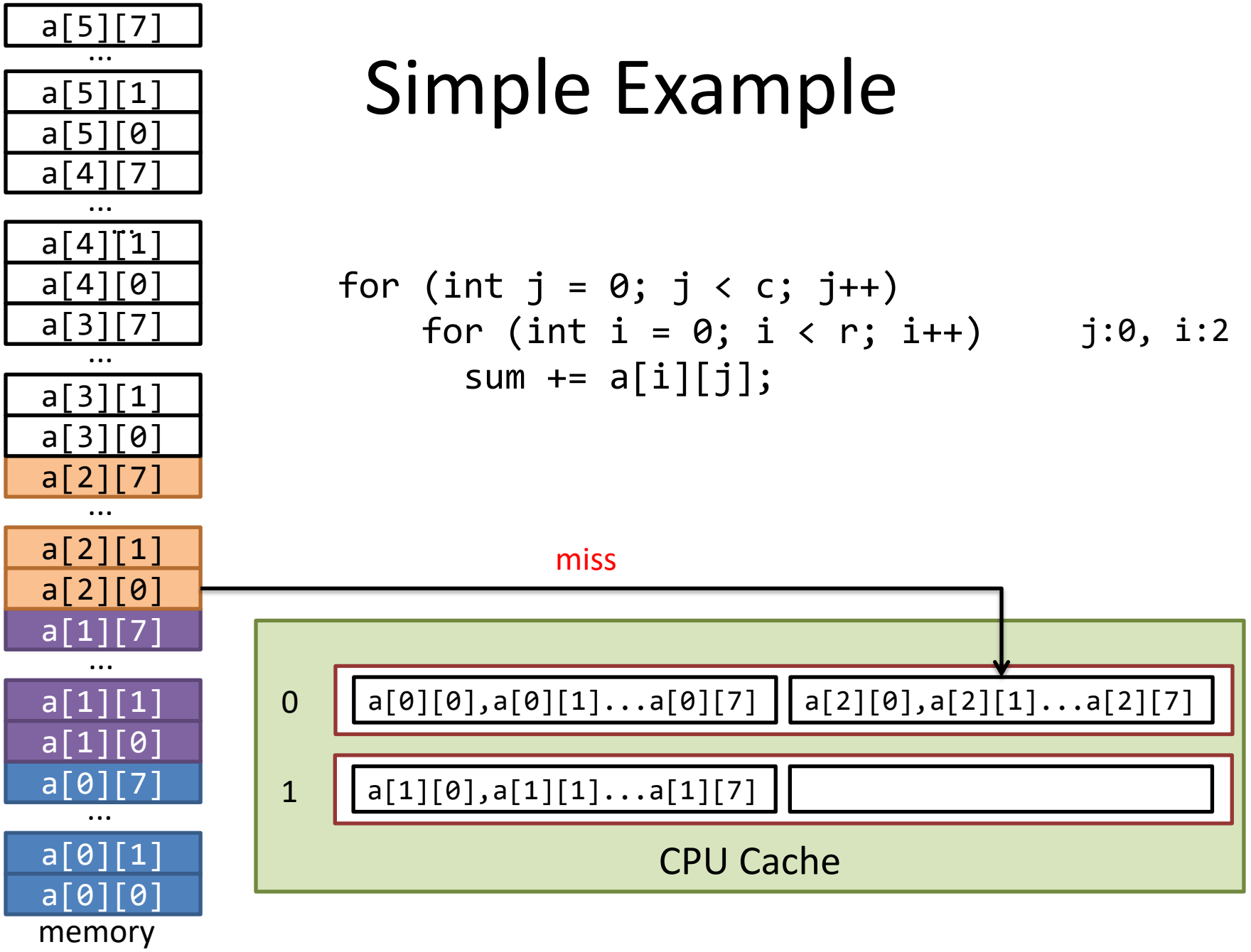
j:0, i:1



Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

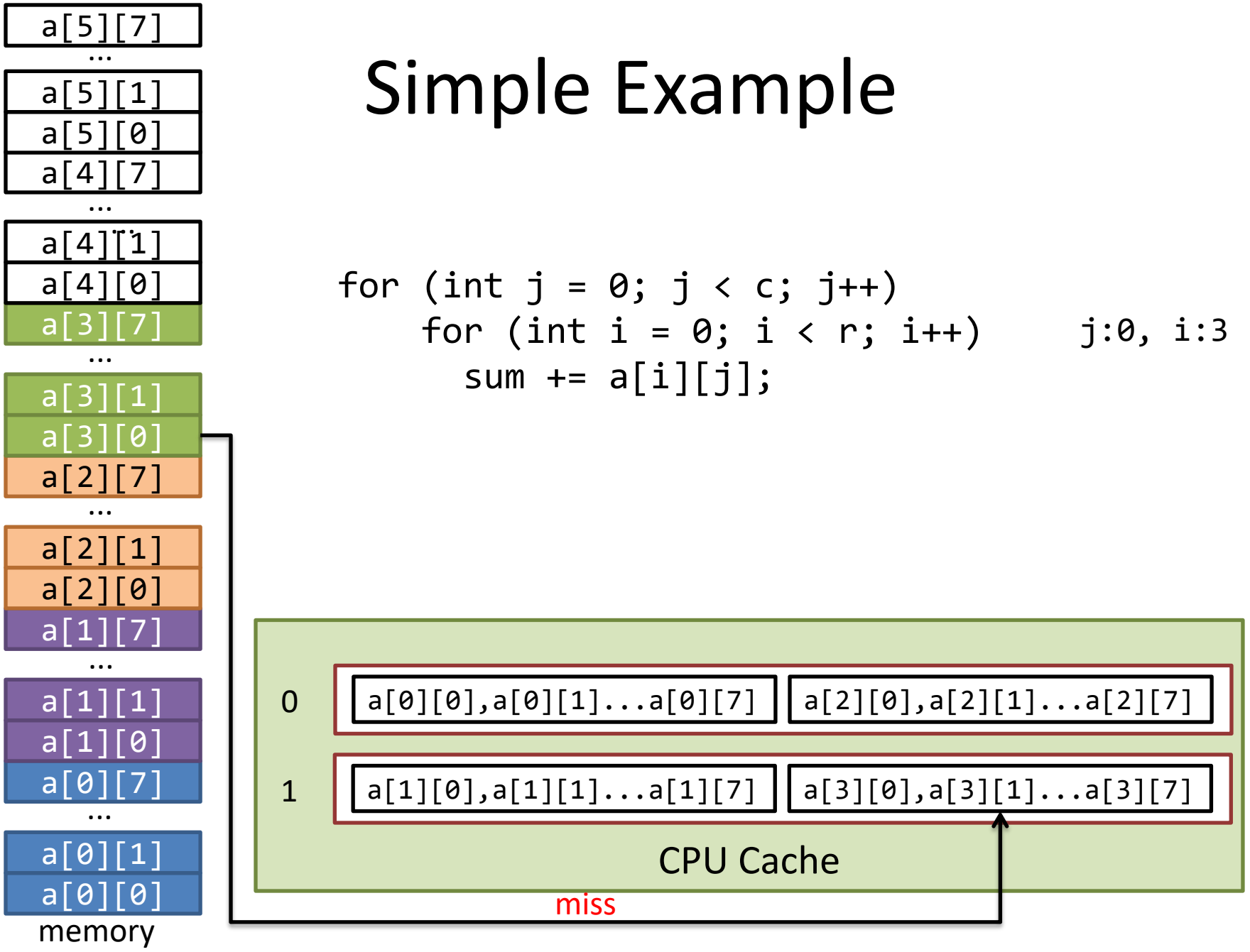
j:0, i:2



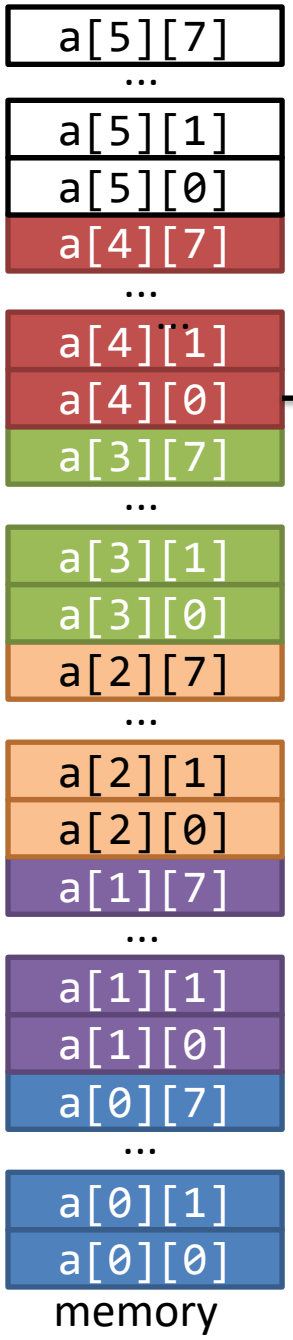
Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

