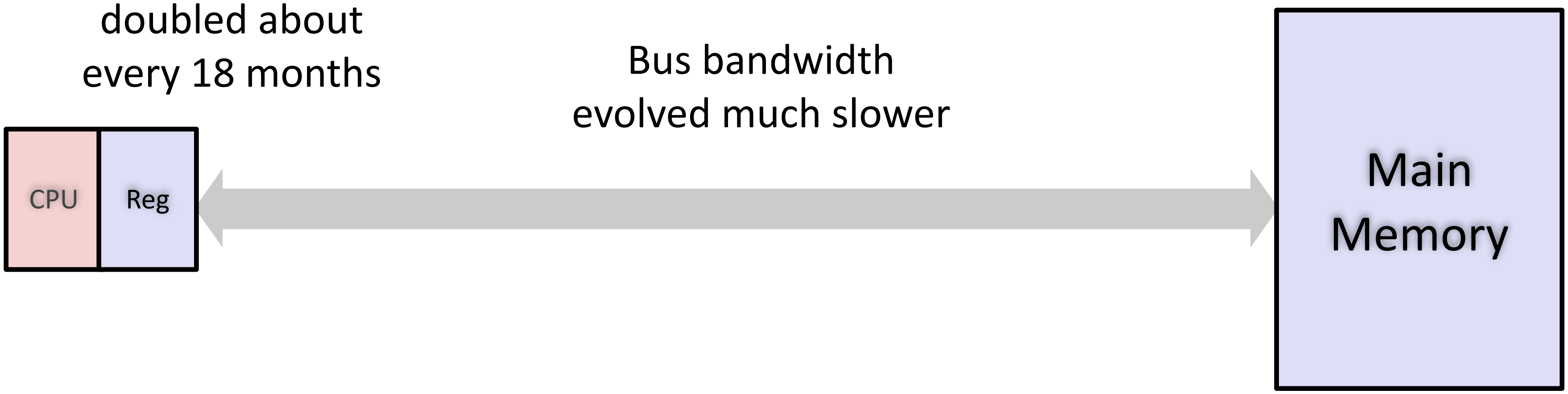
# MEMORY/ CACHES

CAS CS210

6.1.1, 6.1.4 and 6.2--6.5

# Problem: Processor-Memory Bottleneck



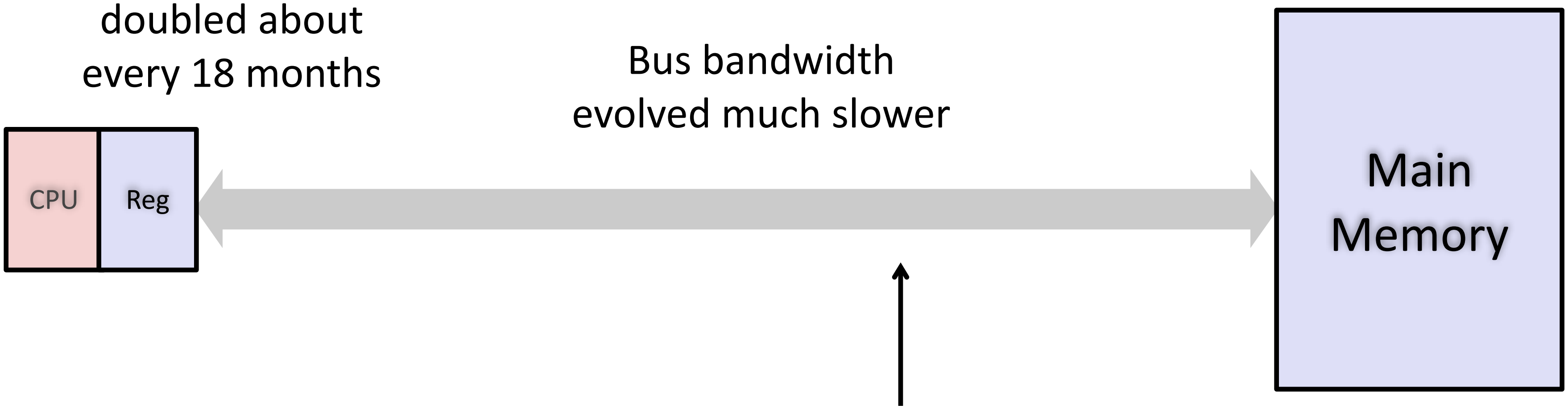
Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

# Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months

Bus bandwidth
evolved much slower



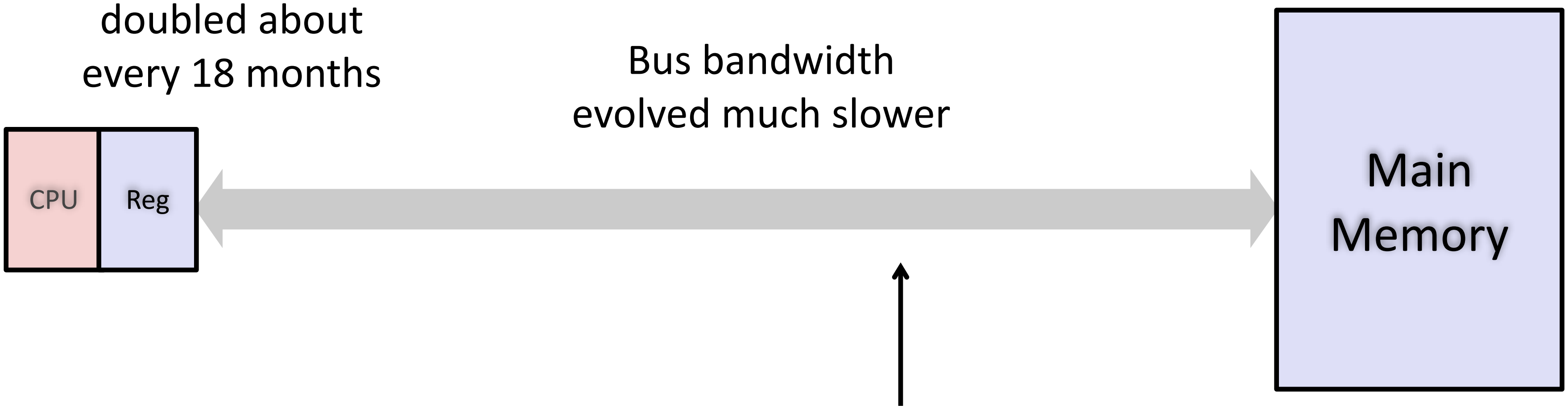
*Core 2 Duo:*
Can process at least
256 Bytes/cycle
(1 SSE two operand add and mult)

*Core 2 Duo:*
Bandwidth
2 Bytes/cycle
Latency
100 cycles

# Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months

Bus bandwidth
evolved much slower



*Core 2 Duo:*
Can process at least
256 Bytes/cycle
(1 SSE two operand add and mult)

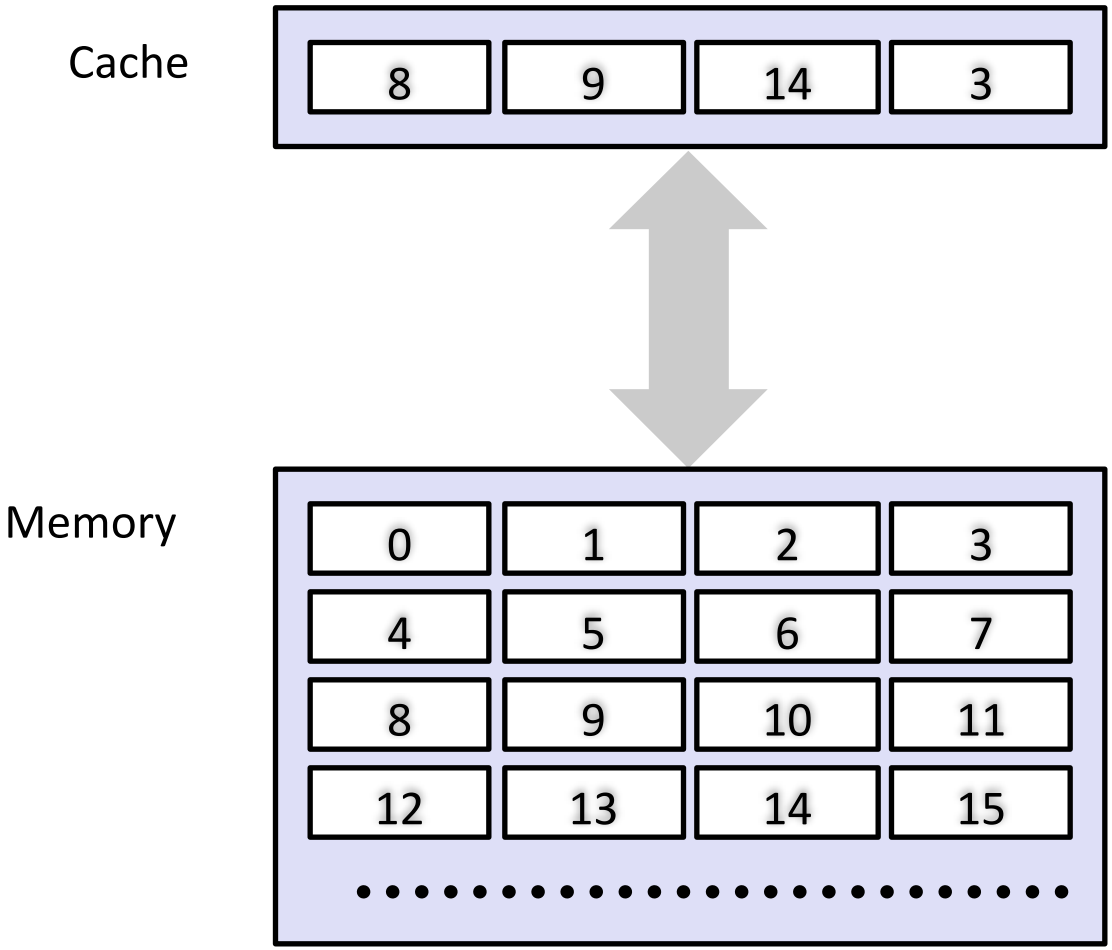
*Core 2 Duo:*
Bandwidth
2 Bytes/cycle
Latency
100 cycles

*Solution: Caches*

# Cache

- **Definition:** Computer memory with short access time used for the storage of frequently or recently used instructions or data

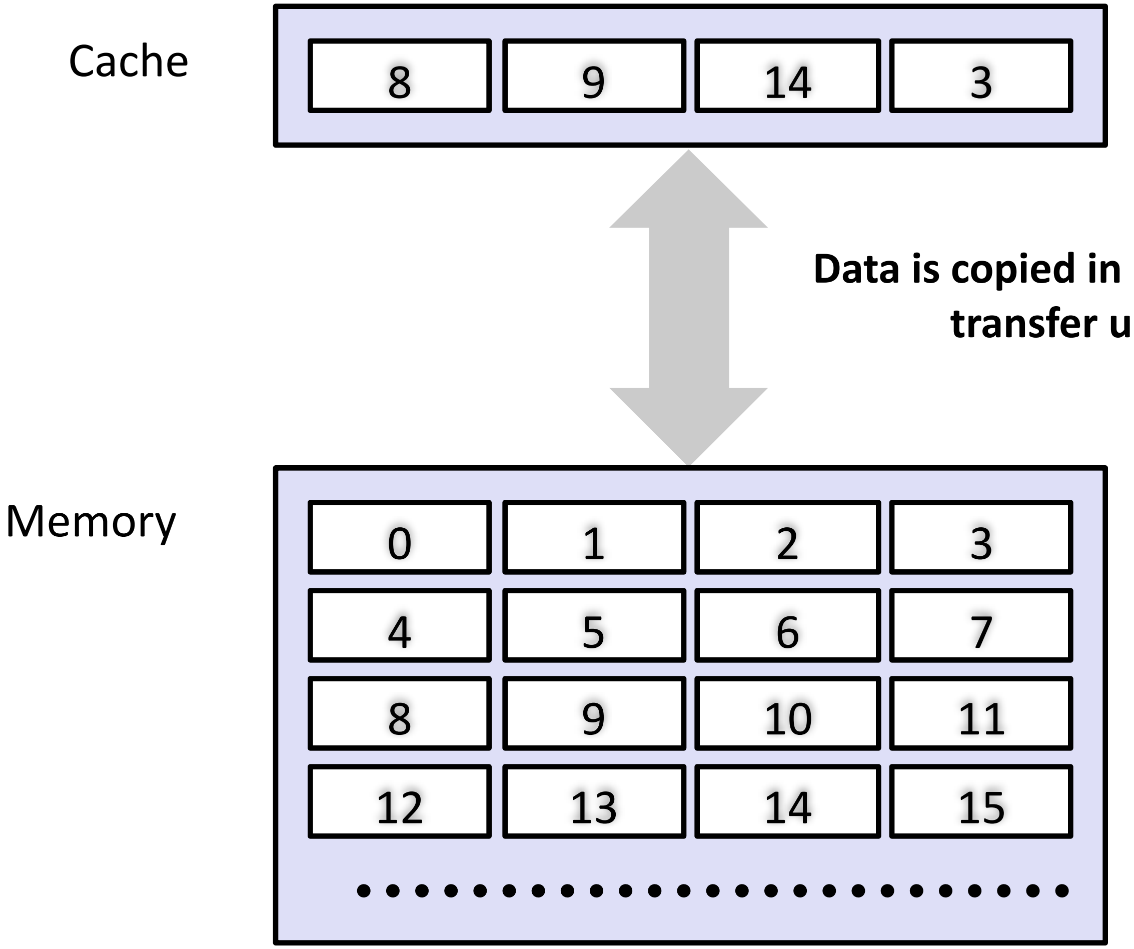
# General Cache Mechanics



**Smaller, faster, more expensive memory caches a subset of the blocks**

**Larger, slower, cheaper memory viewed as partitioned into “blocks”**

# General Cache Mechanics

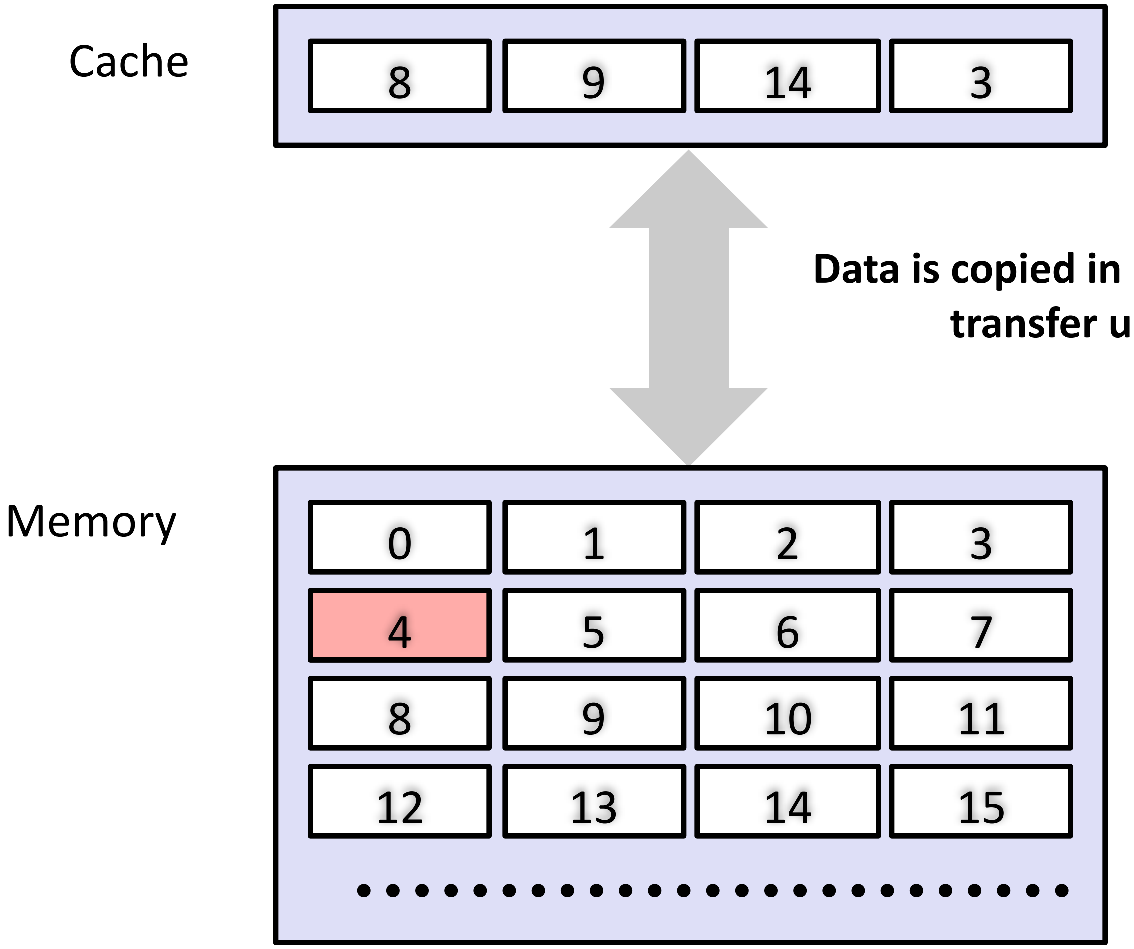


**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into “blocks”**

# General Cache Mechanics



Cache

|  |  |  |  |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

Smaller, faster, more expensive memory caches a  subset of the blocks

Data is copied in block-sized transfer units

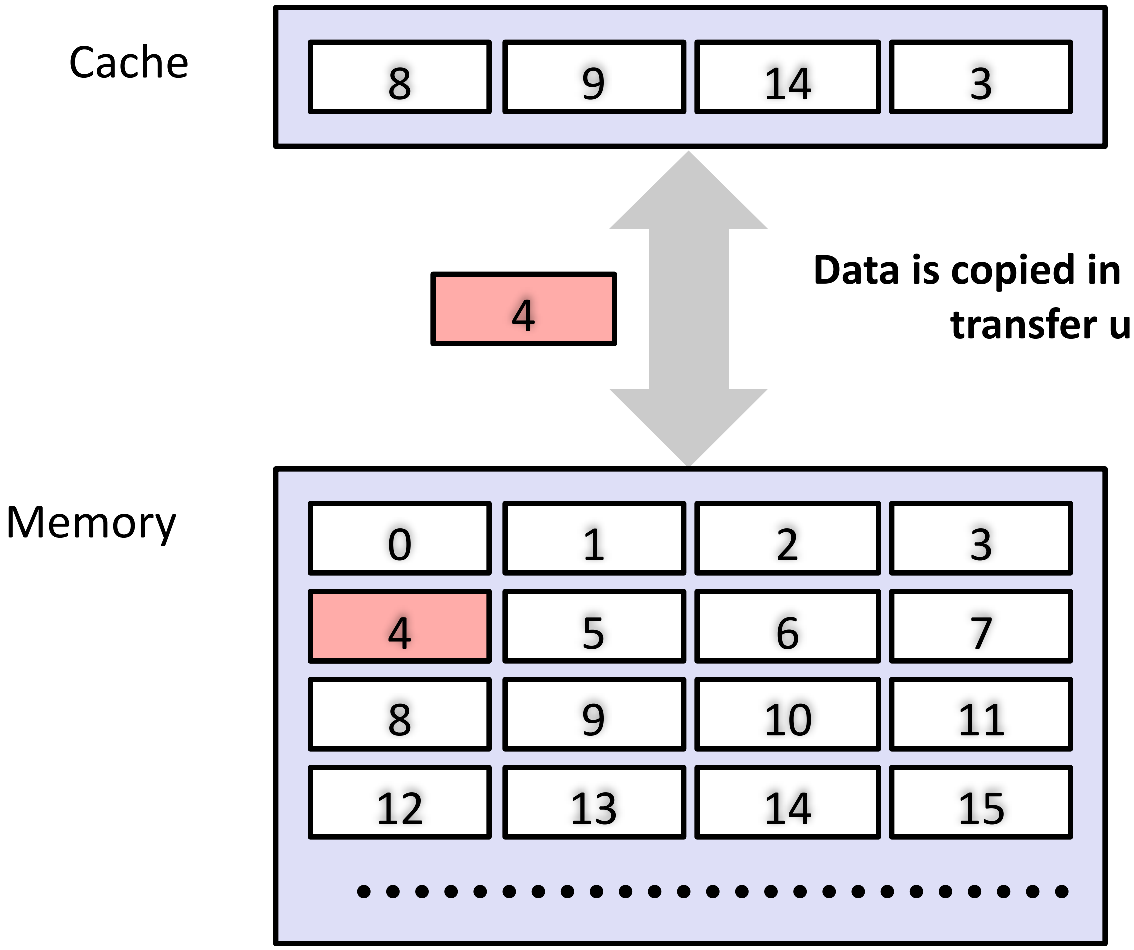
Memory

|  |  |  |  |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

• • • • • • • • • • • • • • • • • • • • • • • • •

Larger, slower, cheaper memory viewed as partitioned into “blocks”