j:0, i:3



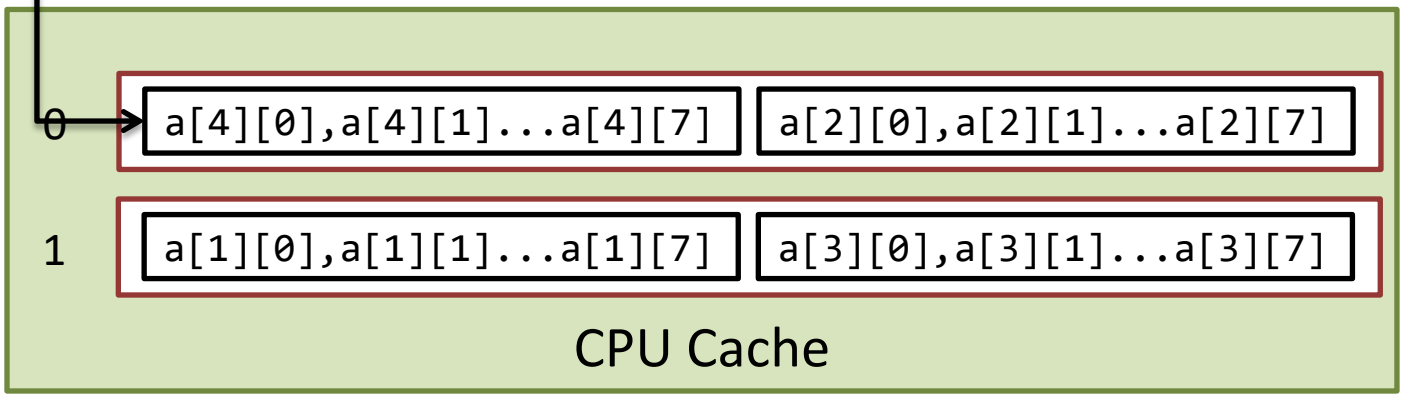
Simple Example



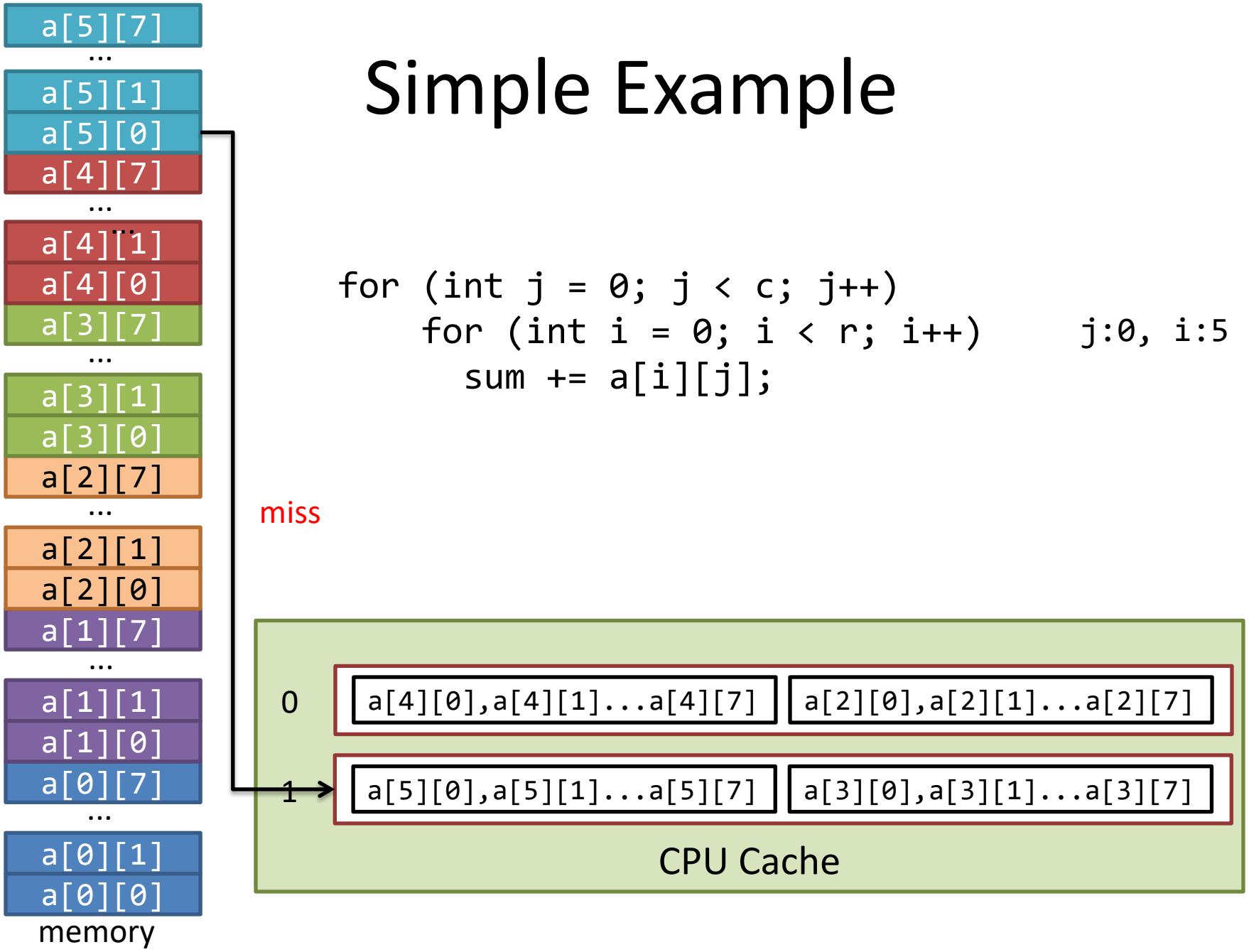

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

j:0, i:4

miss



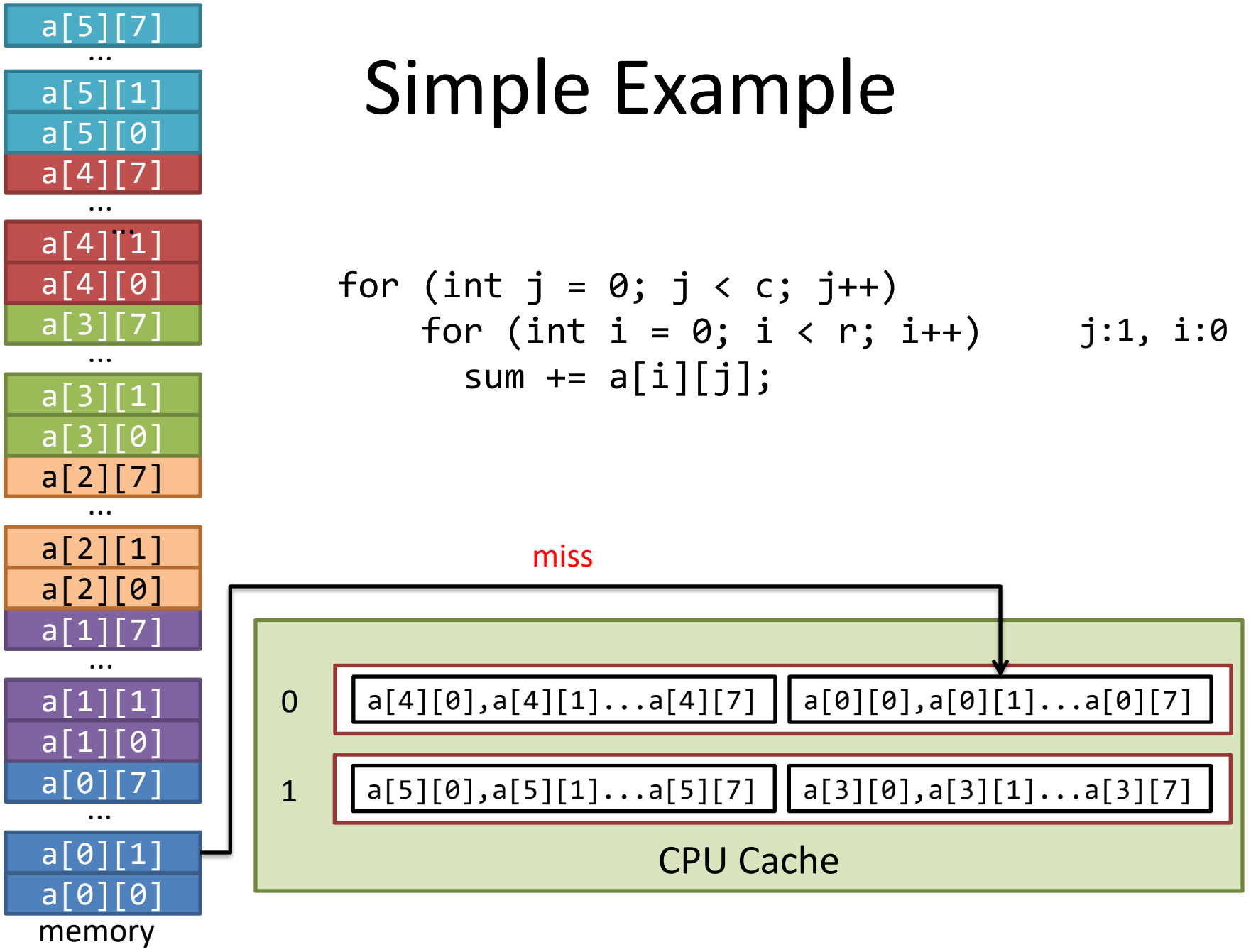
Simple Example



Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

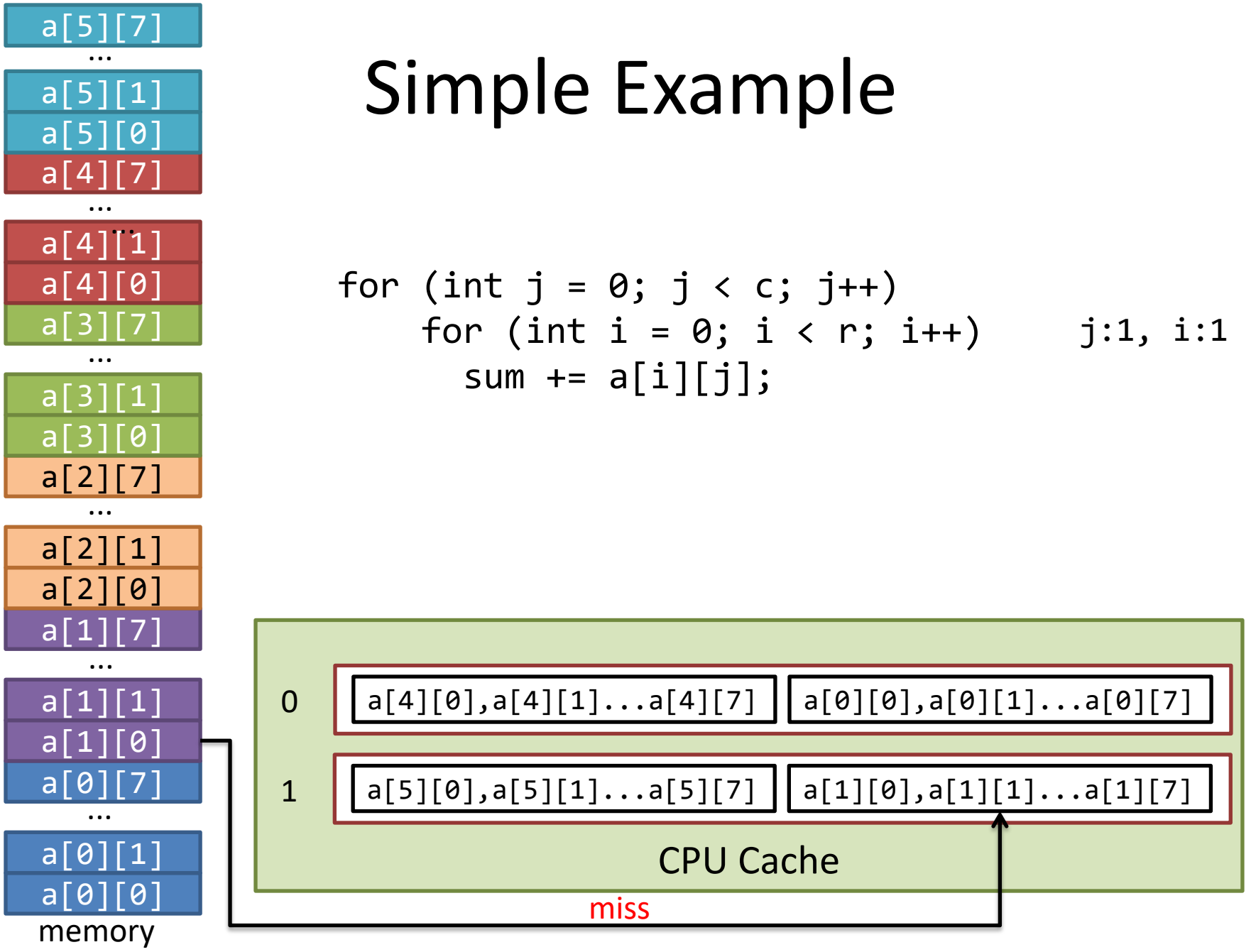