# General Cache Mechanics

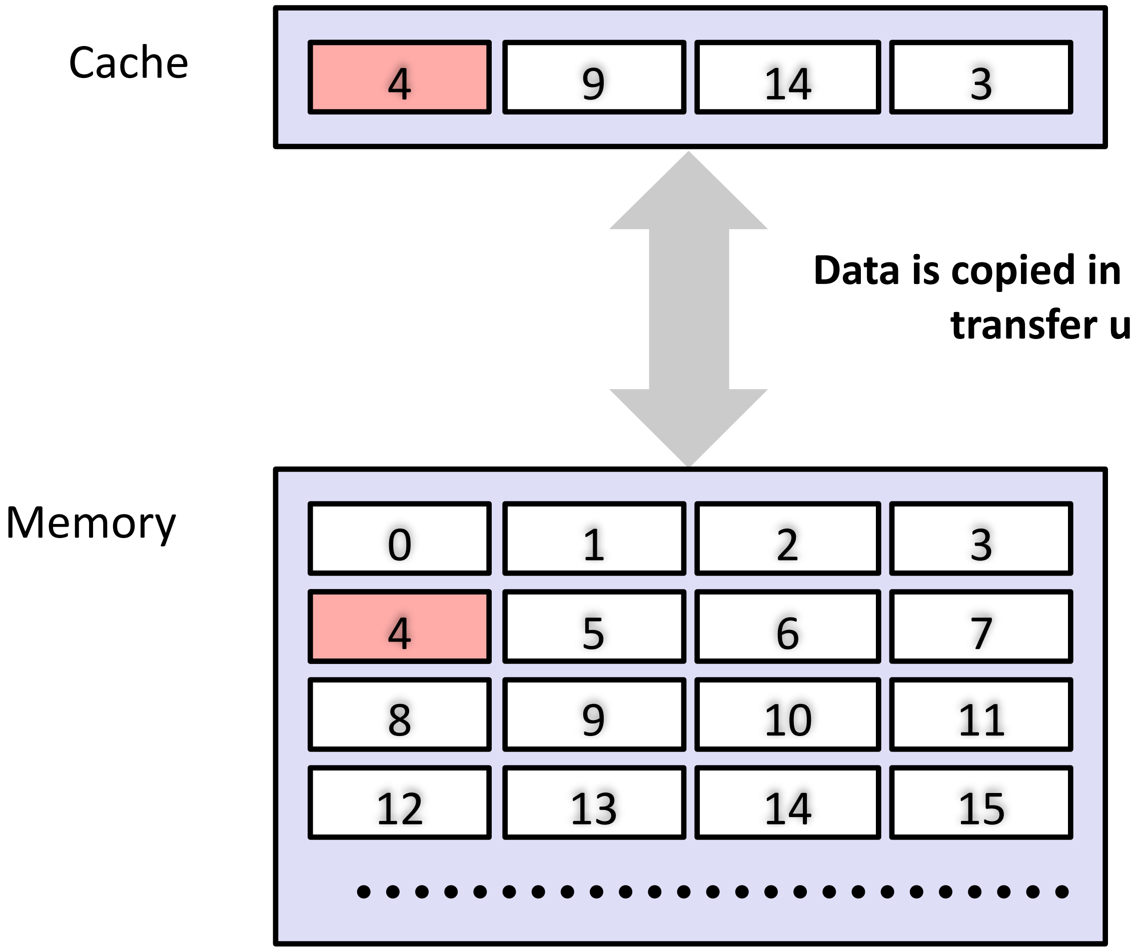


**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into “blocks”**

# General Cache Mechanics

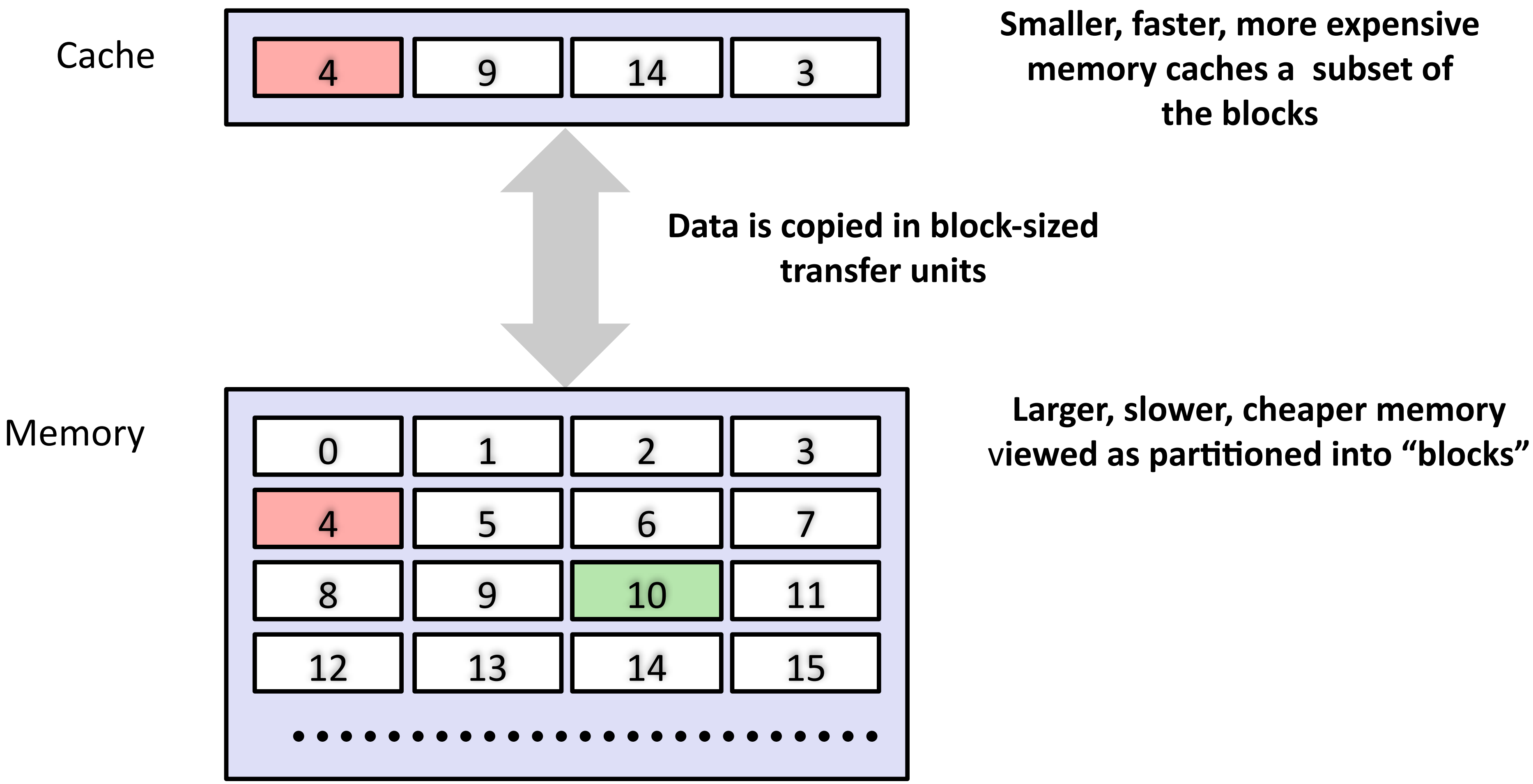


**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

# General Cache Mechanics

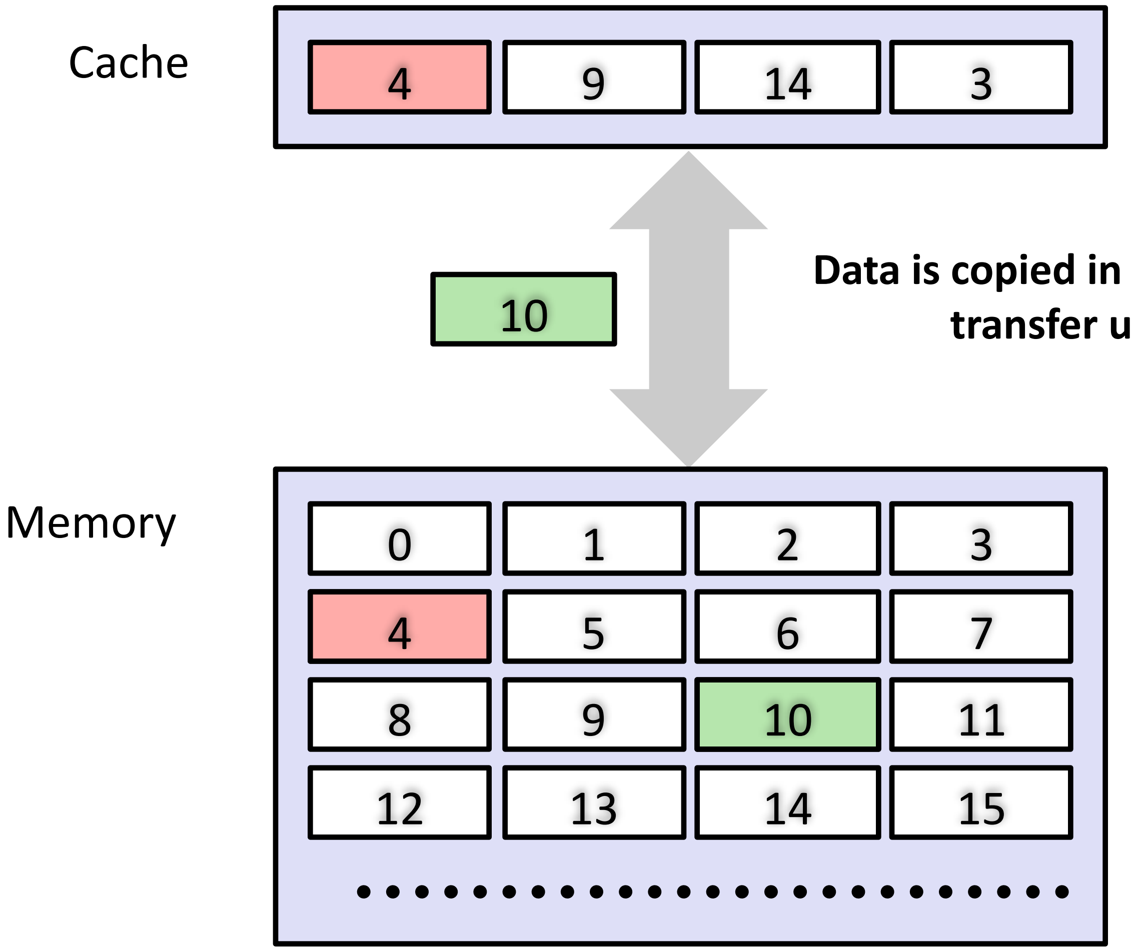


Smaller, faster, more expensive memory caches a  subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

# General Cache Mechanics

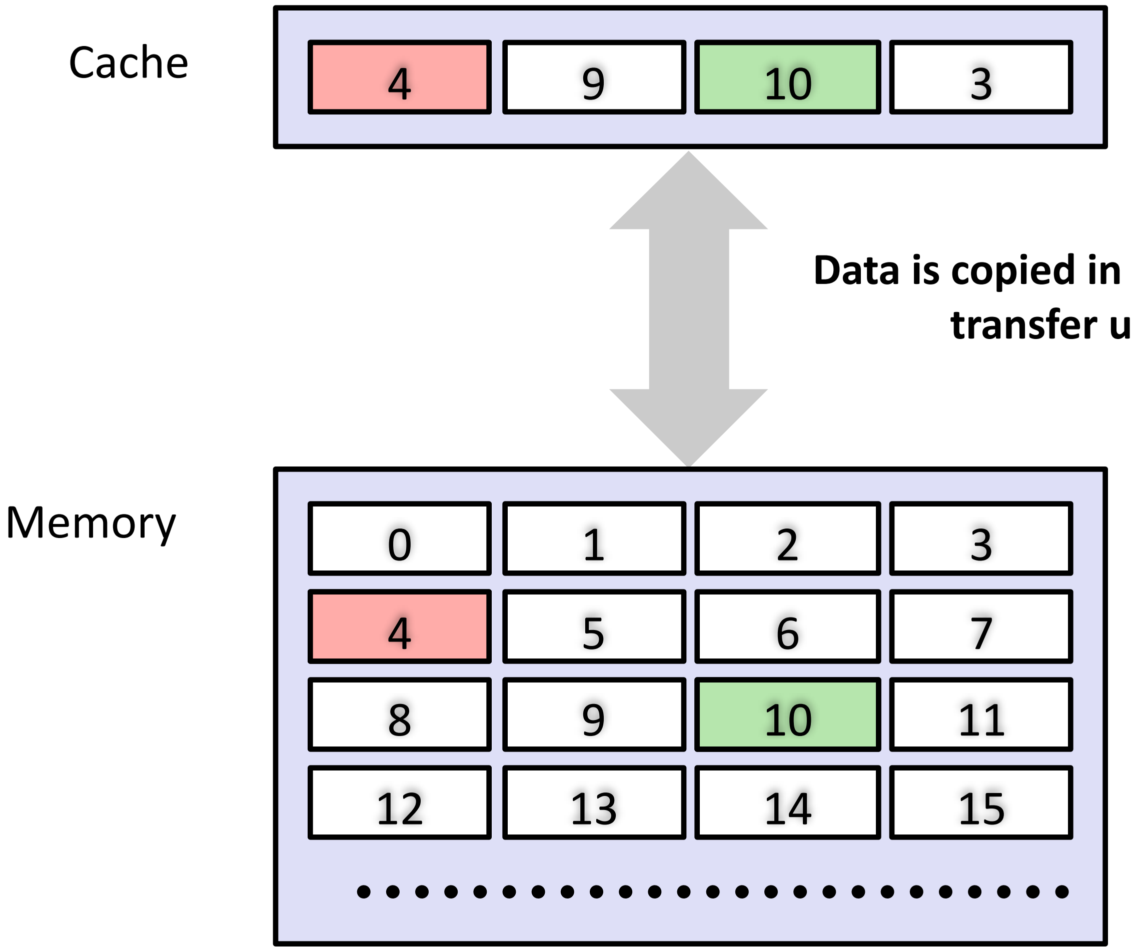


**Smaller, faster, more expensive memory caches a  subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into “blocks”**

# General Cache Mechanics

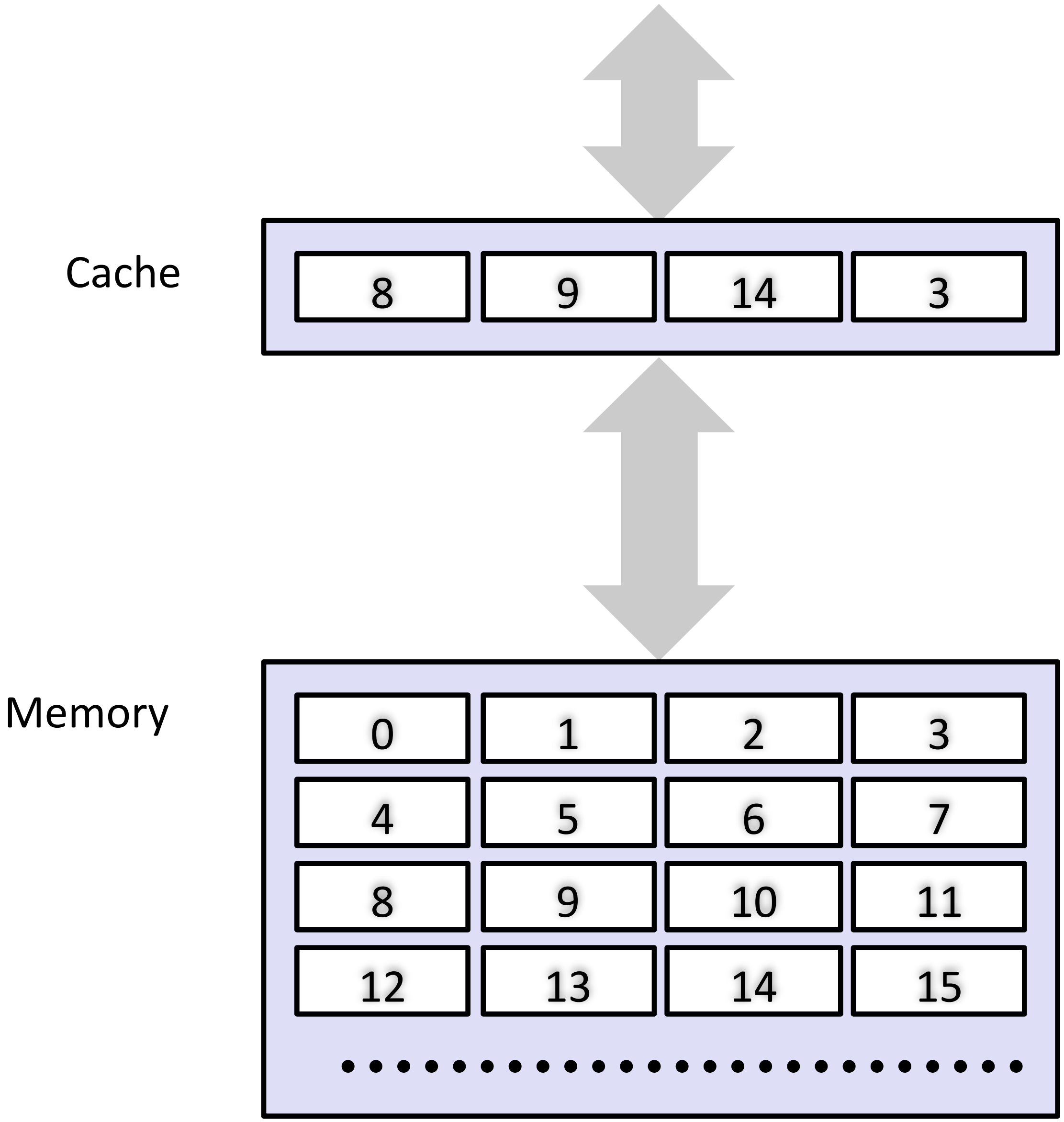


**Smaller, faster, more expensive memory caches a subset of the blocks**

**Data is copied in block-sized transfer units**

**Larger, slower, cheaper memory viewed as partitioned into "blocks"**

# General Cache Concepts: Hit



Cache

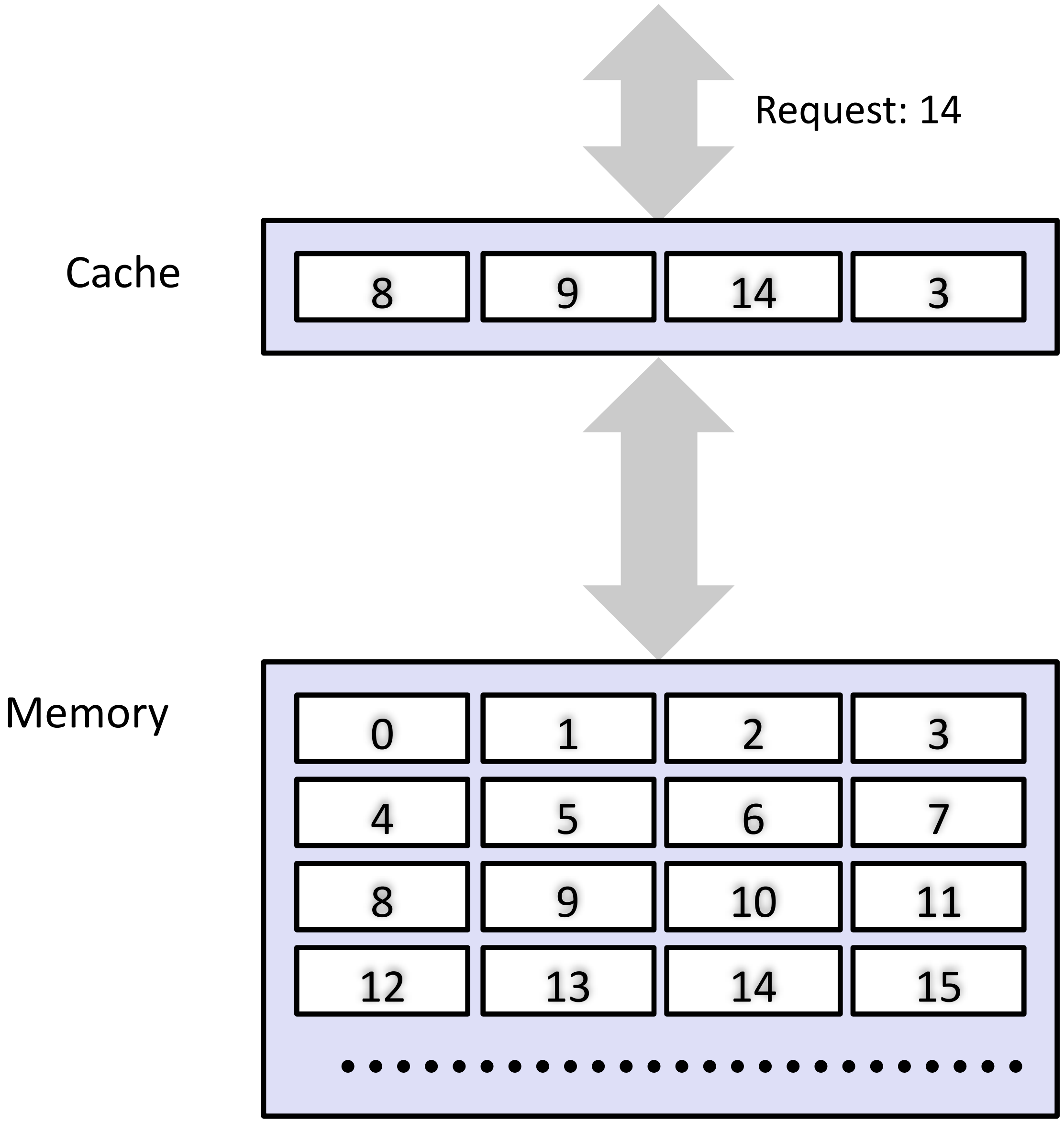
| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

Memory

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

• • • • • • • • • • • • • • • • • • • • • • • • •

# General Cache Concepts: Hit



Request: 14

Cache

| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

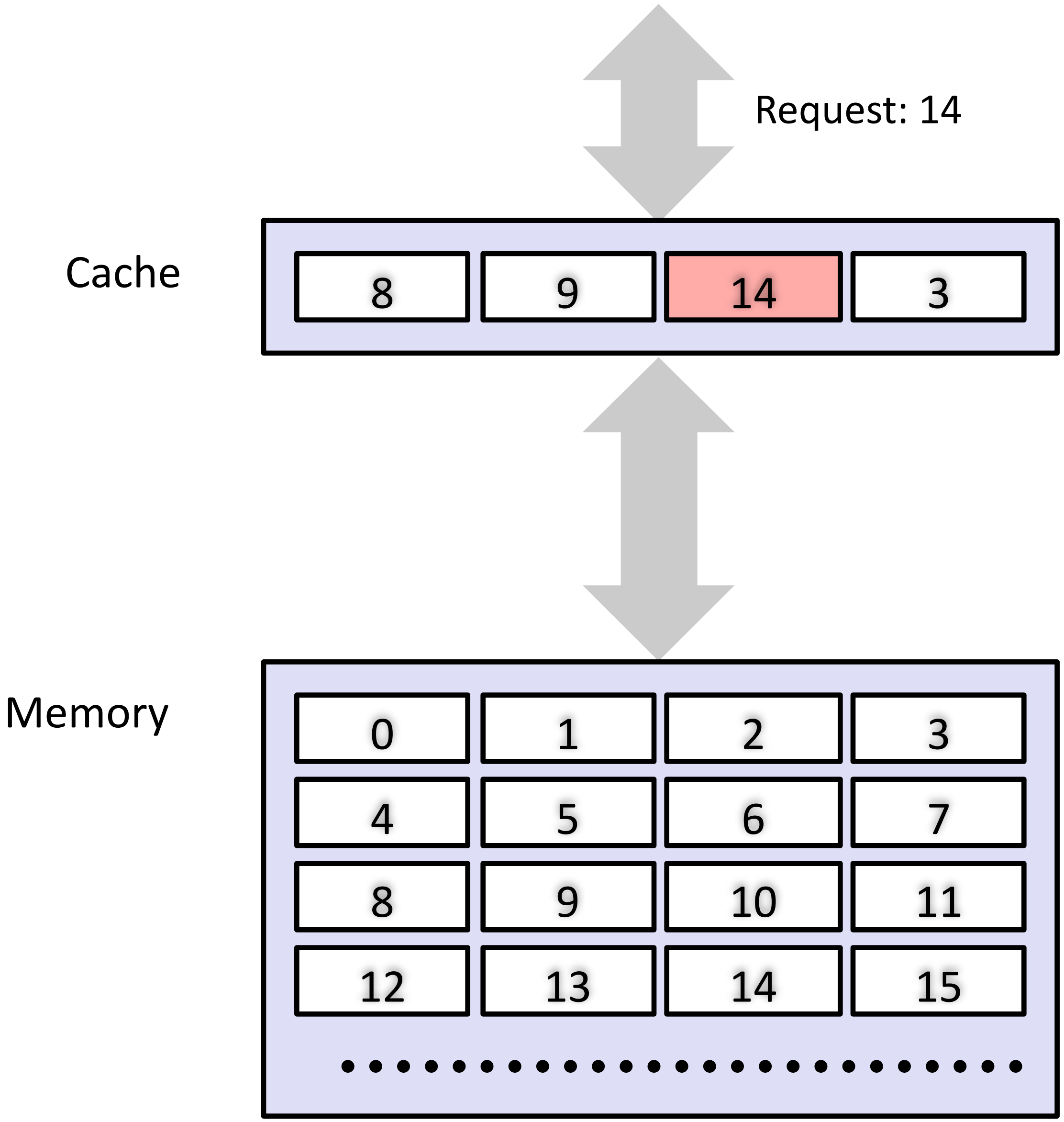
Memory

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

••••••••••••••••••••••••

***Data in block b is needed***

# General Cache Concepts: Hit

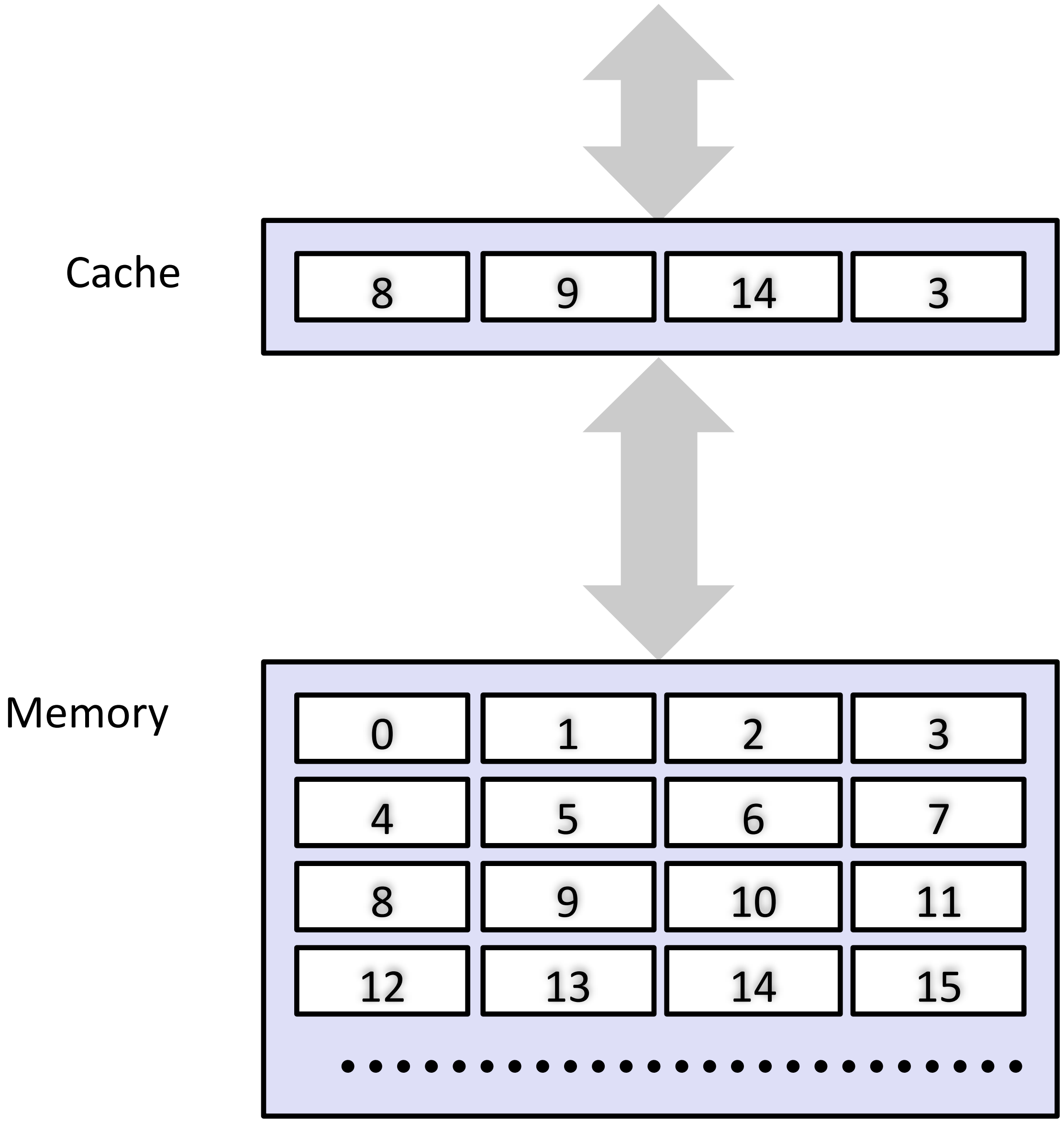


***Data in block b is needed***

***Block b is in cache:***

*Hit!*

# General Cache Concepts: Miss



Cache

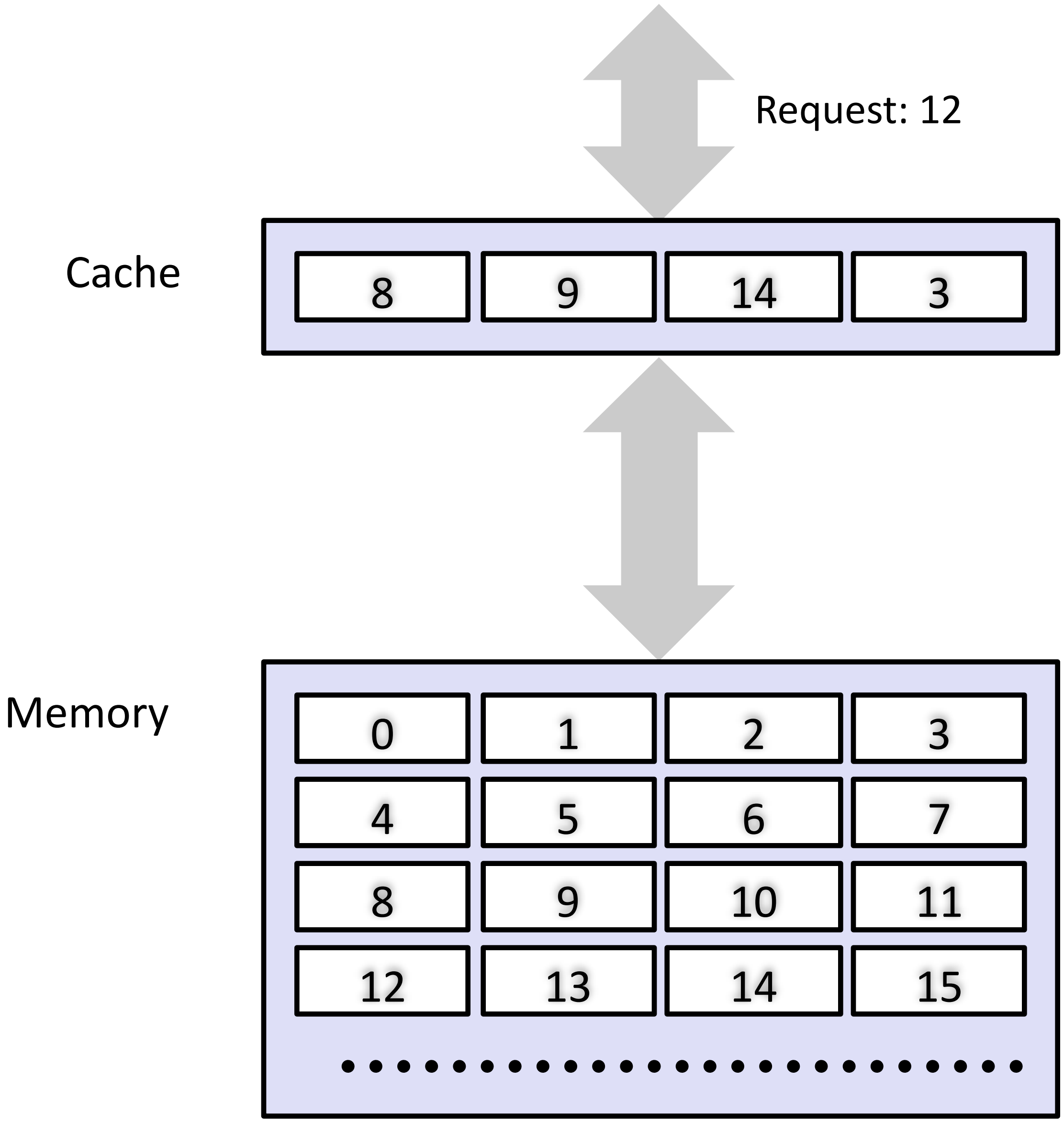
| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

Memory

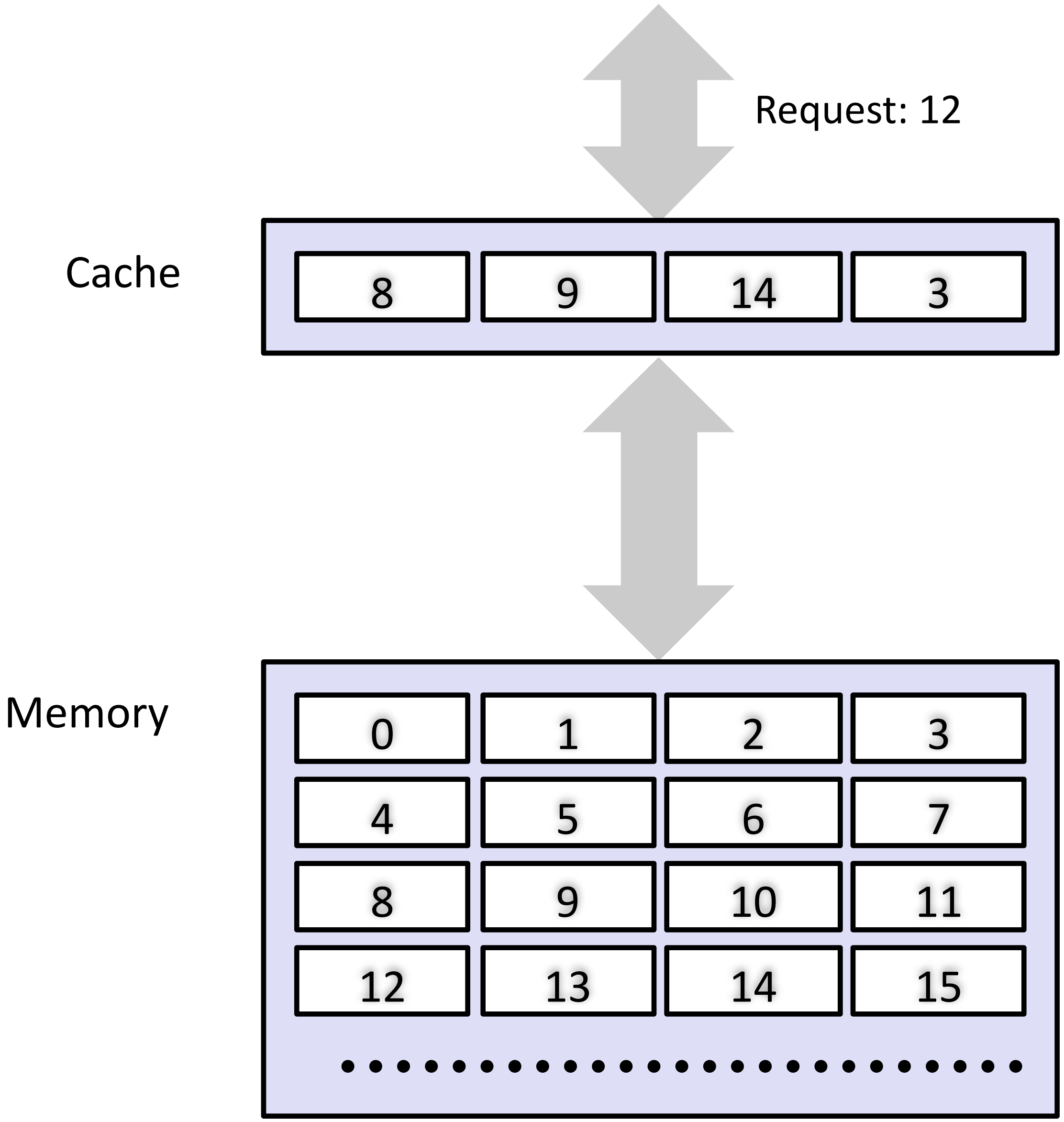
| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

# General Cache Concepts: Miss

***Data in block b is needed***



# General Cache Concepts: Miss



Request: 12

Cache

| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

Memory

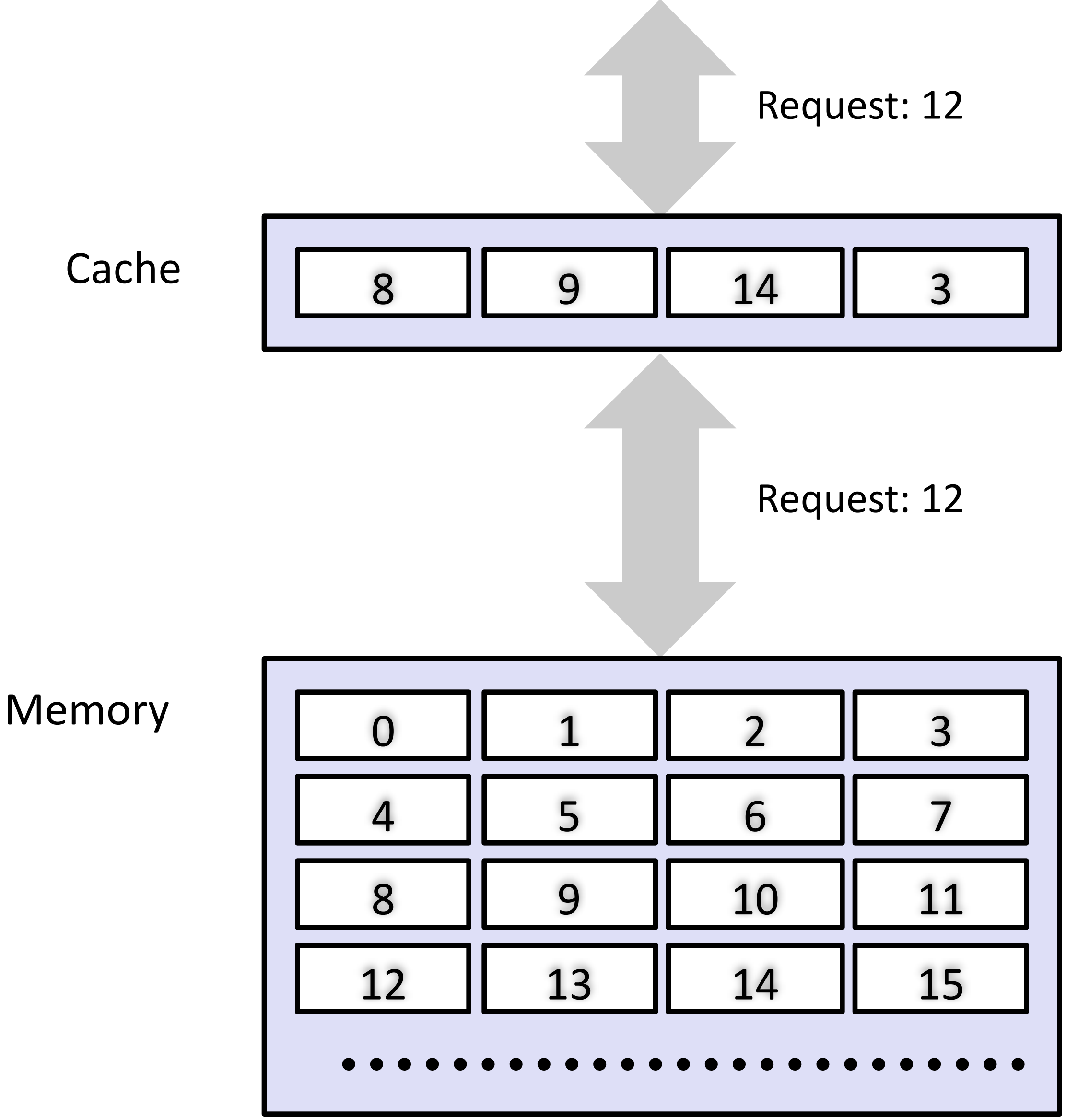
| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

• • • • • • • • • • • • • • • • • • • • • • • • •

***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

# General Cache Concepts: Miss



Request: 12

Cache

| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

Request: 12

Memory

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

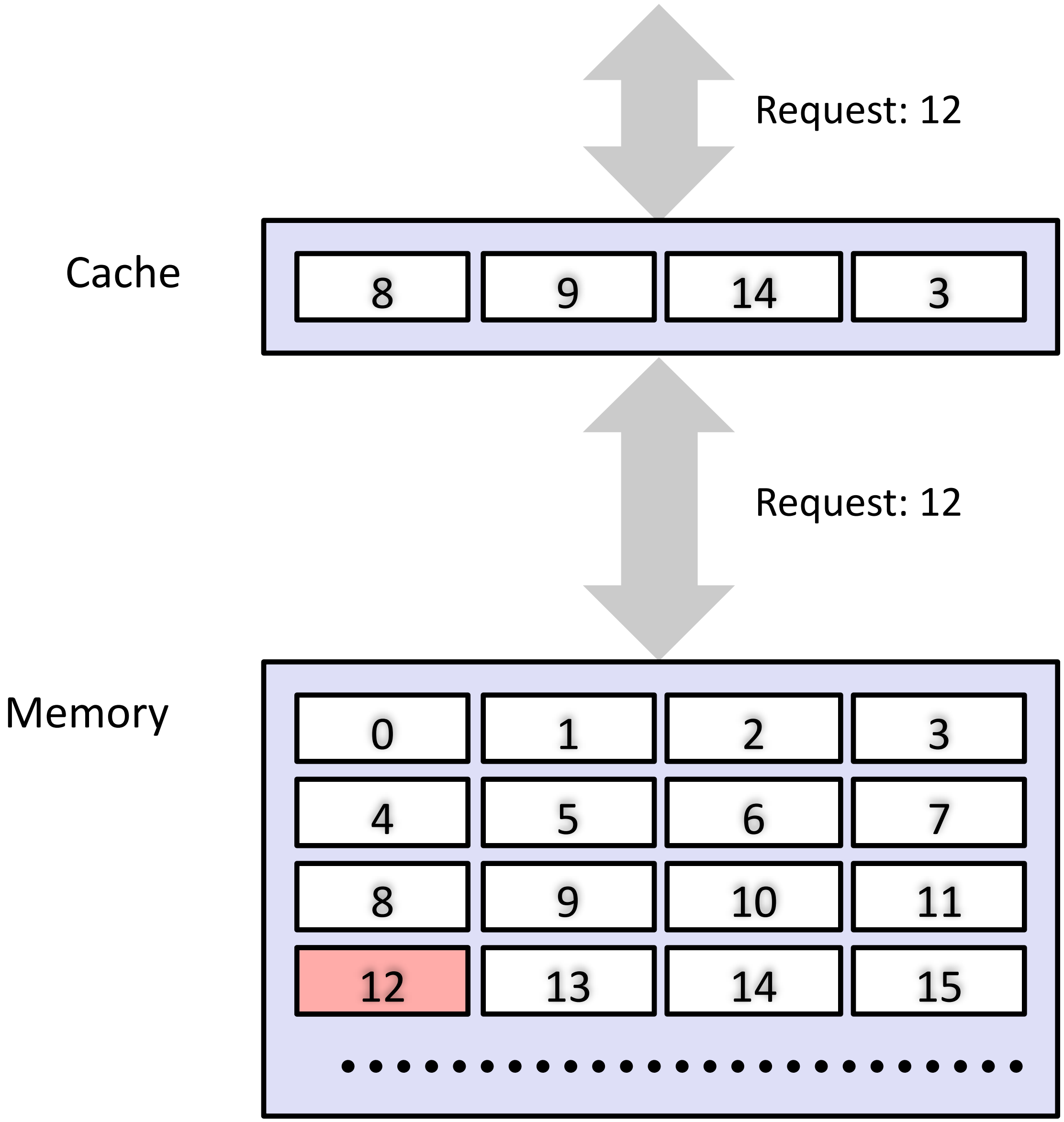
• • • • • • • • • • • • • • • • • • • • • • • • •