j:1, i:0



Simple Example

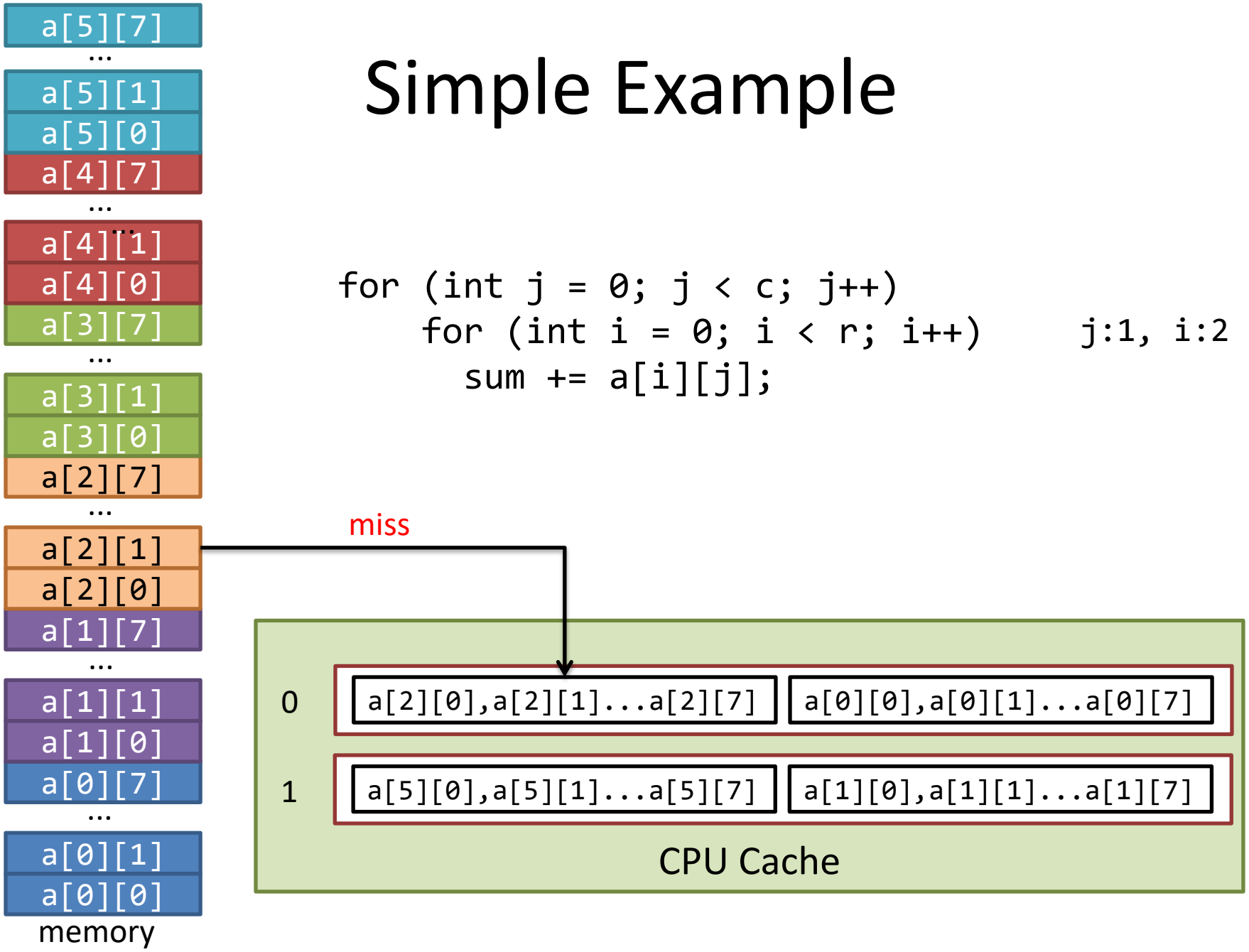
```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

j:1, i:1



Simple Example

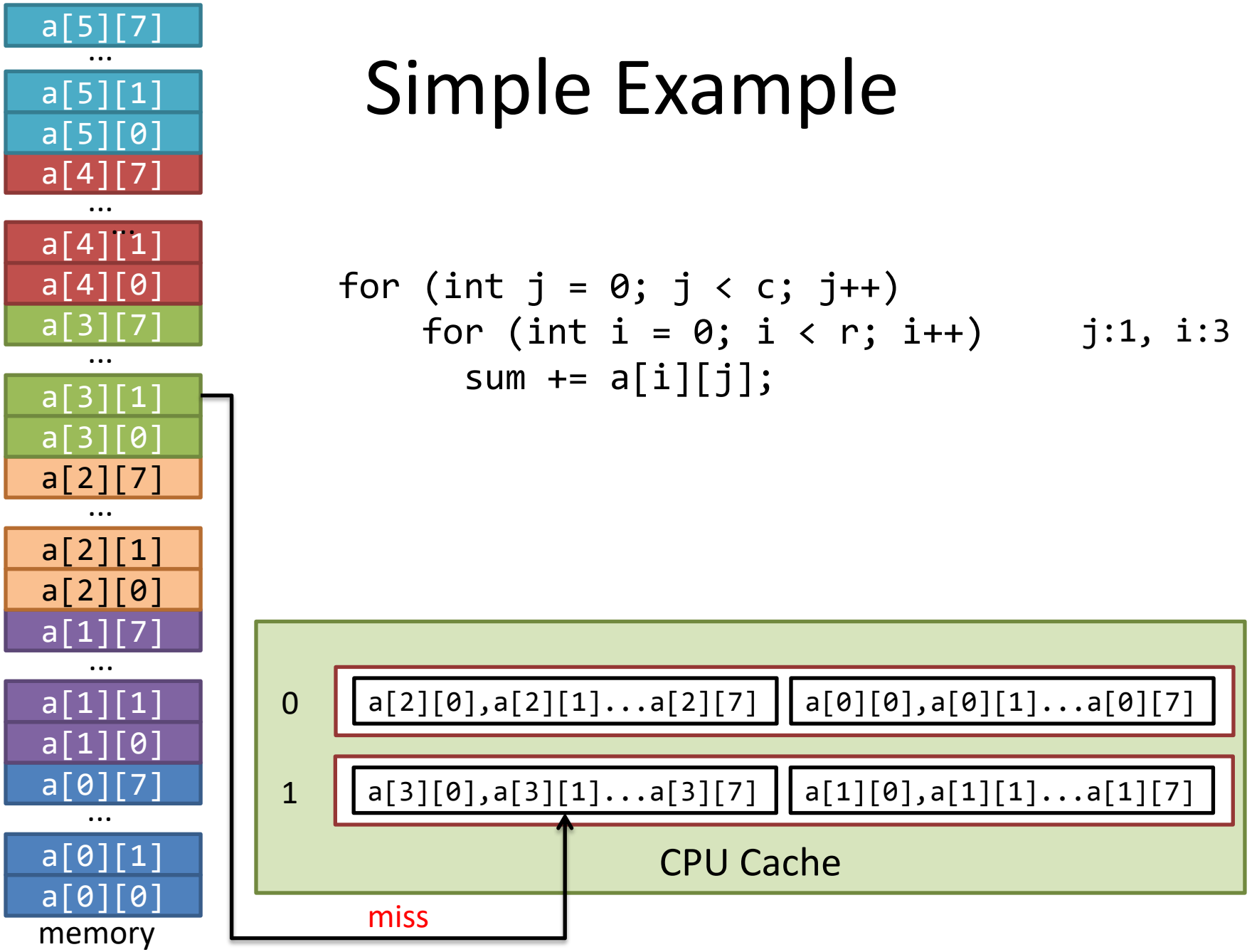
```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)      j:1, i:2
      sum += a[i][j];
```



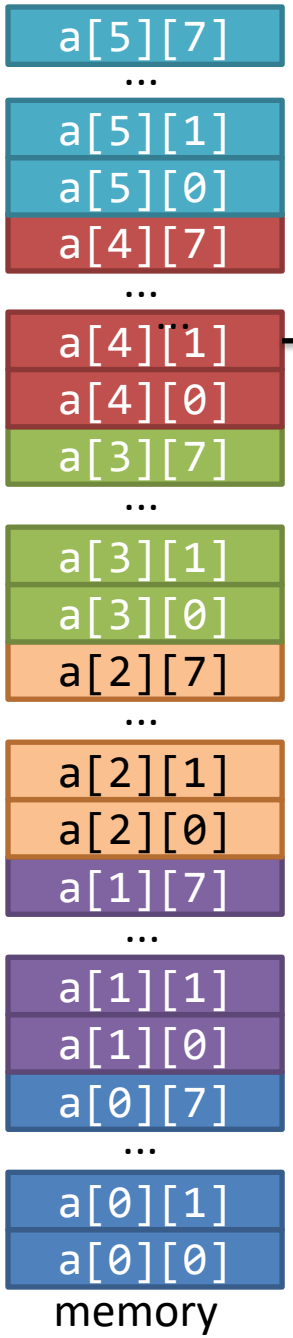
Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

j:1, i:3



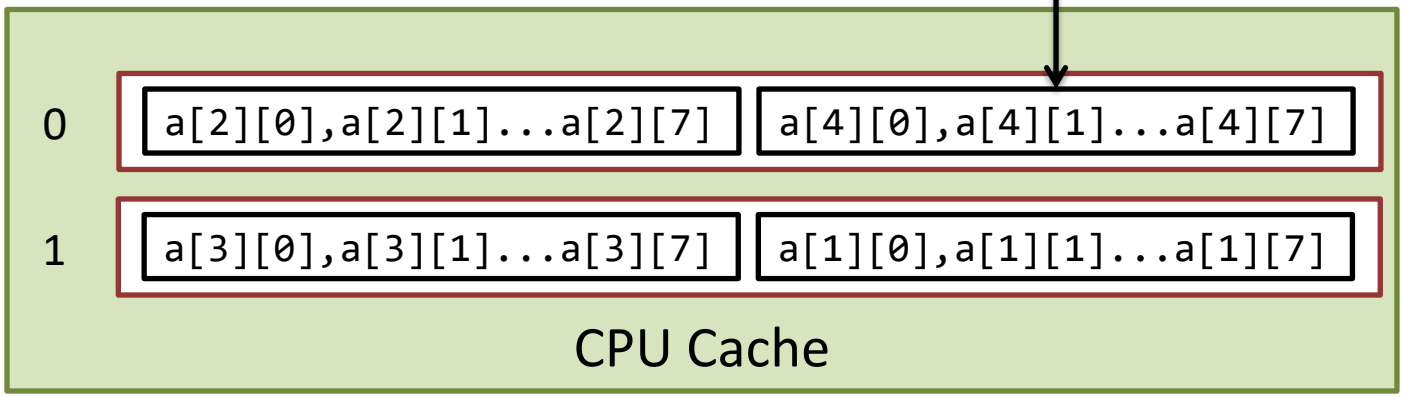
Simple Example



miss

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

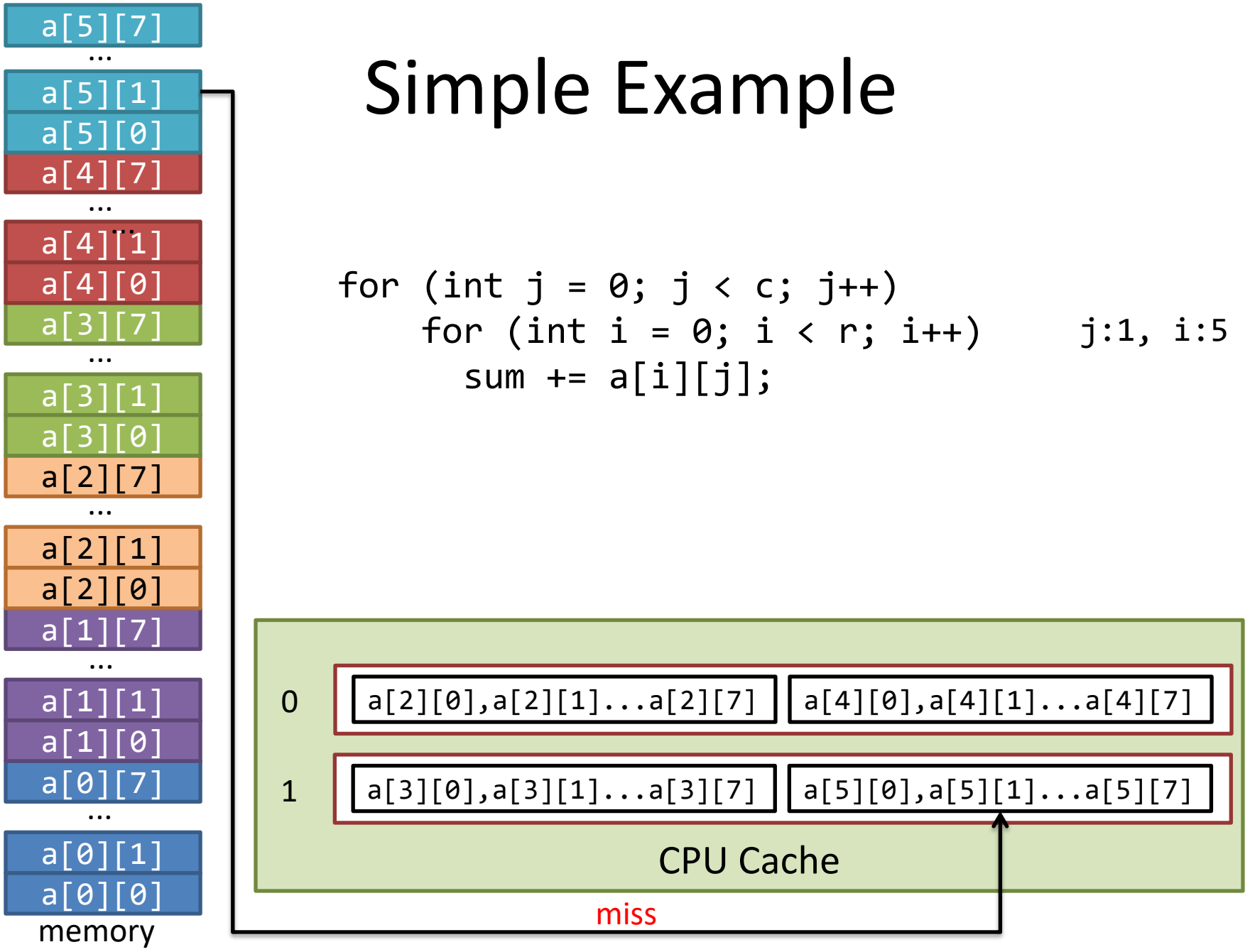
j:1, i:4



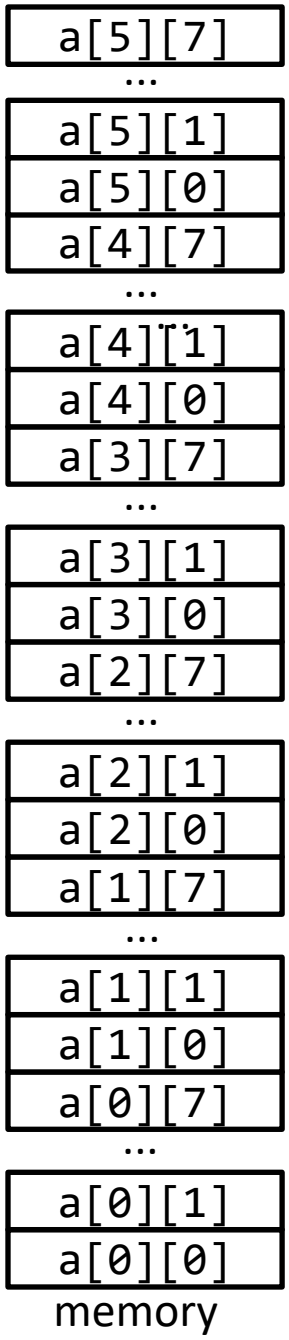
Simple Example