***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

***Block b is fetched from*** *memory*

# General Cache Concepts: Miss



| | | | |
|---|---|---|---|
| 8 | 9 | 14 | 3 |

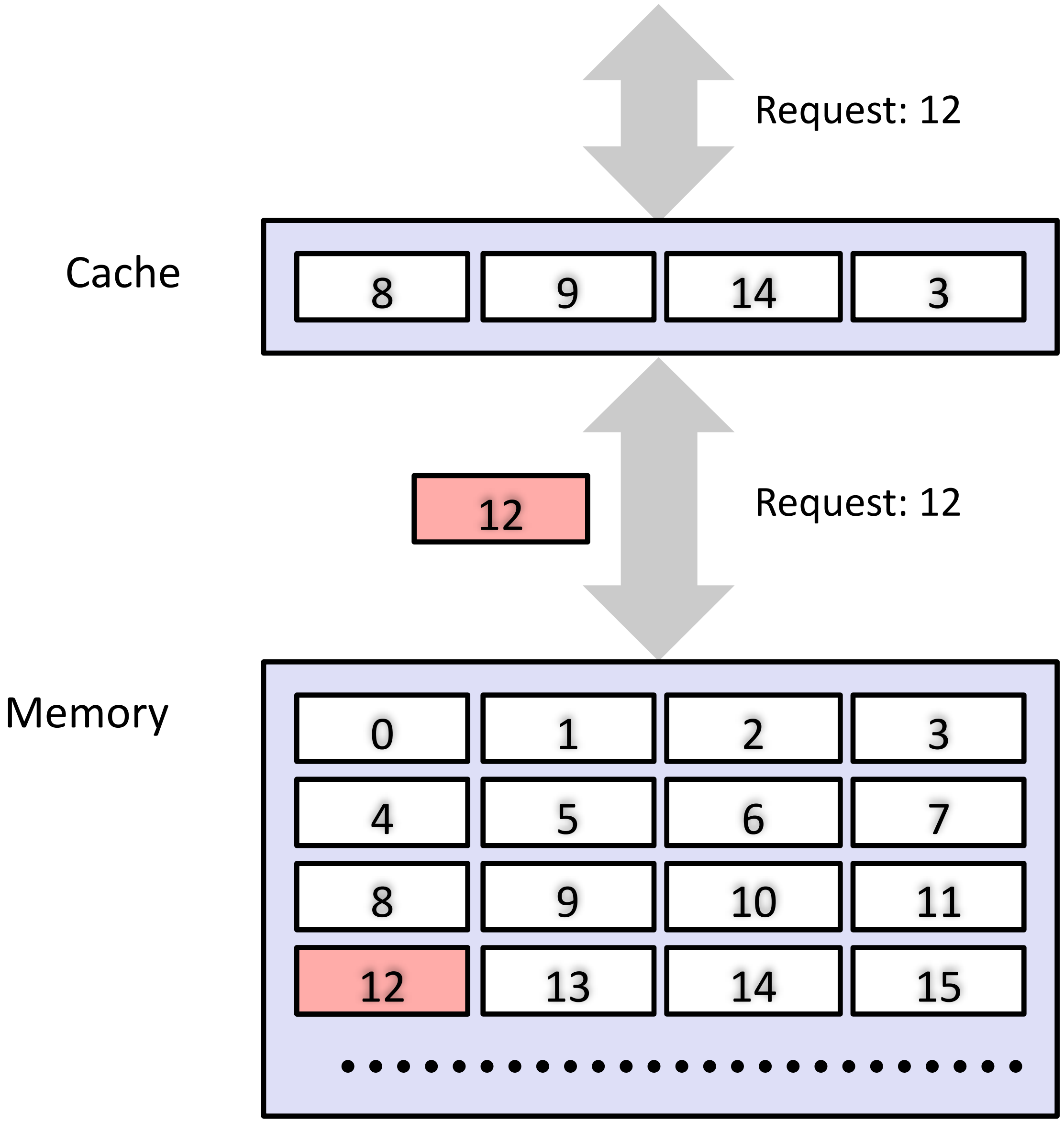
| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

***Block b is fetched from***
*memory*

# General Cache Concepts: Miss

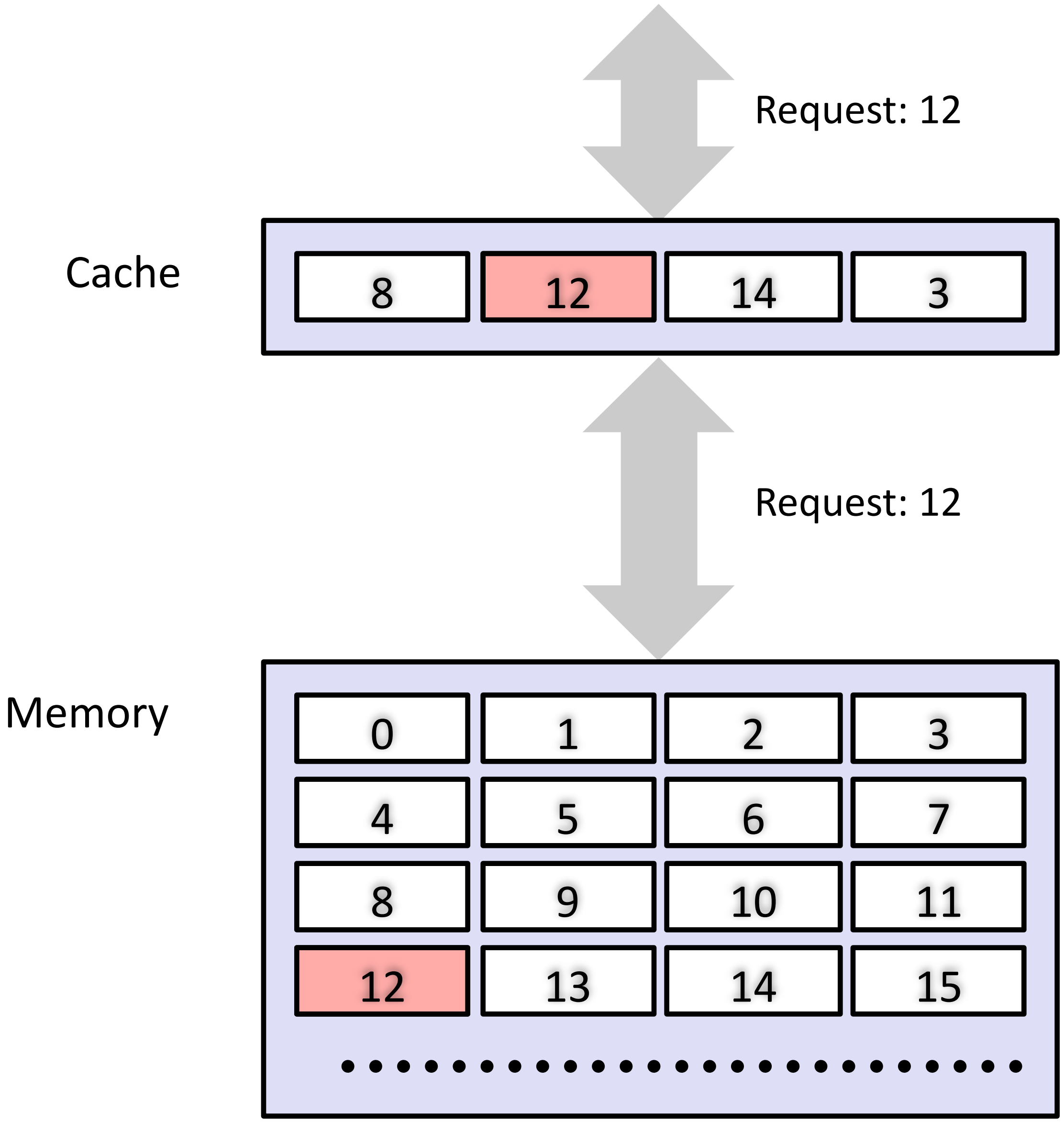


***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

***Block b is fetched from***
*memory*

# General Cache Concepts: Miss



***Data in block b is needed***

***Block b is not in cache:***
***Miss!***

***Block b is fetched from*** *memory*

***Block b is stored in cache***

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

# Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

# Lets think about those numbers

# Lets think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

# Lets think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

# Lets think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:

    97% hits:  1 cycle + 0.03 * 100 cycles = **4 cycles**

    99% hits:  1 cycle + 0.01 * 100 cycles = **2 cycles**

# Lets think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**

    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- **This is why "miss rate" is used instead of "hit rate"**

# Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
    - e.g., block i must be placed in slot (i mod 4)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

# Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
    - e.g., block i must be placed in slot (i mod 4)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache

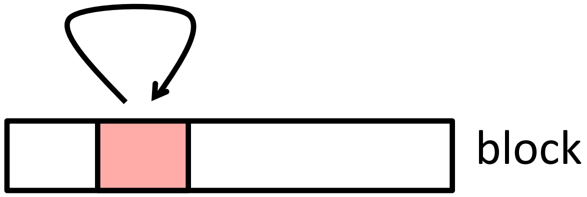
# Why Caches Work

# Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

# Why Caches Work

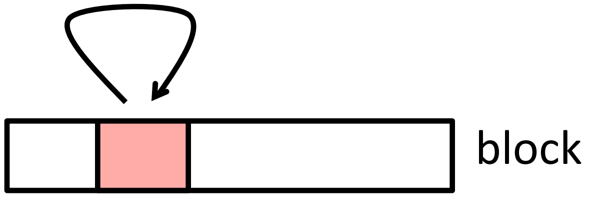
- **Locality:** **Programs tend to use data and instructions with addresses near or equal to those they have used recently**



# Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

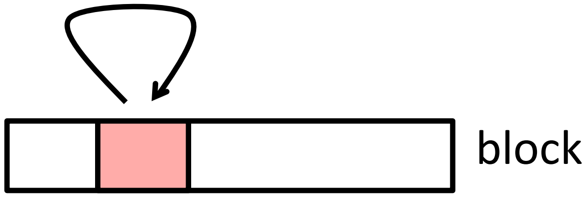
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



# Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

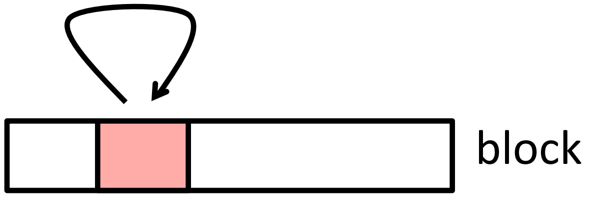
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



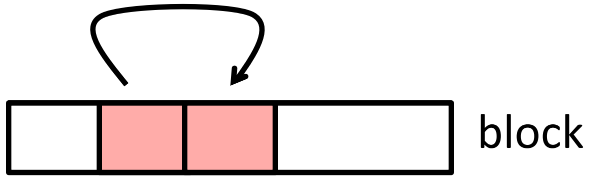
# Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



# Example: Locality?

```
       sum = 0;
for (i = 0; i < n; i++)
          sum += a[i];
      return sum;
```

# Example: Locality?

```
       sum = 0;
for (i = 0; i < n; i++)
          sum += a[i];
      return sum;
```

- **Data:**
  - Temporal: **`sum`** referenced in each iteration
  - Spatial: array **`a[]`** accessed in stride-1 pattern

# Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data:**
  - Temporal: **`sum`** referenced in each iteration
  - Spatial: array **`a[]`** accessed in stride-1 pattern
- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

# Example: Locality?

```
          sum = 0;
   for (i = 0; i < n; i++)
             sum += a[i];
        return sum;
```

- **Data:**
  - Temporal: **`sum`** referenced in each iteration
  - Spatial: array **`a[]`** accessed in stride-1 pattern
- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

# Example: Locality?

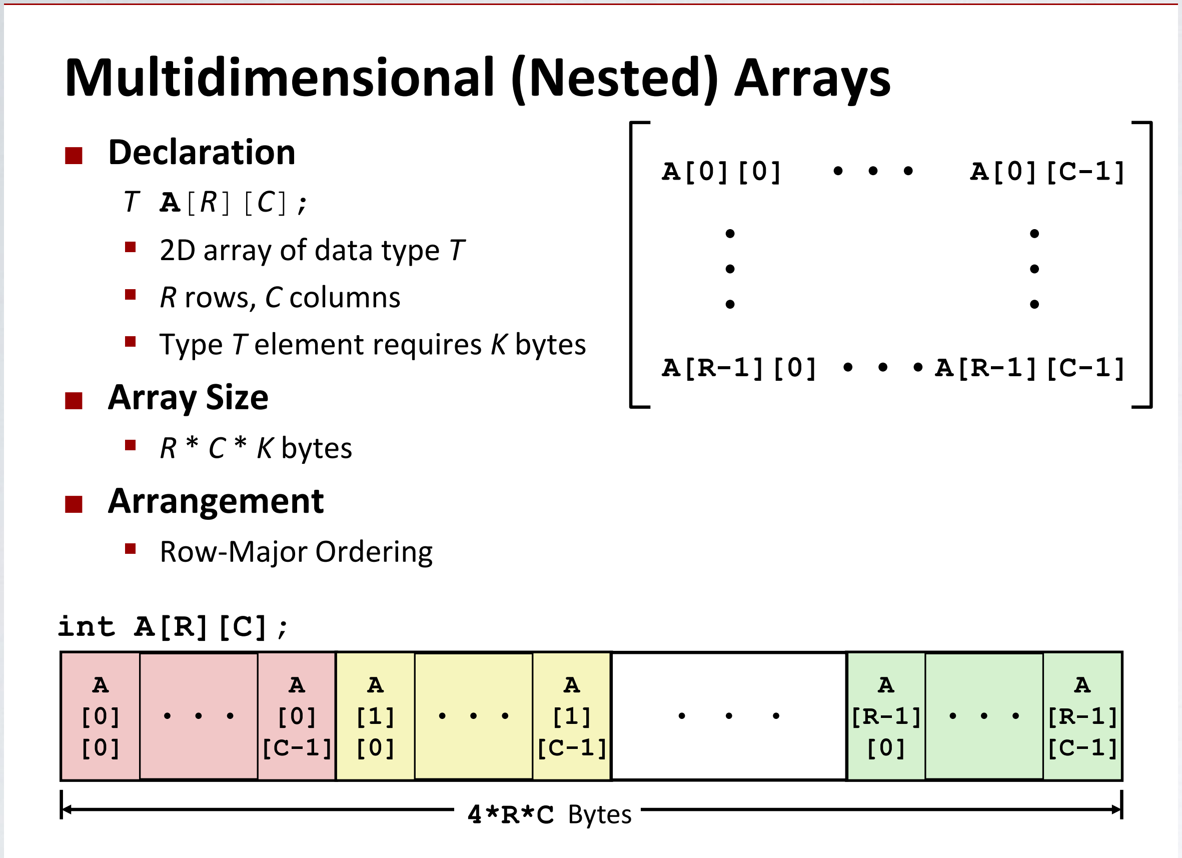
```
         sum = 0;
for (i = 0; i < n; i++)
           sum += a[i];
       return sum;
```

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern
- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**

# Locality Example #1

```
int sum_array_rows(int a[M][N])
               {
         int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```



## Multidimensional (Nested) Arrays

- **Declaration**

  *T* **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

```
int A[R][C];
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A<br>[0]<br>[0] | • • • | A<br>[0]<br>[C-1] | A<br>[1]<br>[0] | • • • | A<br>[1]<br>[C-1] | • • • | A<br>[R-1]<br>[0] | • • • | A<br>[R-1]<br>[C-1] |

**4*R*C** Bytes

*A*[*i*][*j*] is element of type *T*, which requires *K* bytes

Address:

$A + i*(C*K) + j* K$

$= A + (i*C+j) * K$

# Locality Example #1

```
int sum_array_rows(int a[M][N])
             {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
             }
```

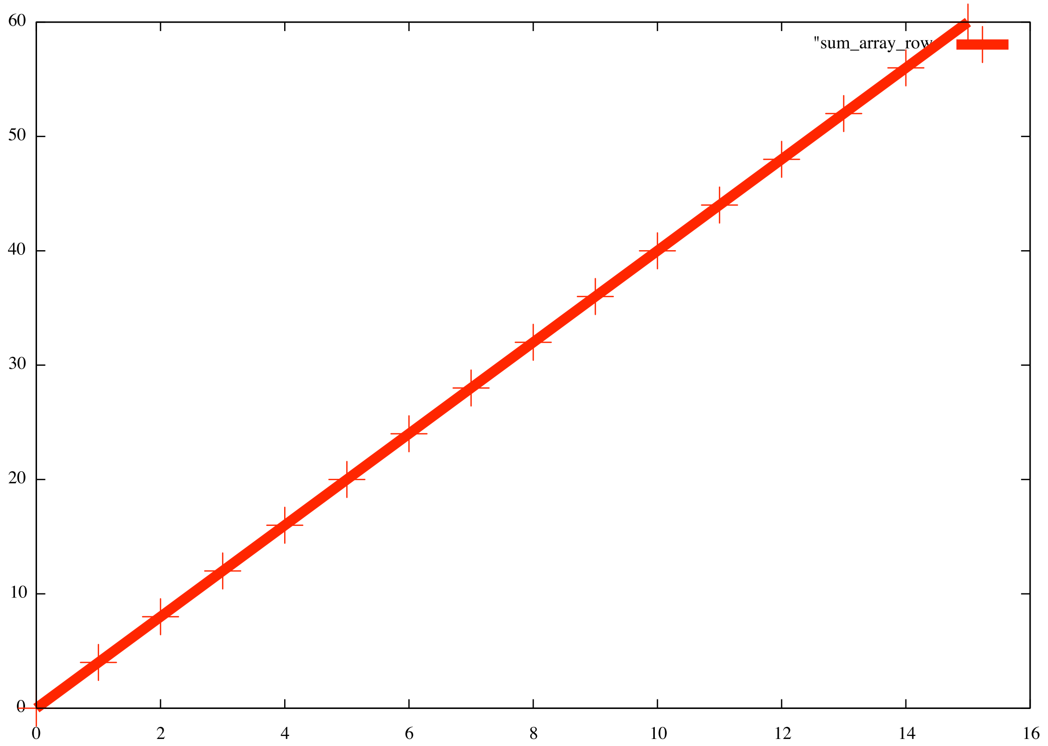
***A[i][j]*** **is element of type** ***T*****,**
**which requires** ***K*** **bytes**

**Address:**

$$A + i^*(C^*K) + j^* K$$
$$= A + (i^*C+j) * K$$

a + (i*N+j) * sizeof(int) = a + 4(i*N+j) : N=4,M=4

| x | a | x | a |
|---|---|---|---|
| 0 | 0 | 8 | 32 |
| 1 | 4 | 9 | 36 |
| 2 | 8 | 10 | 40 |
| 3 | 12 | 11 | 44 |
| 4 | 16 | 12 | 48 |
| 5 | 20 | 13 | 52 |
| 6 | 24 | 14 | 56 |
| 7 | 28 | 15 | 60 |



# Locality Example #2

```
int sum_array_cols(int a[M][N])
               {
         int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```

# Locality Example #2

```
int sum_array_cols(int a[M][N])
                {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
                }
```

*A[i][j]* is element of type *T*, which requires *K* bytes

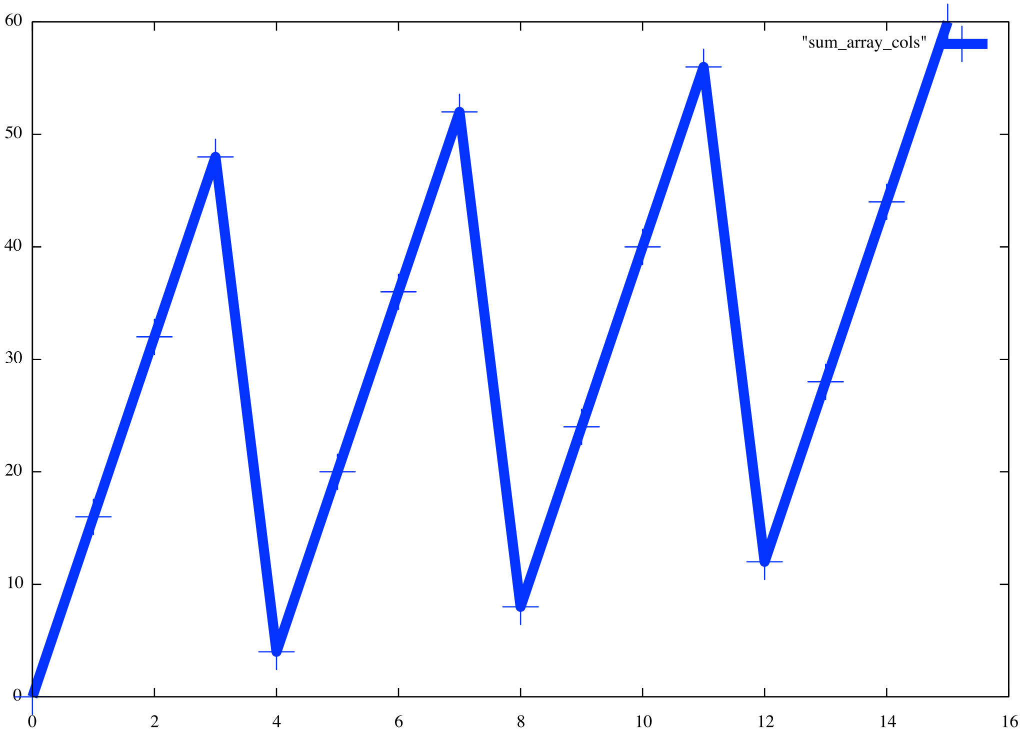
Address:

$A + i*(C*K) + j*K$

$= A + (i*C+j) * K$

a + (i*N+j) * sizeof(int) = a + 4(i*N+j) : N=4,M=4

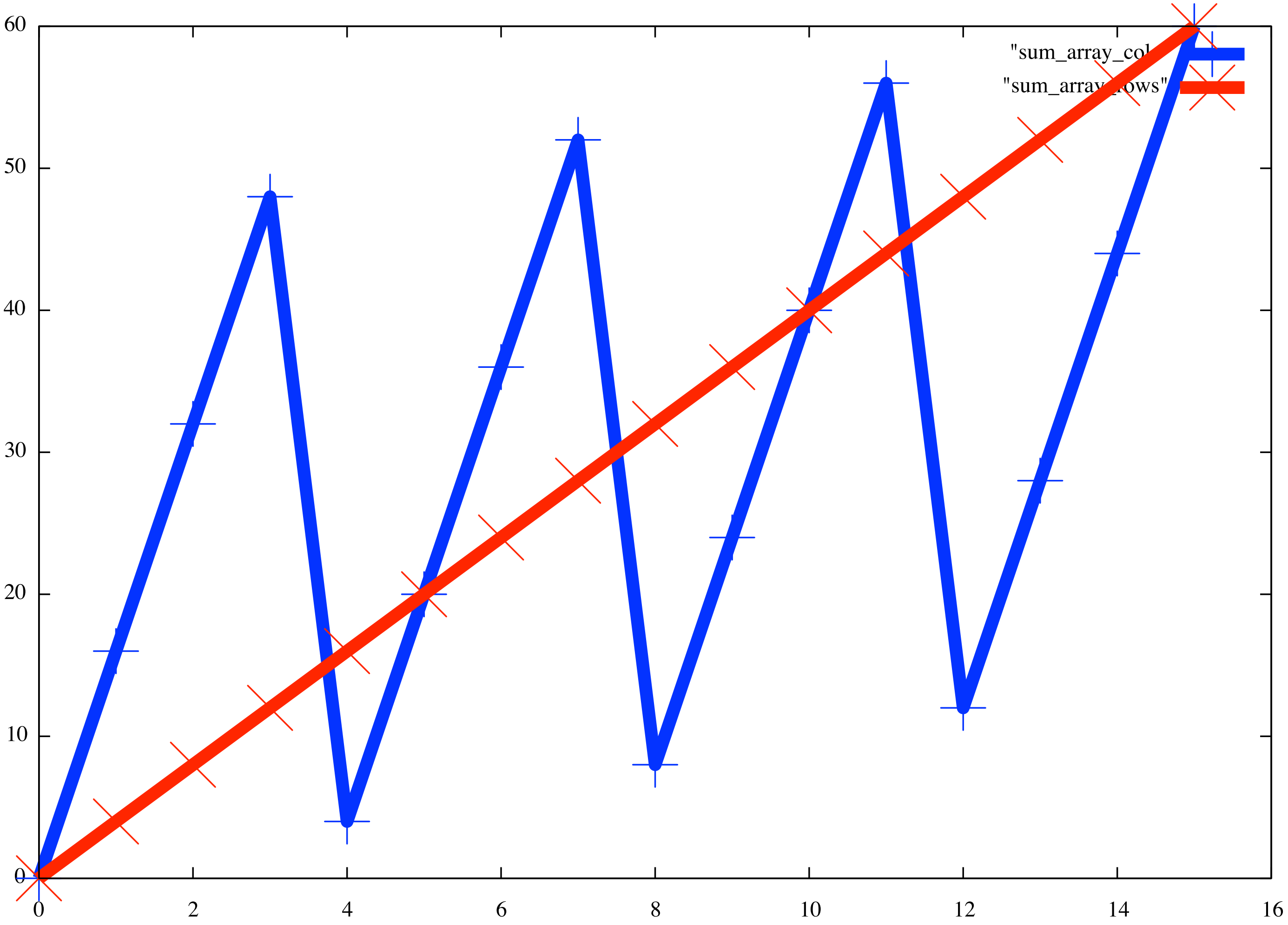
| x | a | x | a |
|---|---|---|---|
| 0 | 0 | 8 | 8 |
| 1 | 16 | 9 | 24 |
| 2 | 32 | 10 | 40 |
| 3 | 48 | 11 | 56 |
| 4 | 4 | 12 | 12 |
| 5 | 20 | 13 | 28 |
| 6 | 36 | 14 | 44 |
| 7 | 52 | 15 | 60 |



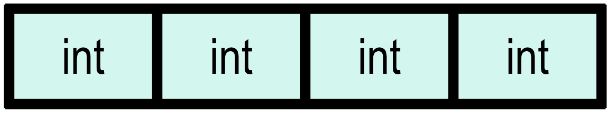
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

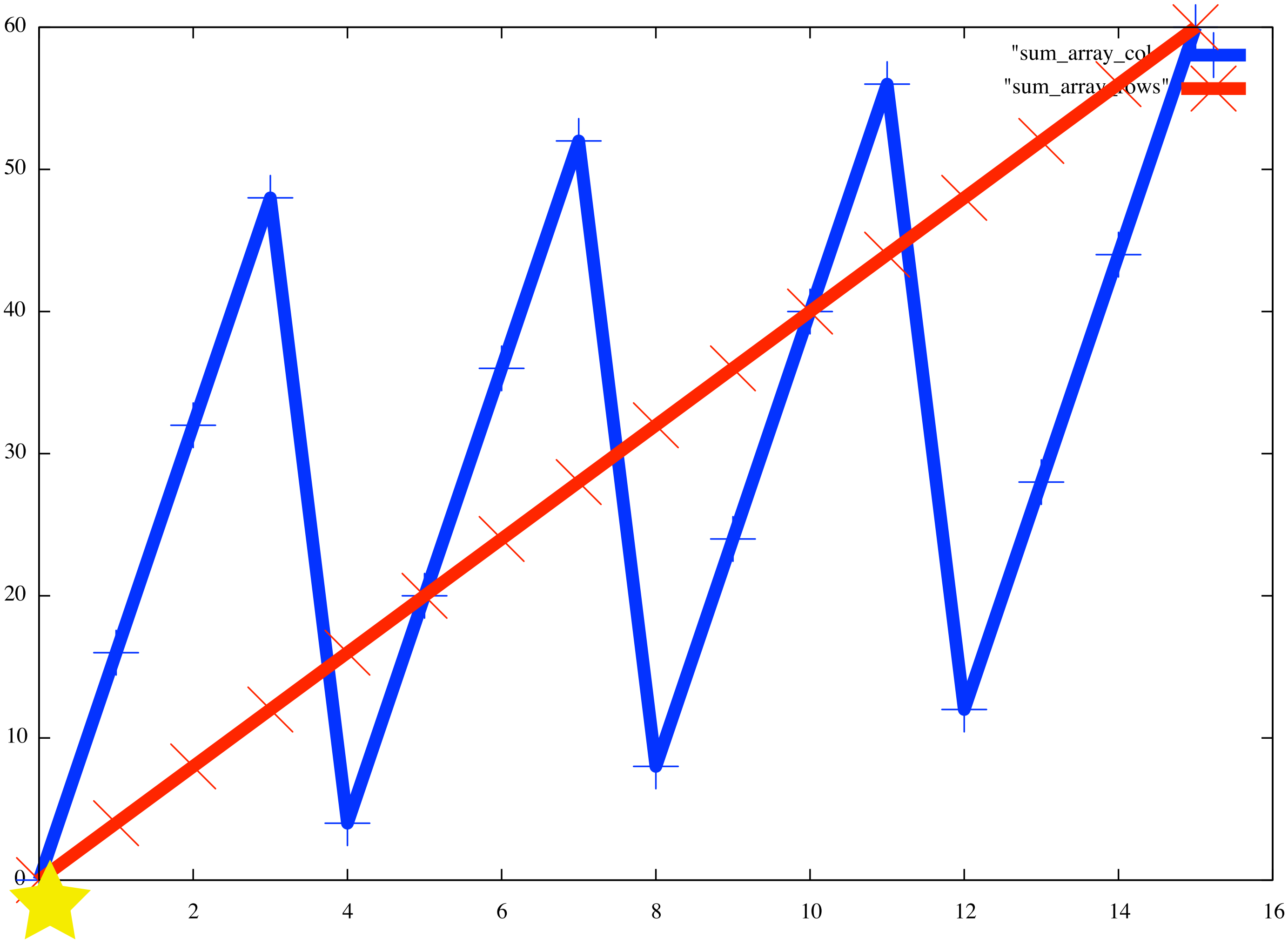


int int int int

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



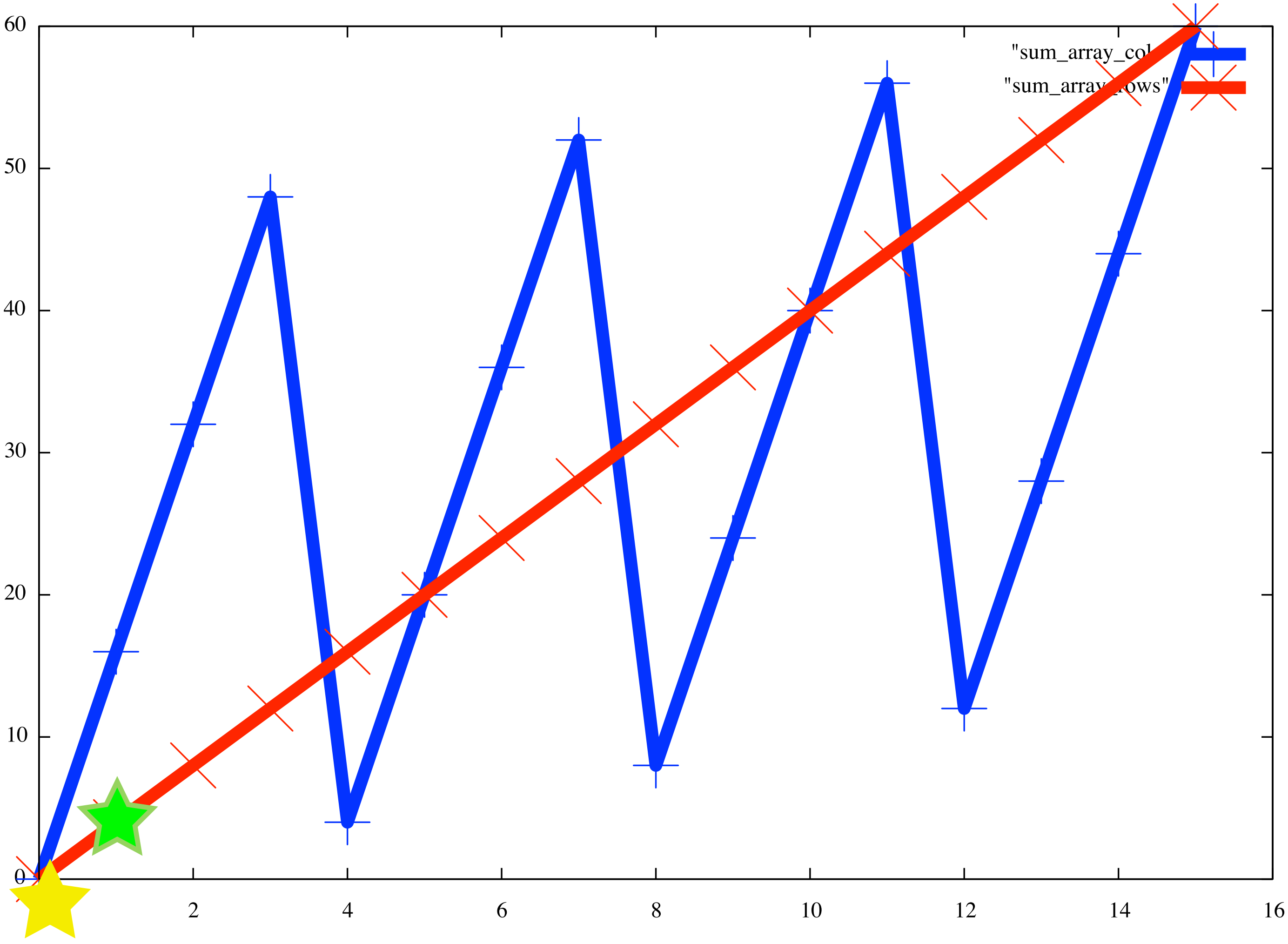
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



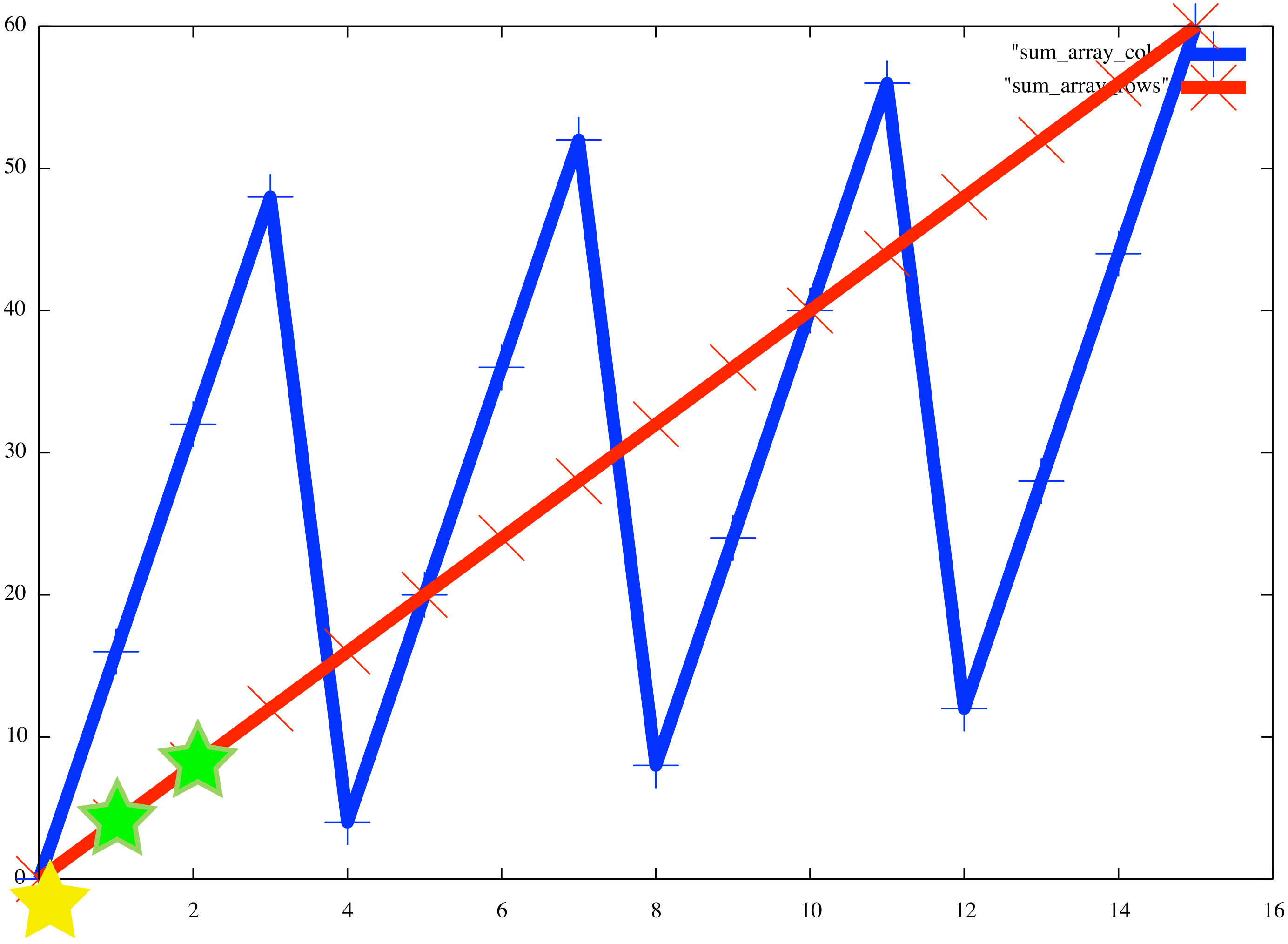
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

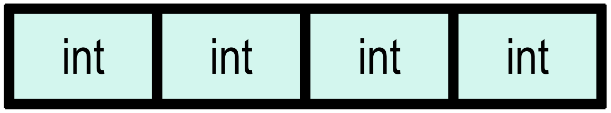
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



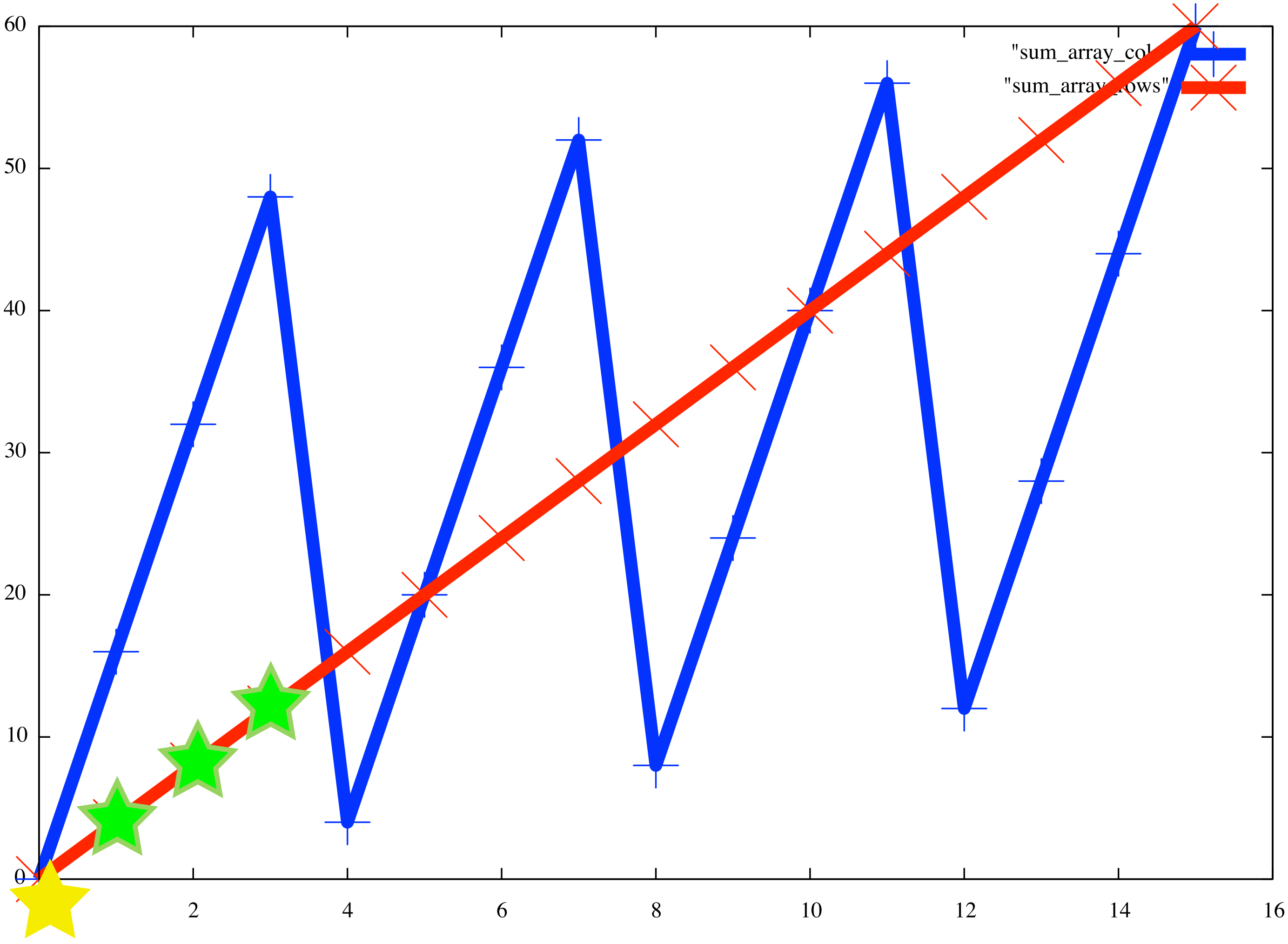