```
for (int j = 0; j < c; j++)
    for (int i = 0; i < r; i++)
      sum += a[i][j];
```

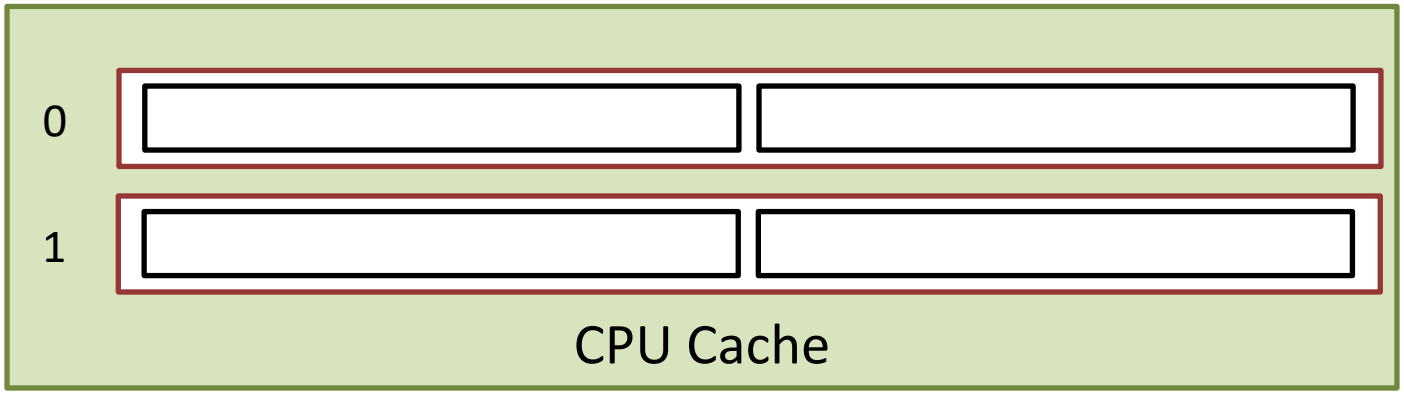
j:1, i:5



Simple Example



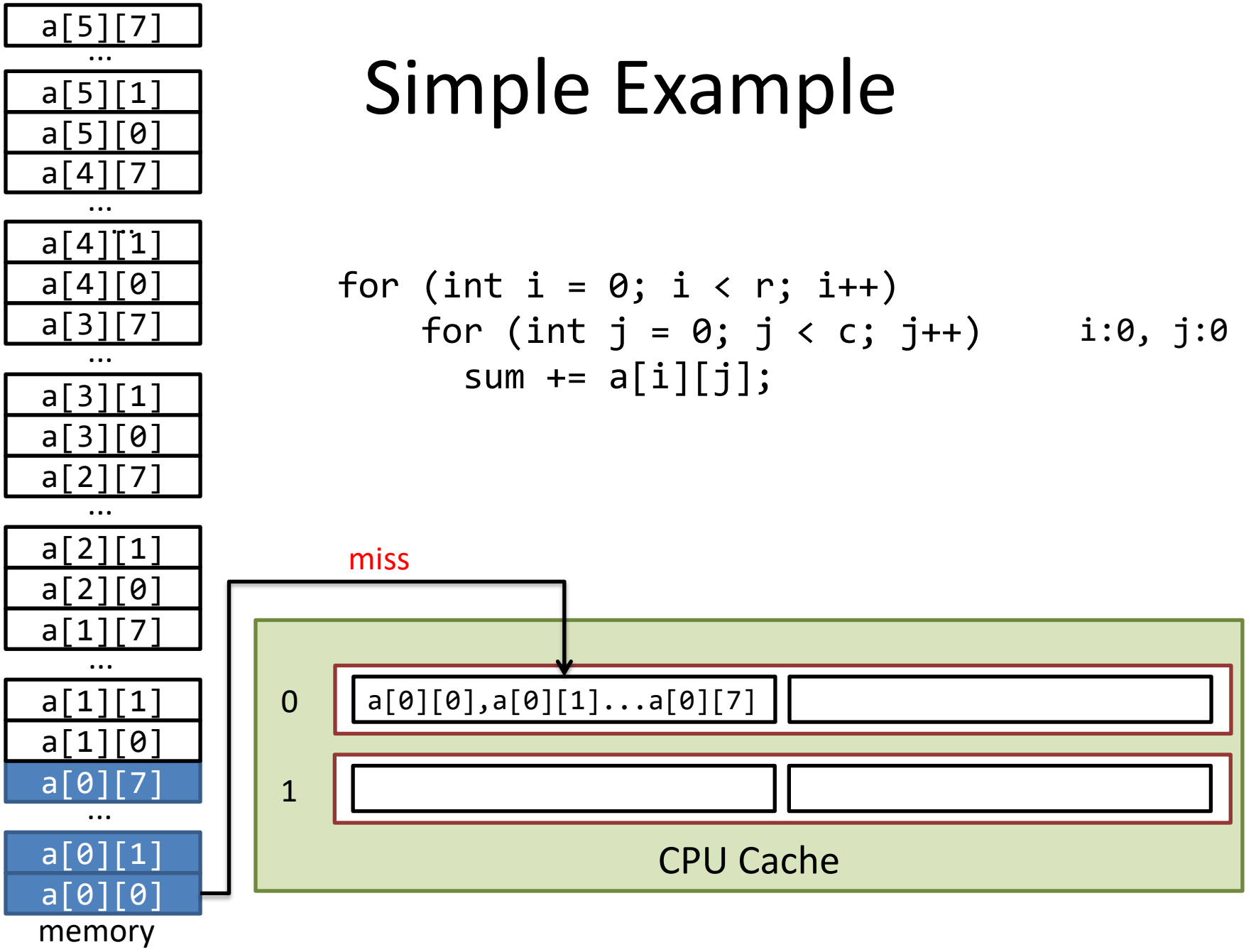

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