single line 16 byte cache


int int int int

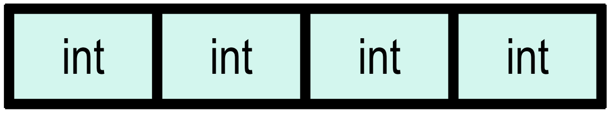
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



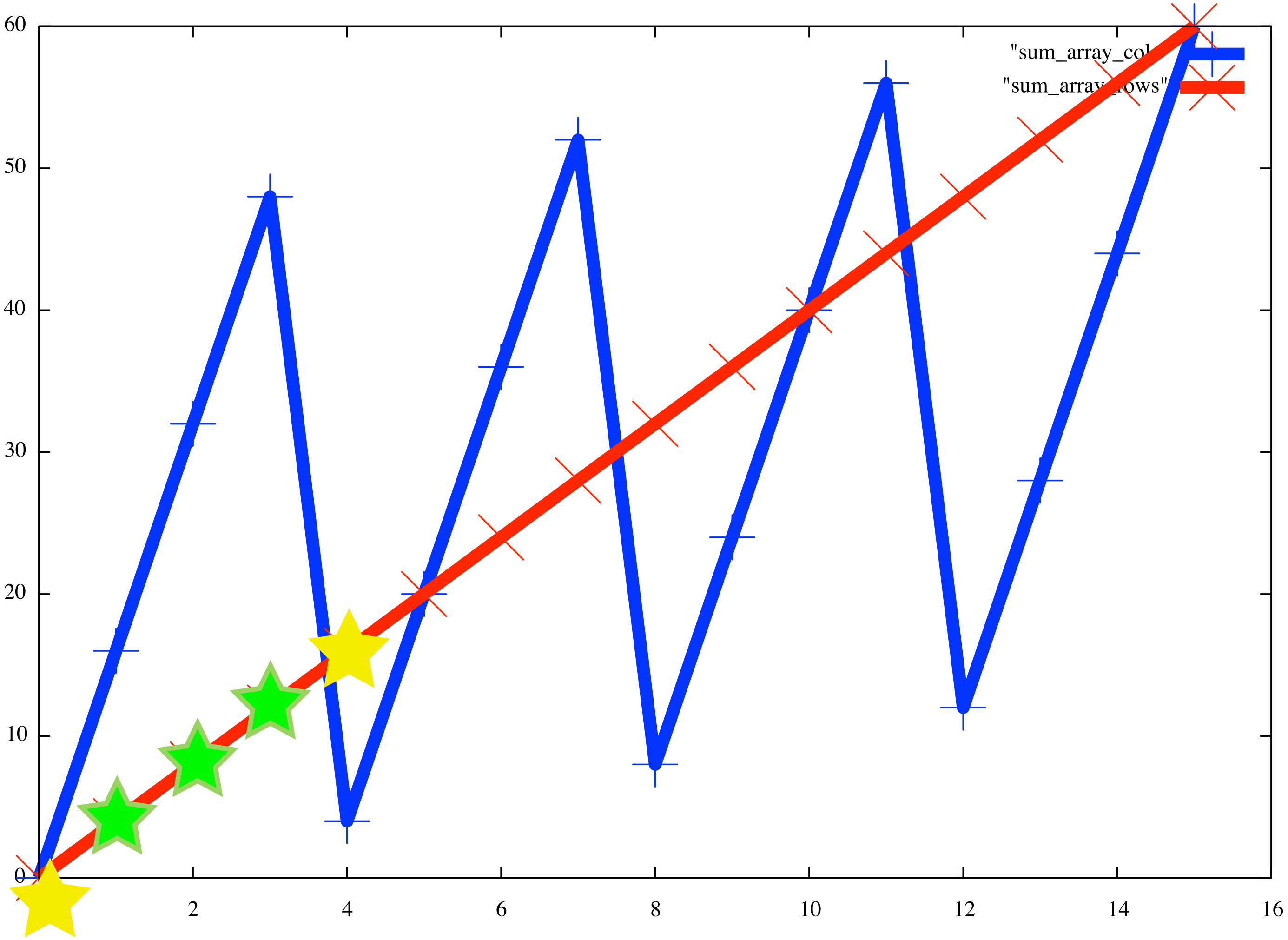
single line 16 byte cache



# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



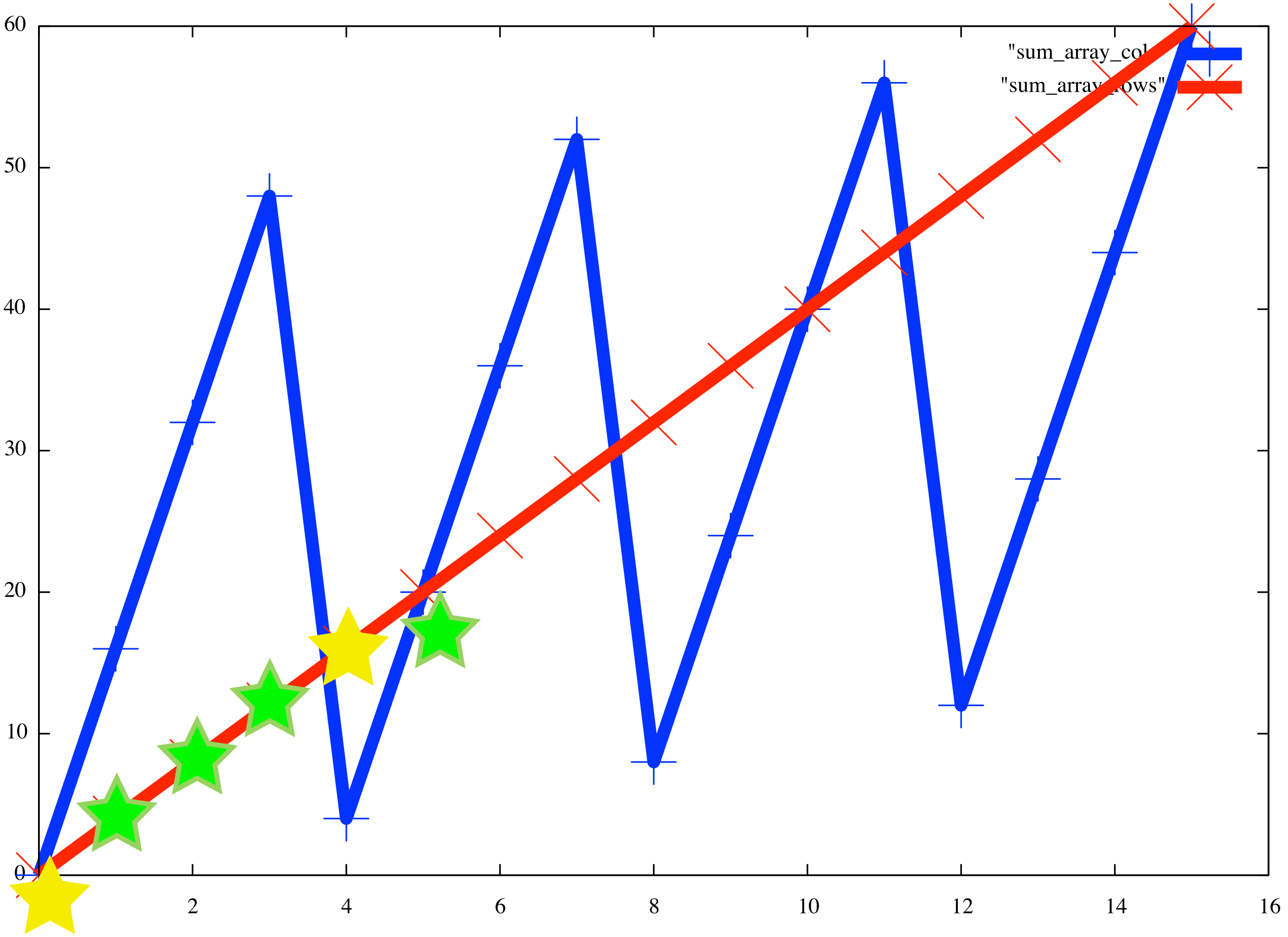
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

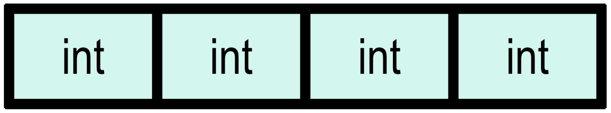
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



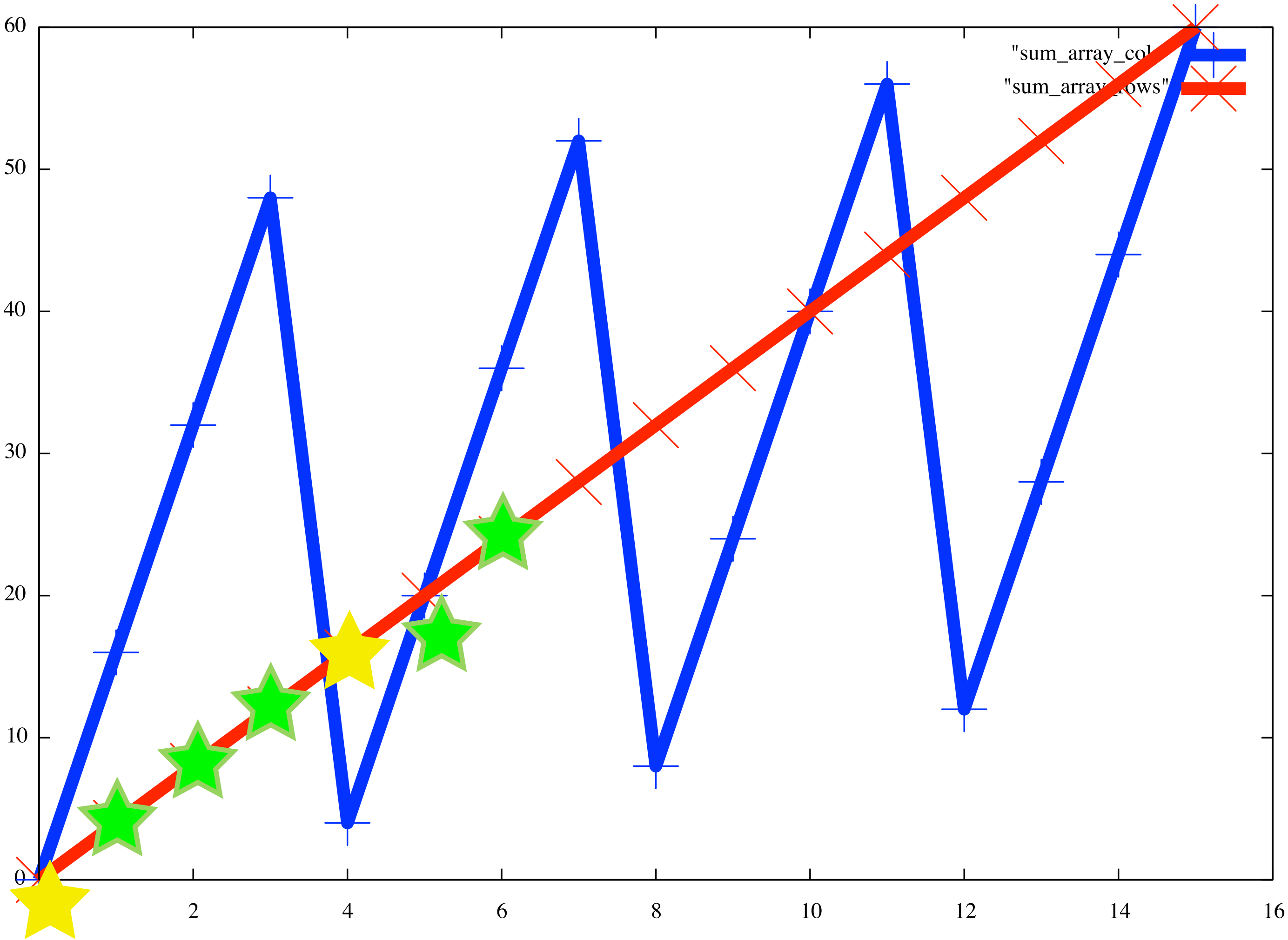
single line 16 byte cache


int int int int

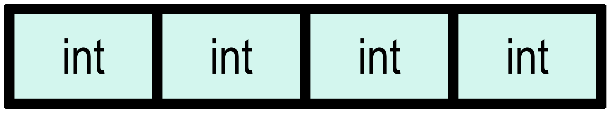
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

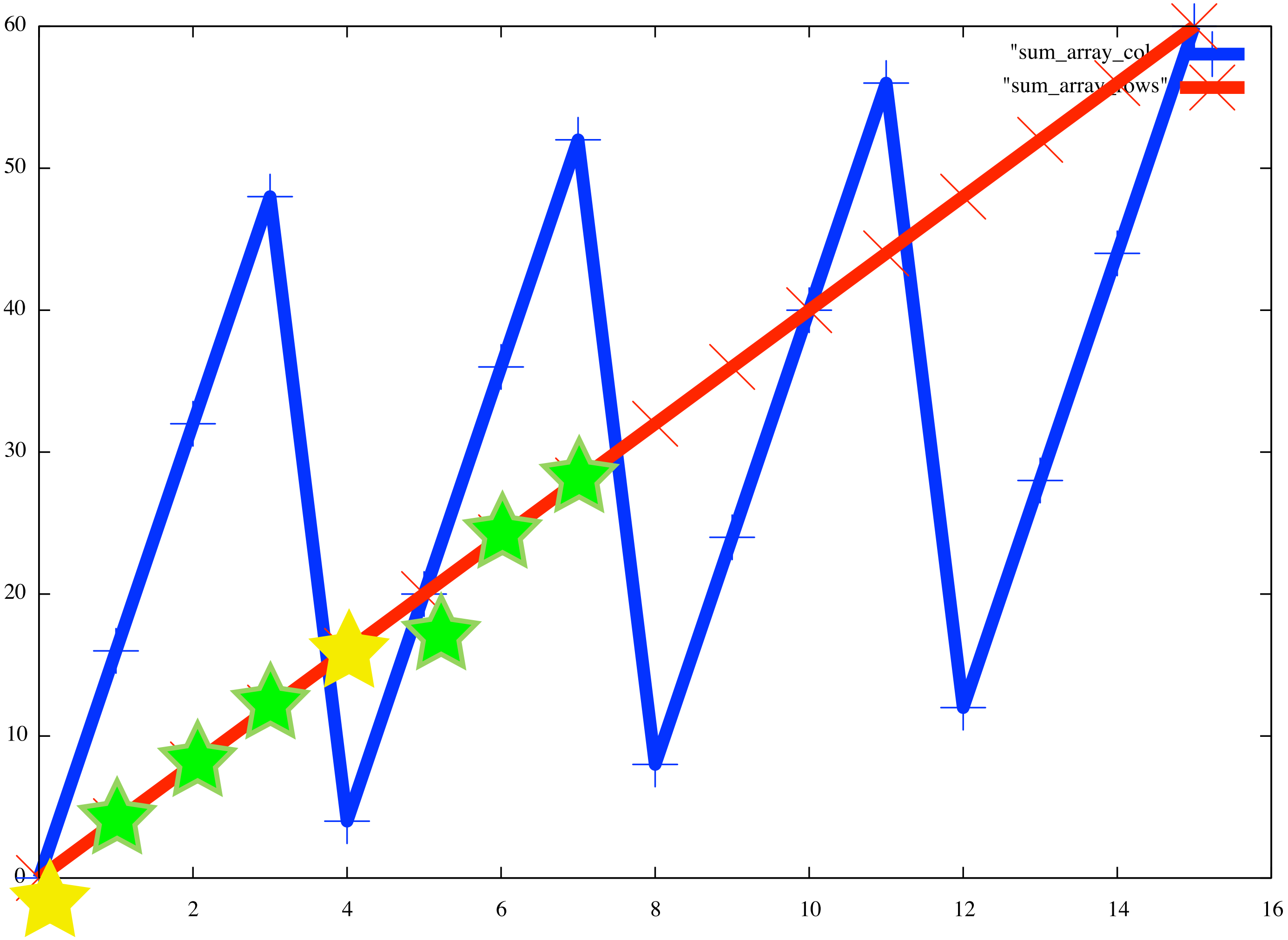


| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



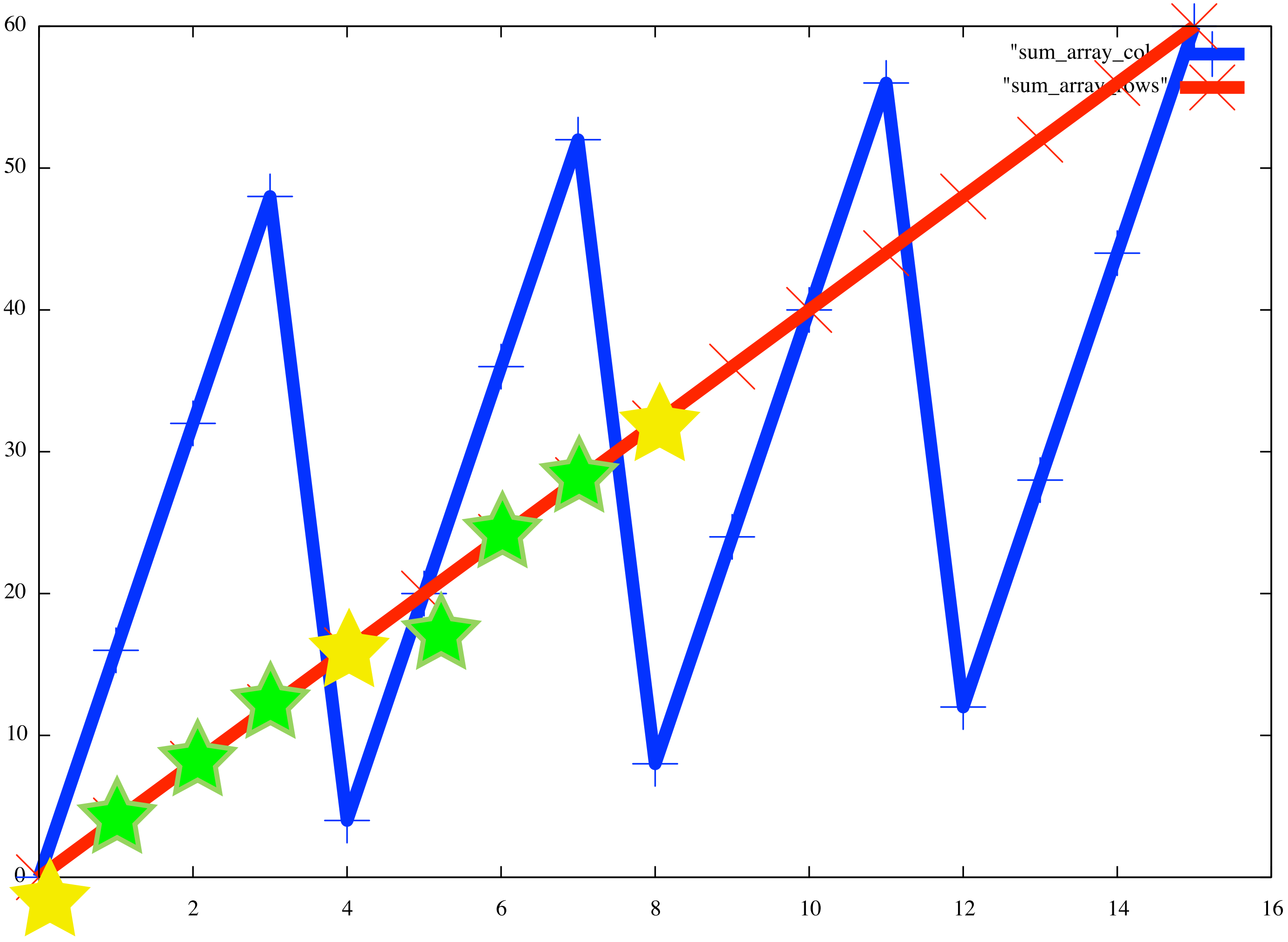
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



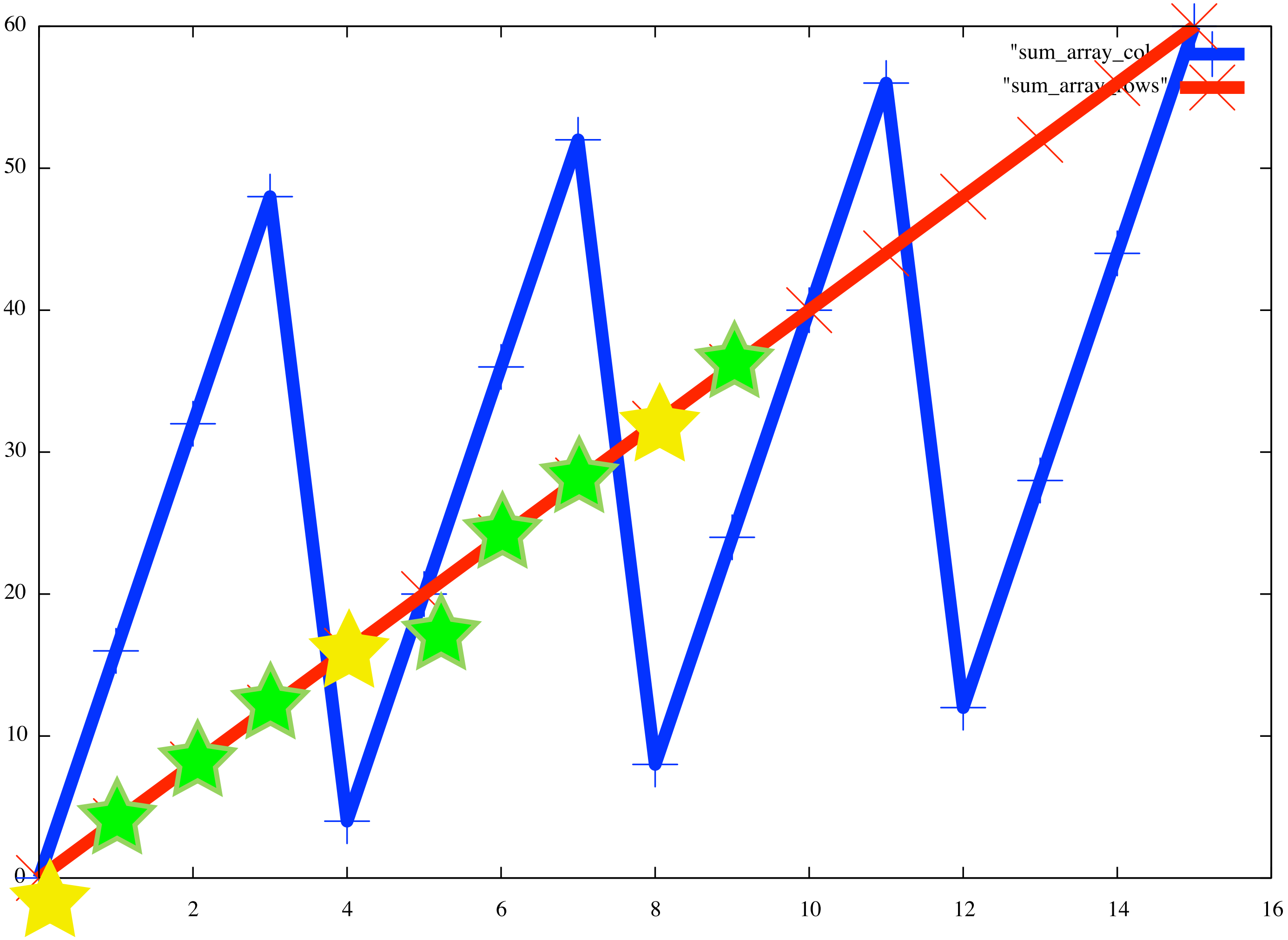
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

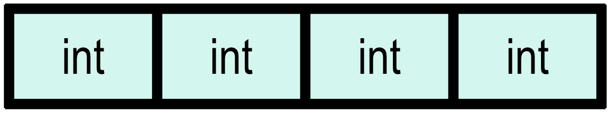
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



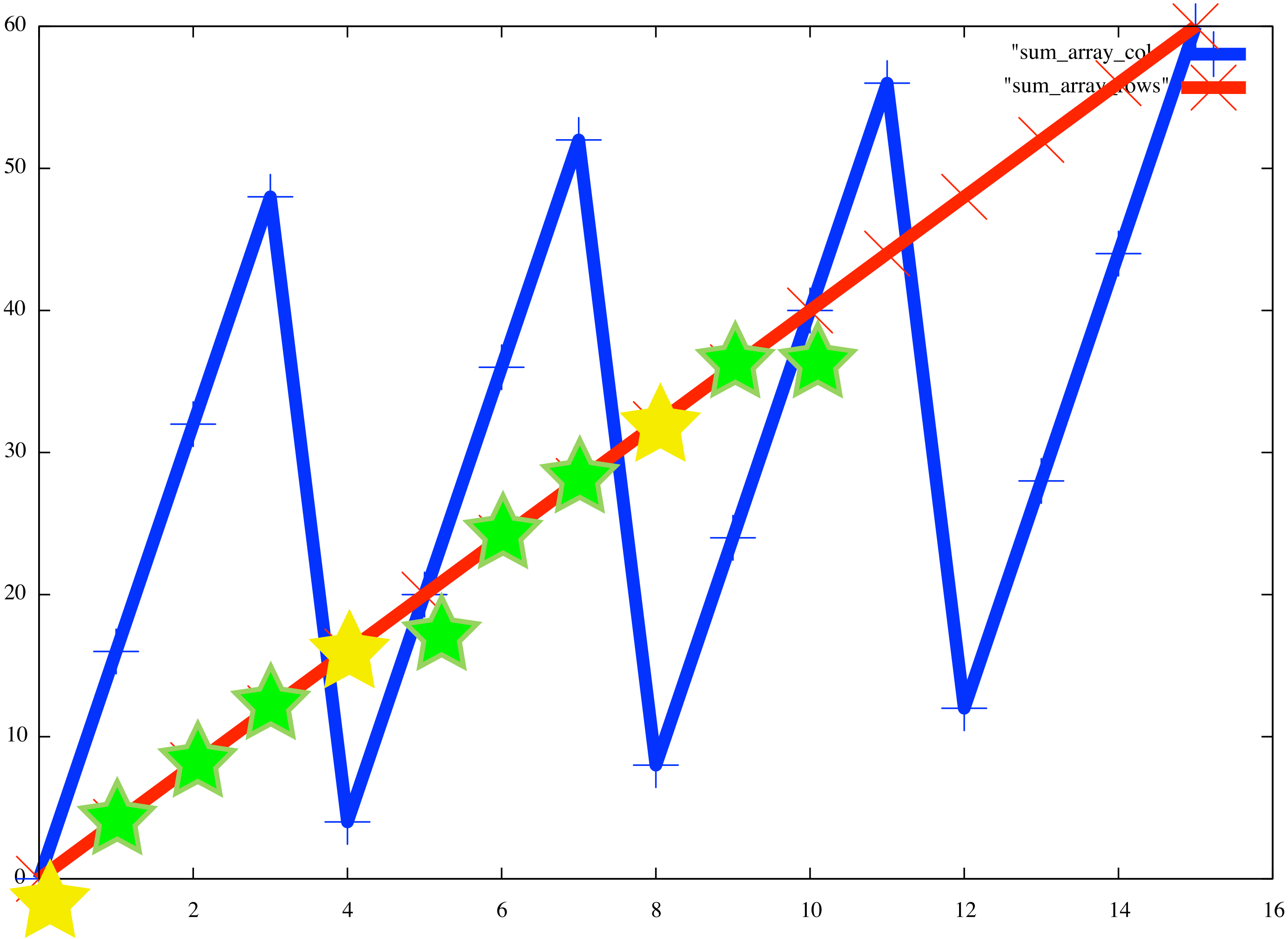
single line 16 byte cache



# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



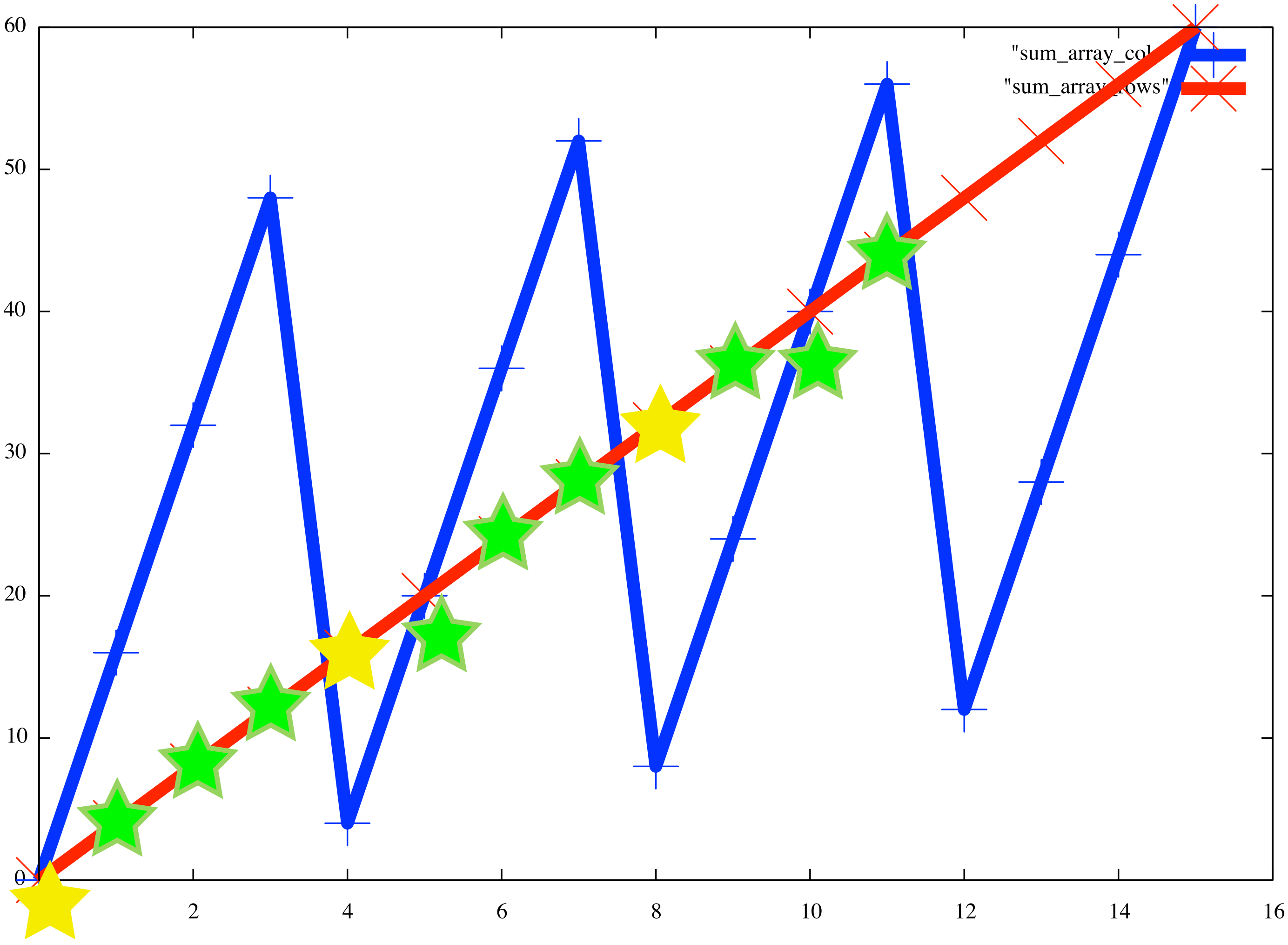
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

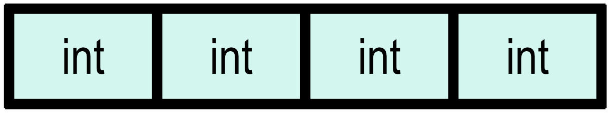
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



single line 16 byte cache

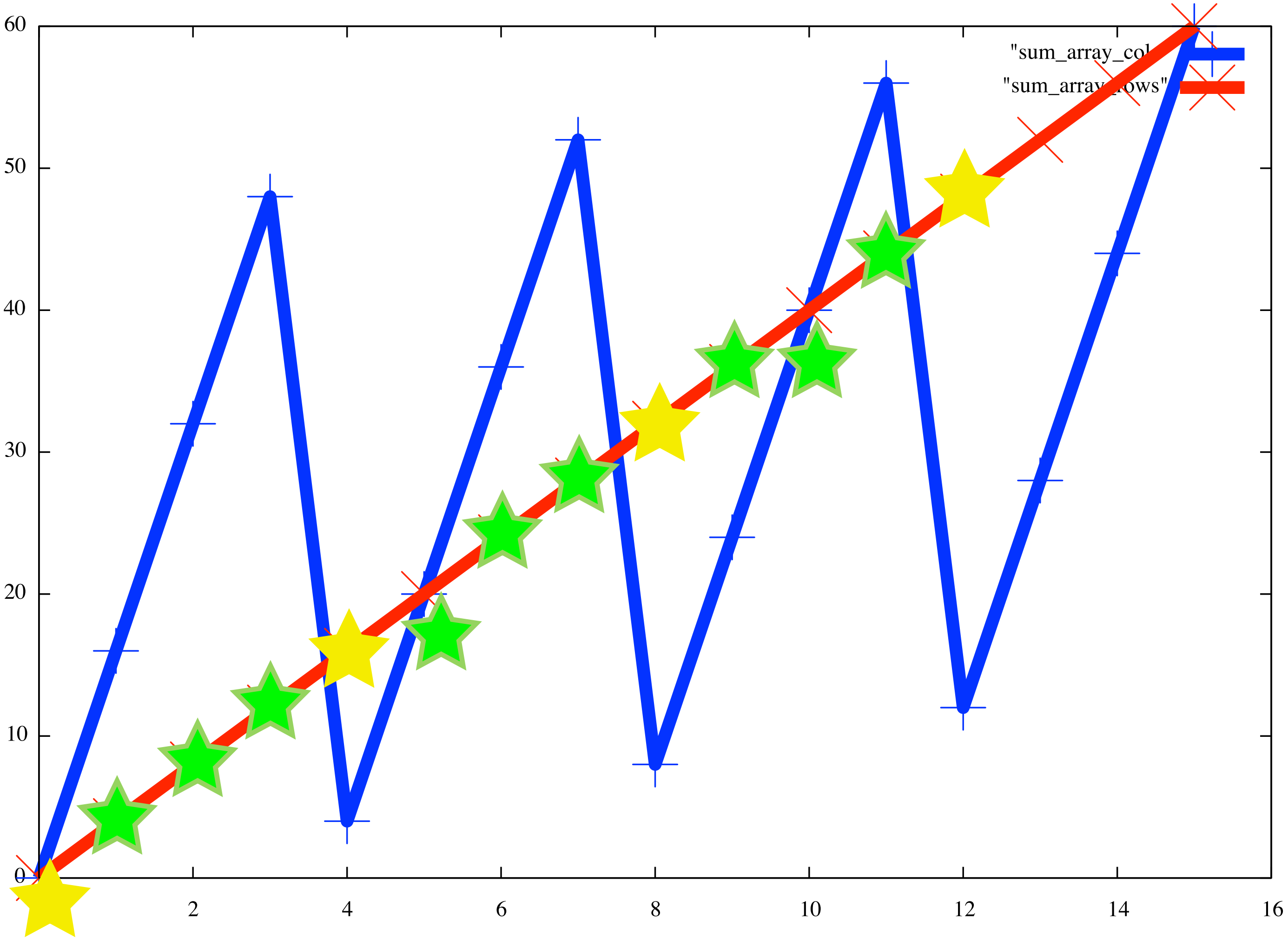


| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



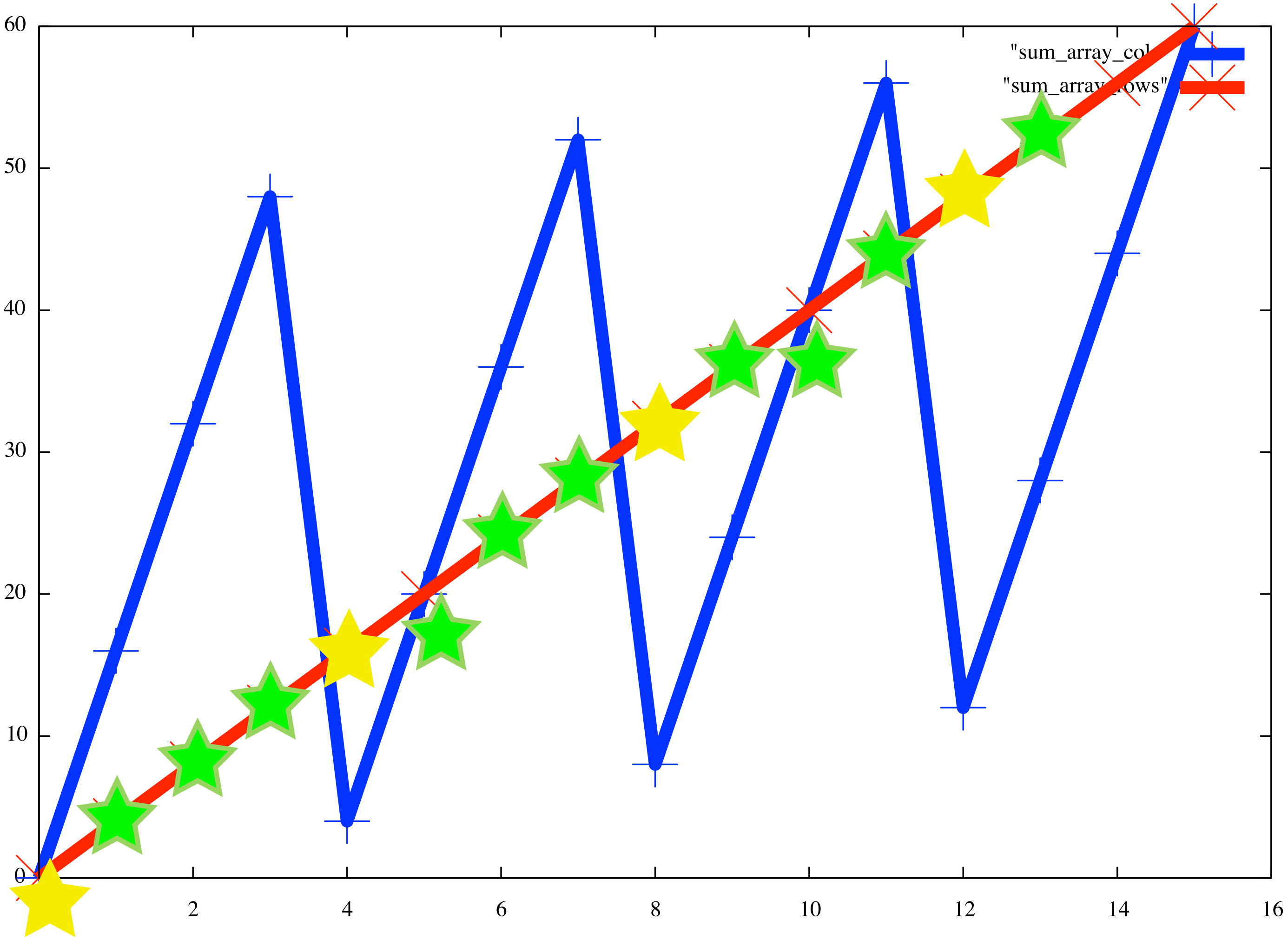
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

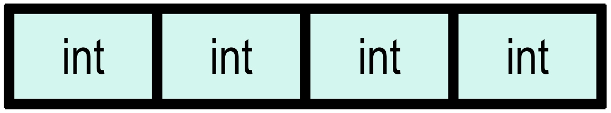
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



single line 16 byte cache

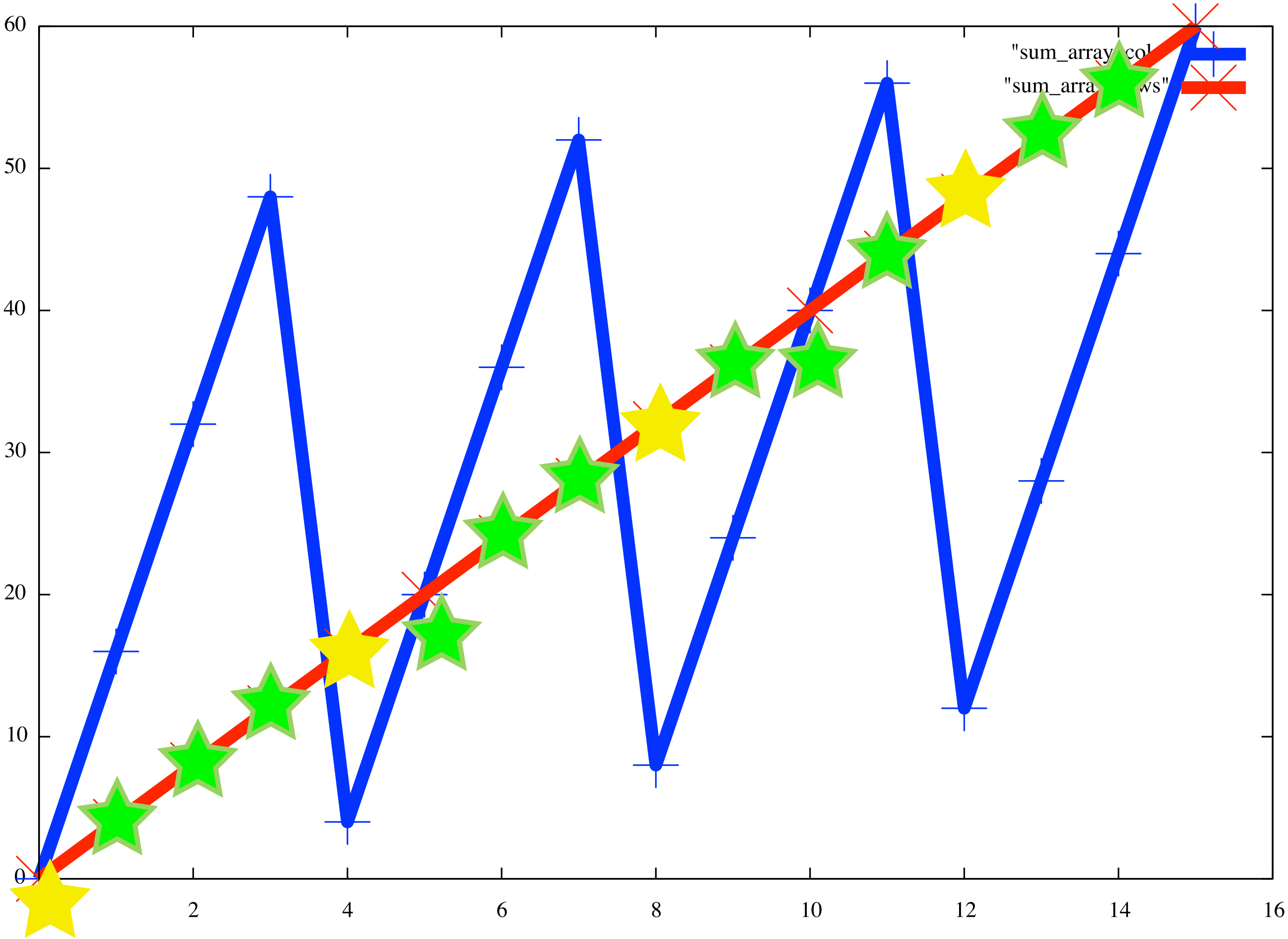


| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



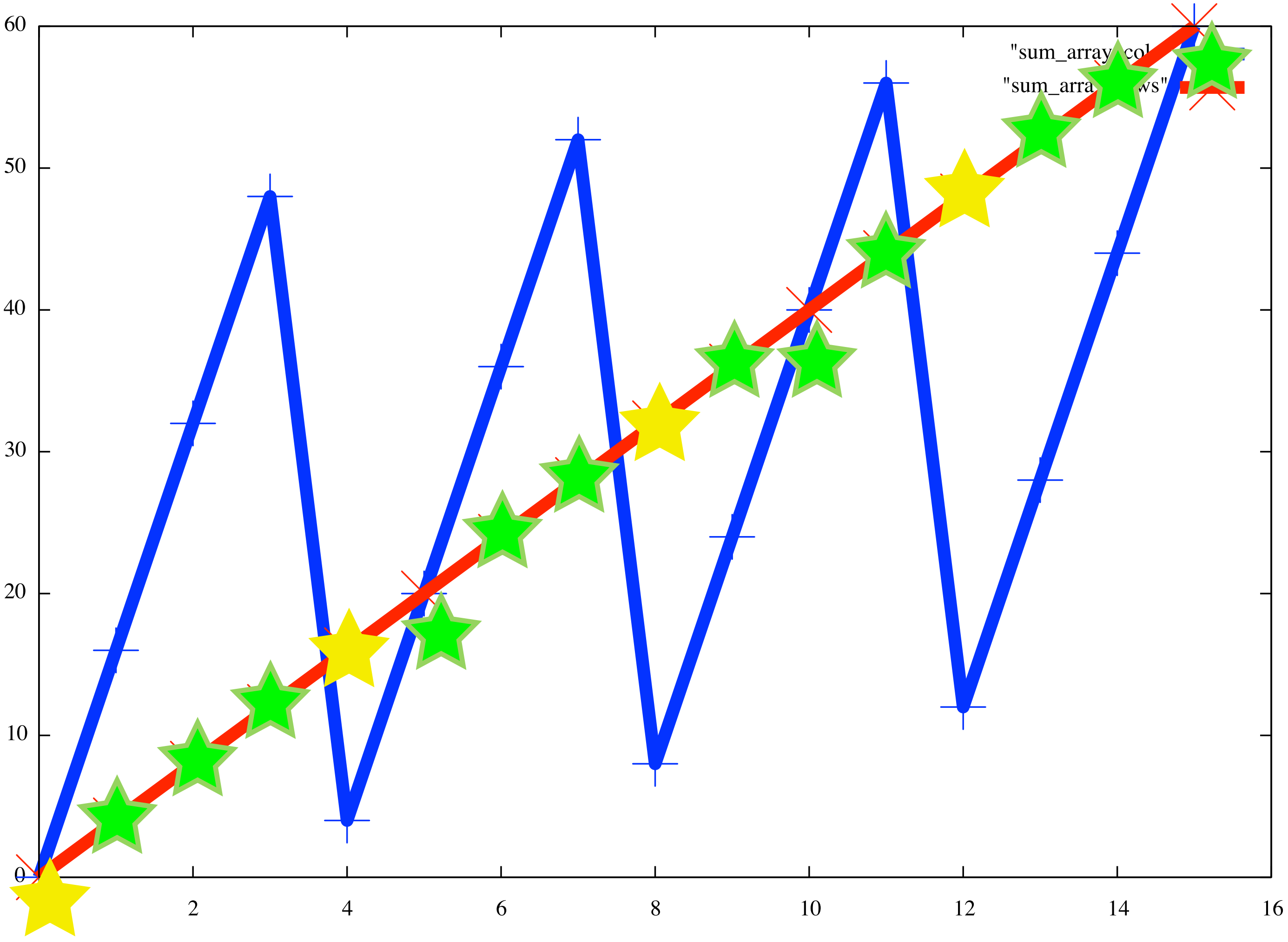
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

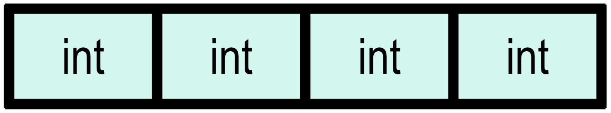
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

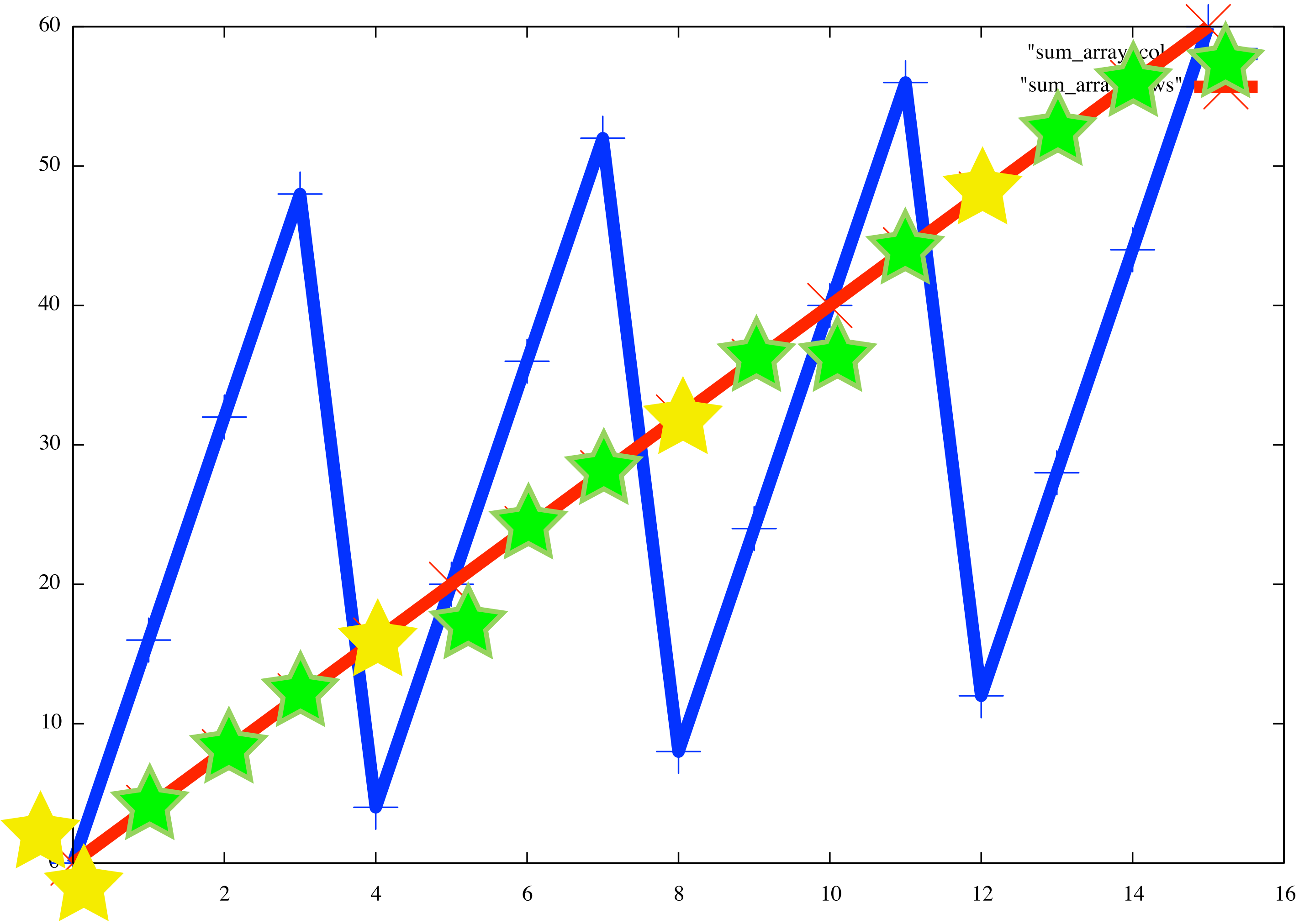


| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



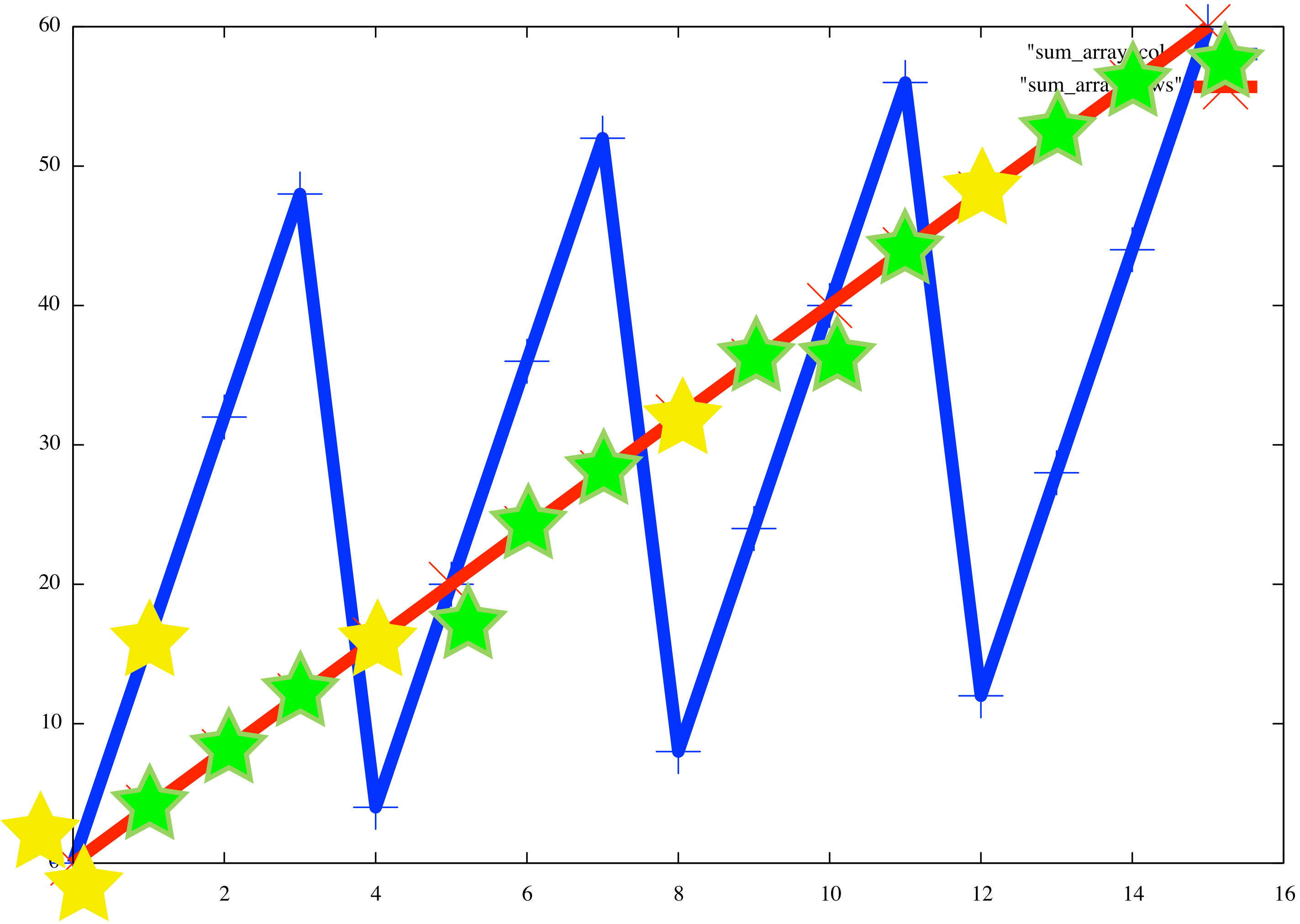
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

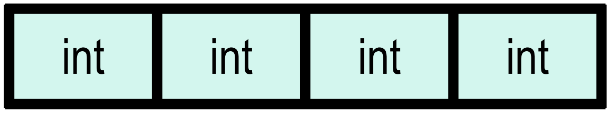
# Which one do you prefer?