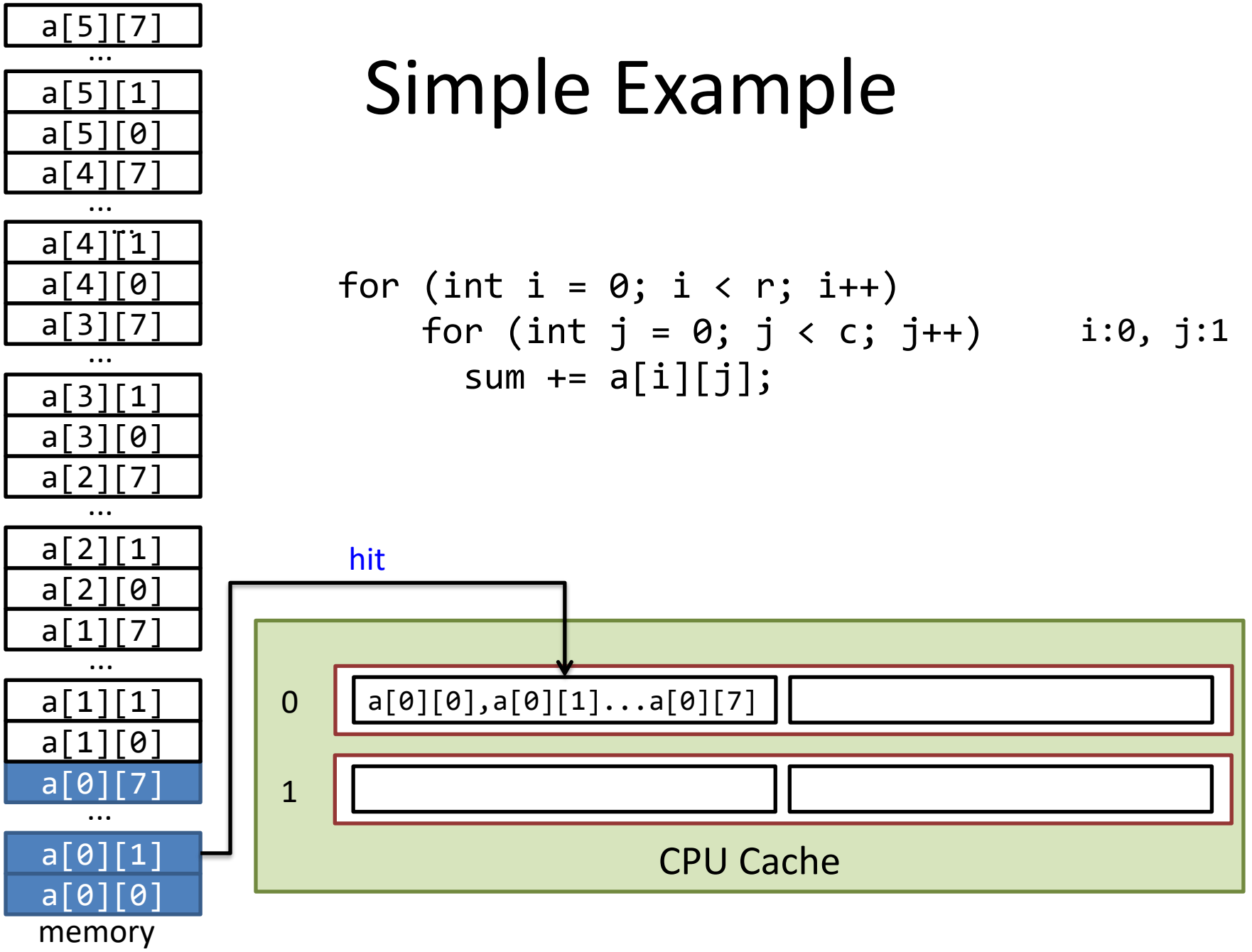
i:0, j:0



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

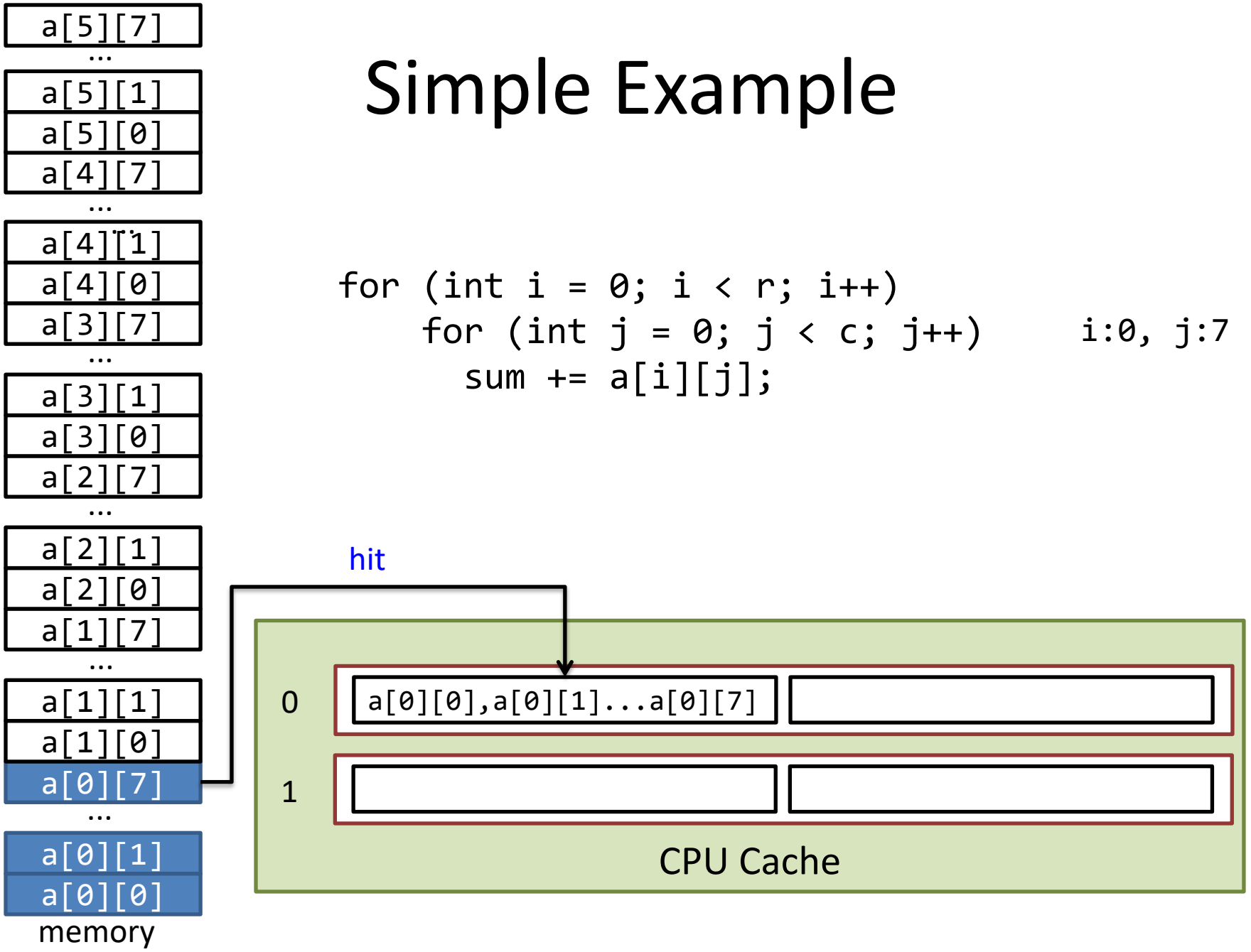
i:0, j:1



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

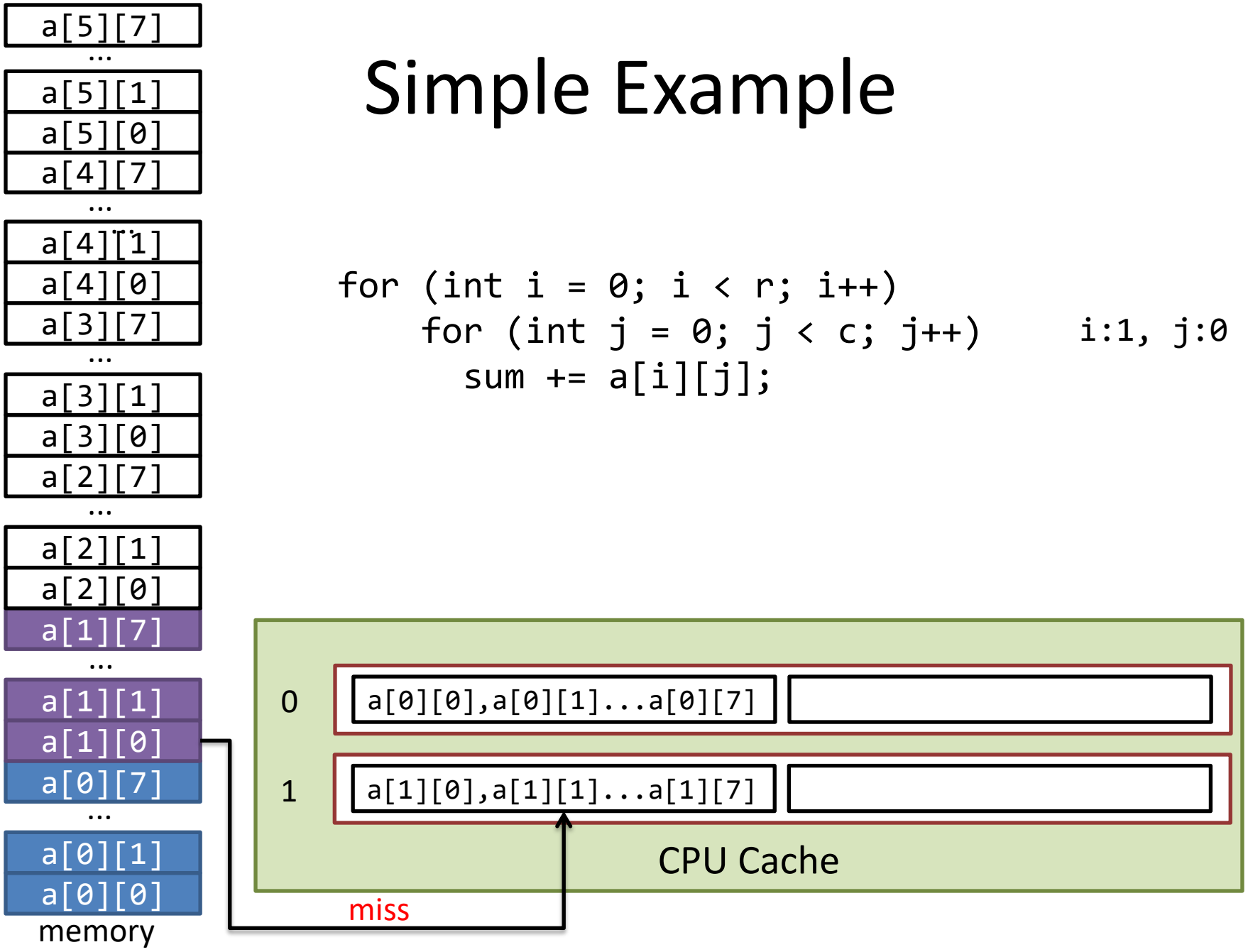
i:0, j:7



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

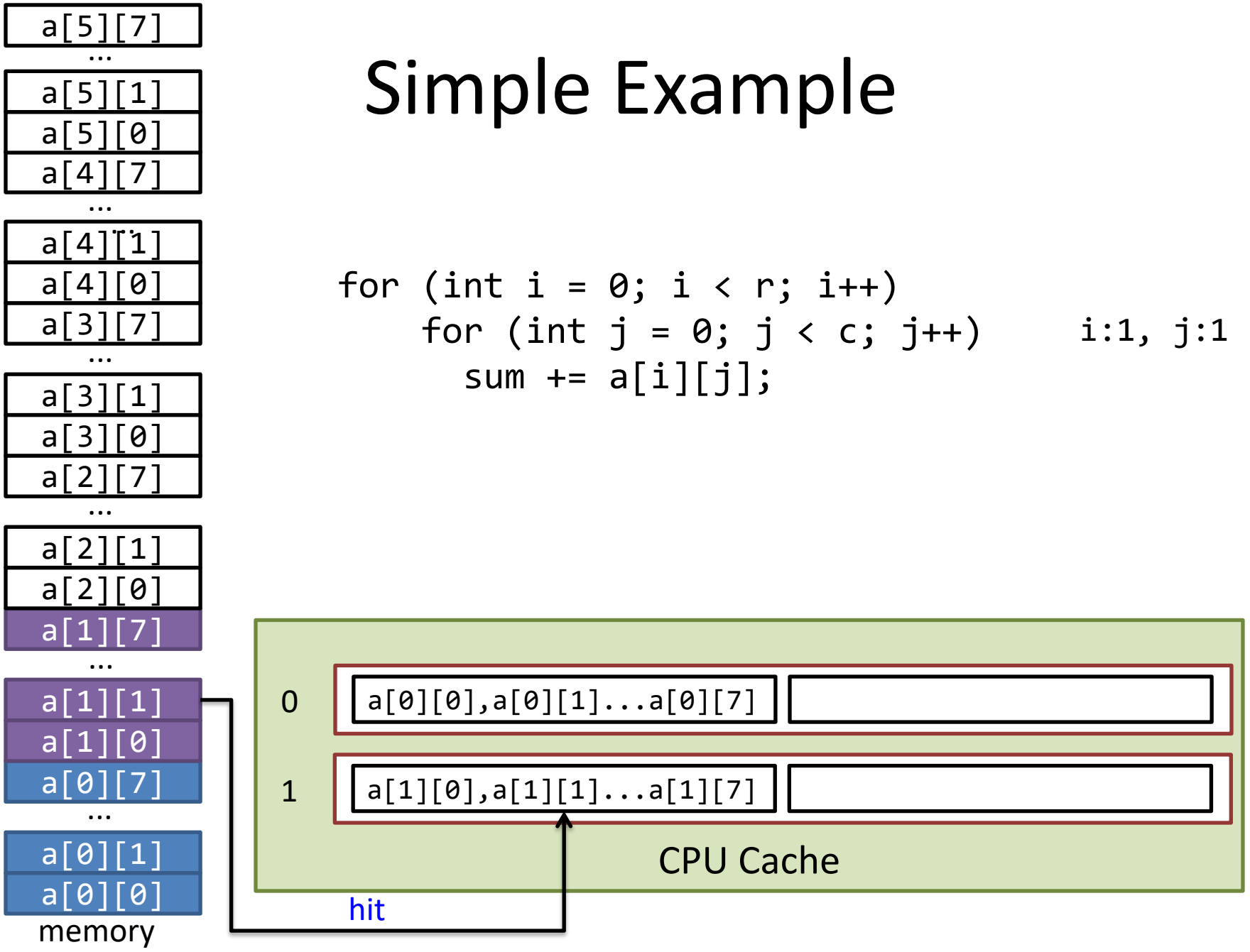
i:1, j:0



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

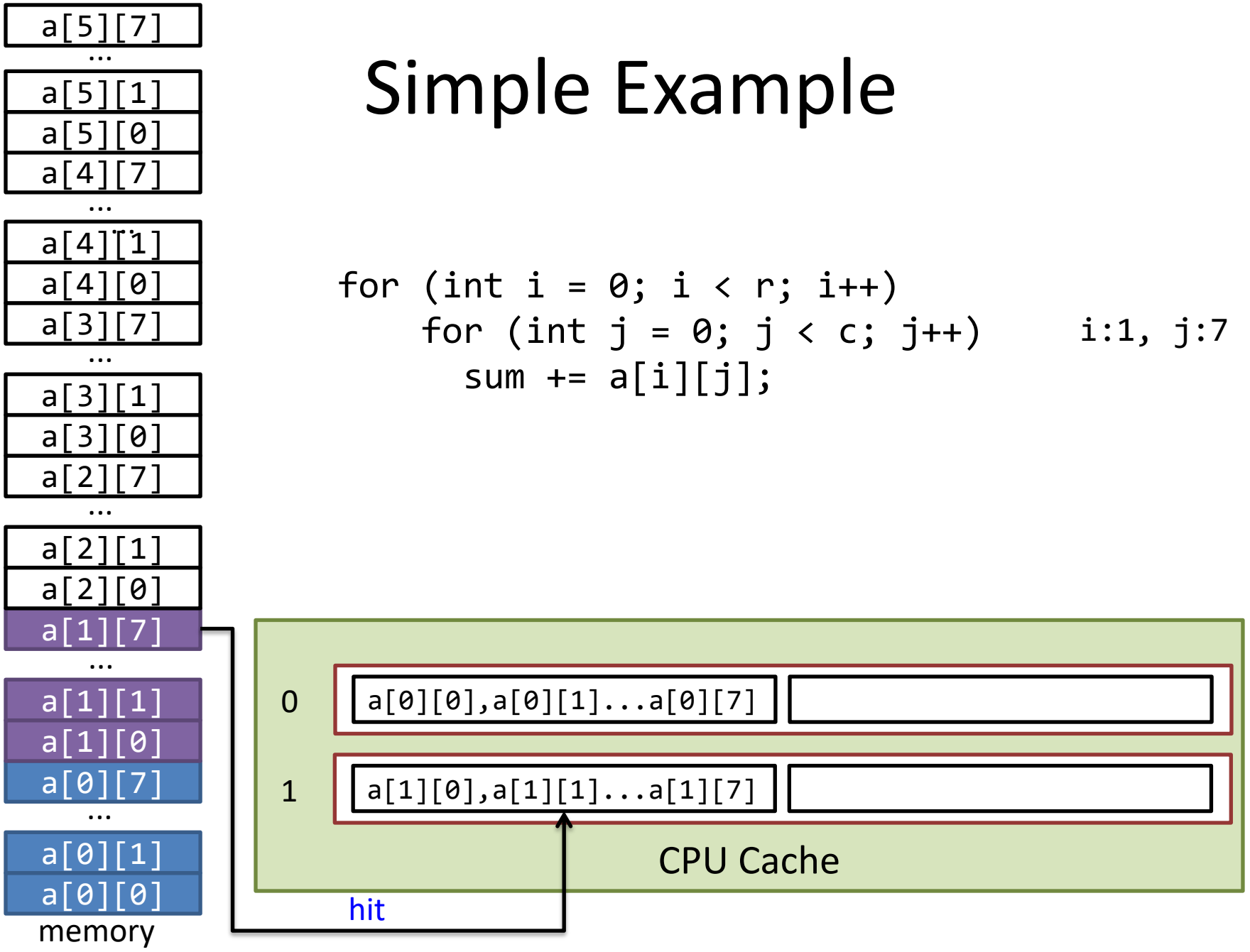
i:1, j:1



Simple Example

```
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
      sum += a[i][j];
```

i:1, j:7



Matrix Multiplication

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```
for (int i=0; i < N; i++) {
  for (int k=0; k < N; k++) {
    for (int j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

MM—ikj

```
for (int i=0; i < N; i++) {
  for (int j=0; j < N; j++) {
    for (int k=0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

MM—ijk

```
for (int k=0; k < N; k++) {
  for (int j=0; j < N; j++) {
    for (int i=0; i < N; i++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

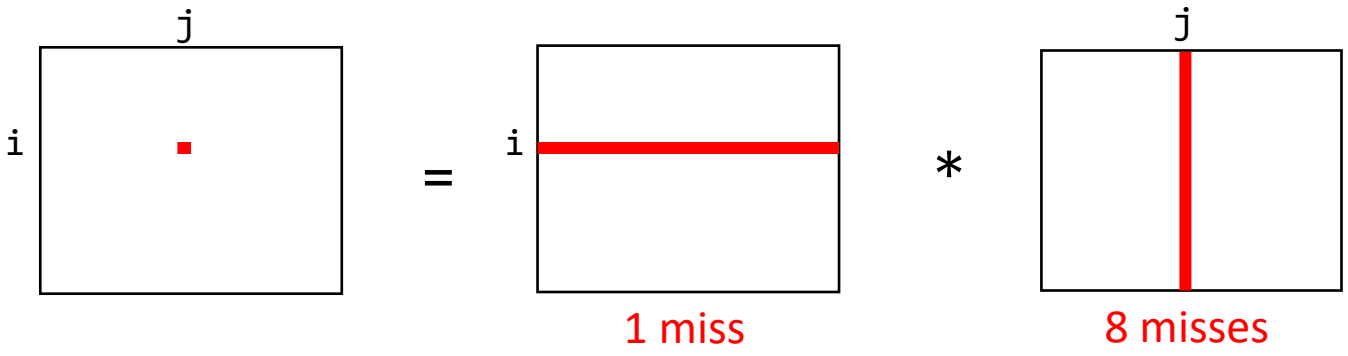
MM—kji

Which one is cache friendly, which one is worst?

Matrix Multiplication (ijk)

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

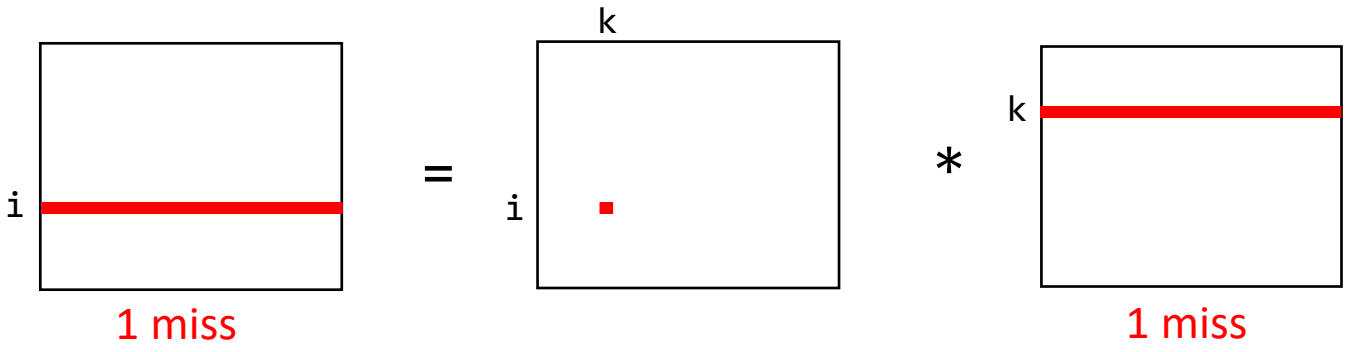
```
for (int i=0; i < N; i++) {
  for (int j=0; j < N; j++) {
    for (int k=0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```



Matrix Multiplication (ikj)

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```
for (int i=0; i < N; i++) {
  for (int k=0; k < N; k++) {
    for (int j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```



Matrix Multiplication (kji)

Cache: 2-way, 2 sets, 64B cache line
Matrix A, B, C: double[8][8]
1st elements of A,B,C, are 64B-aligned

```
for (int k=0; k < N; k++) {
  for (int j=0; j < N; j++) {
    for (int i=0; i < N; i++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```