```
        int sum_array_rows(int a[M][N])
                      {
                 int i, j, sum = 0;

               for (i = 0; i < M; i++)
                 for (j = 0; j < N; j++)
                       sum += a[i][j];
                     return sum;
                      }
```

```
        int sum_array_cols(int a[M][N])
                      {
                 int i, j, sum = 0;

               for (j = 0; j < N; j++)
                 for (i = 0; i < M; i++)
                       sum += a[i][j];
                     return sum;
                      }
```



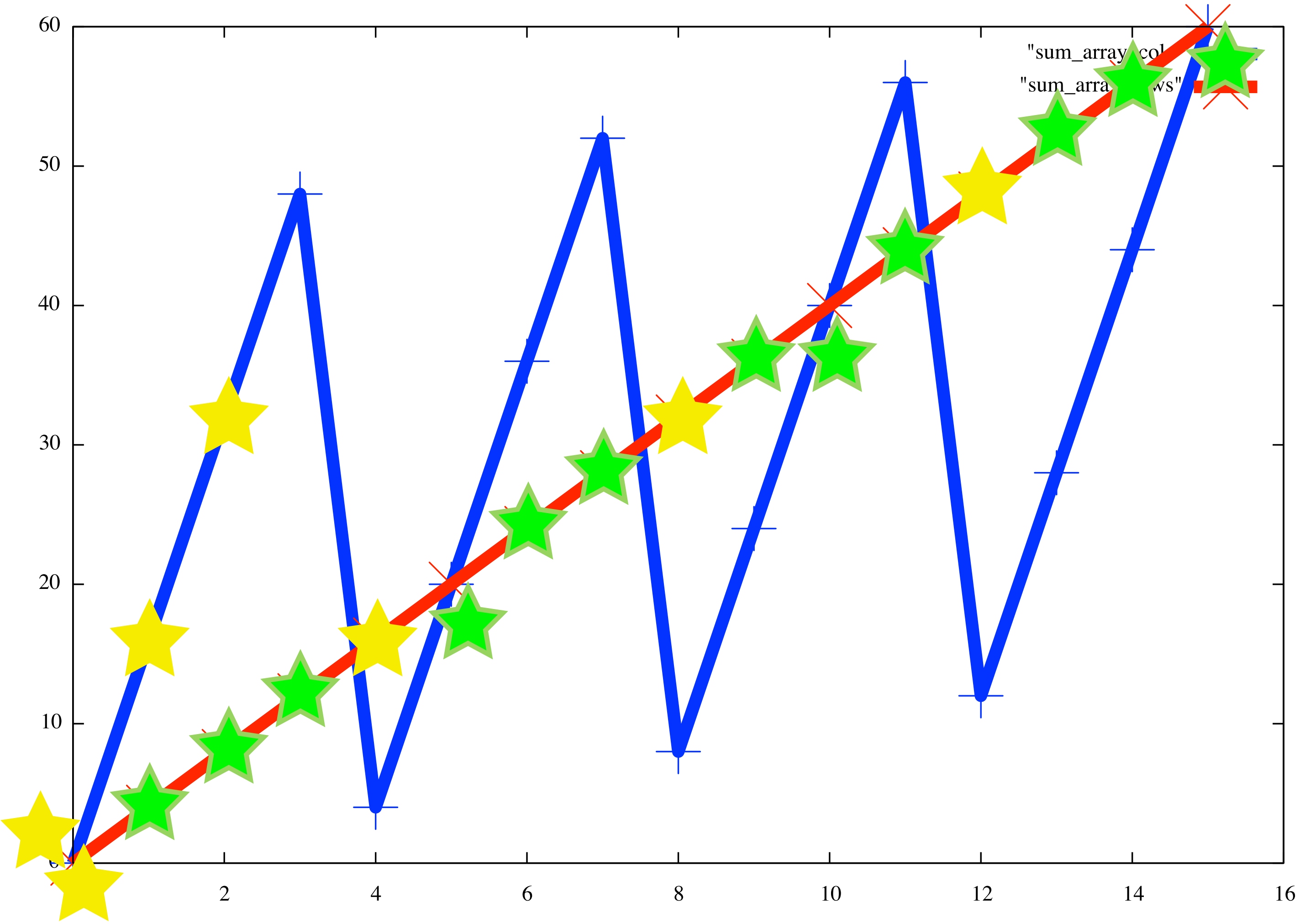
single line 16 byte cache



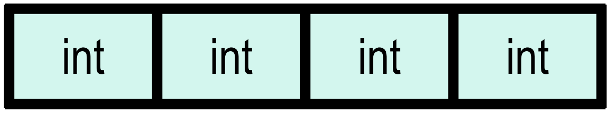
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

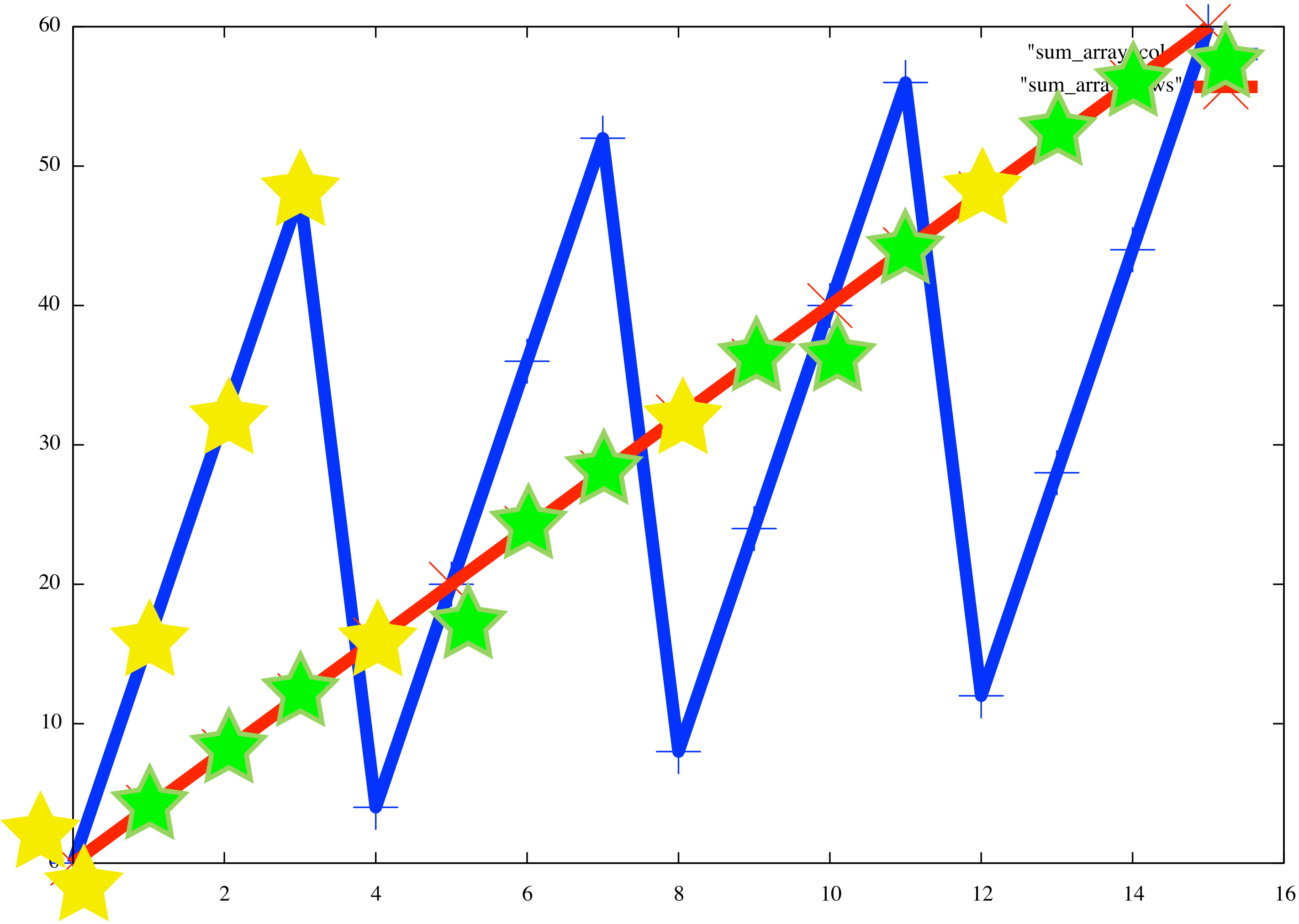


| | | | |
|---|---|---|---|
| int | int | int | int |

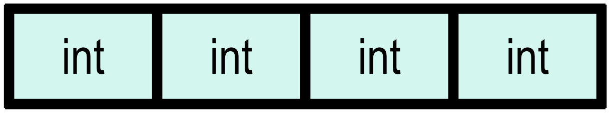
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



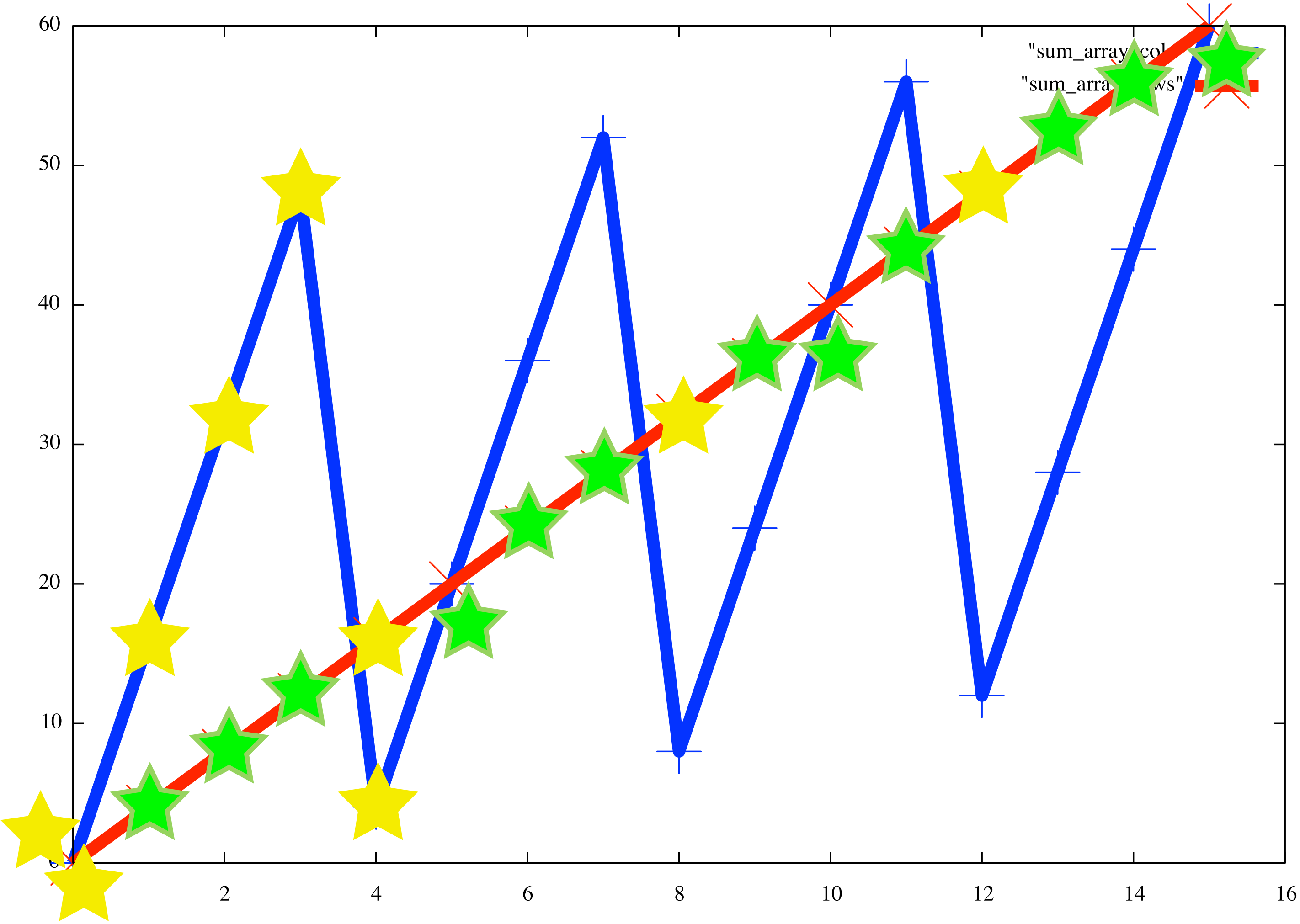
single line 16 byte cache



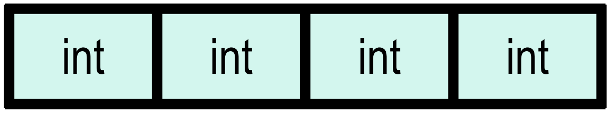
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

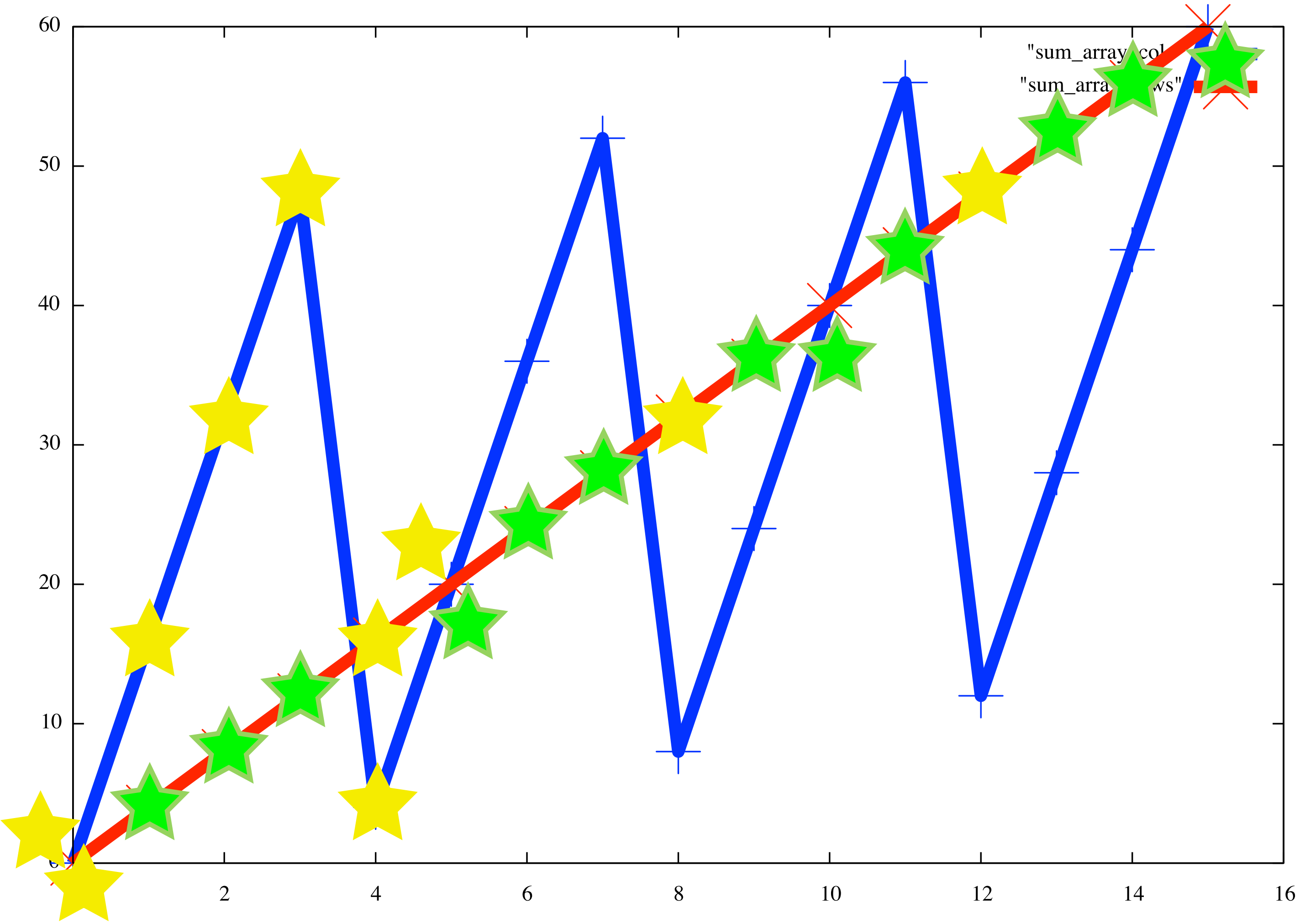


| int | int | int | int |
|---|---|---|---|

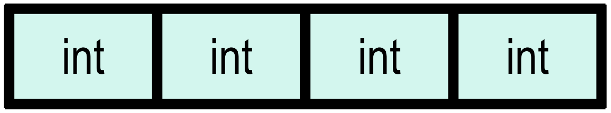
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

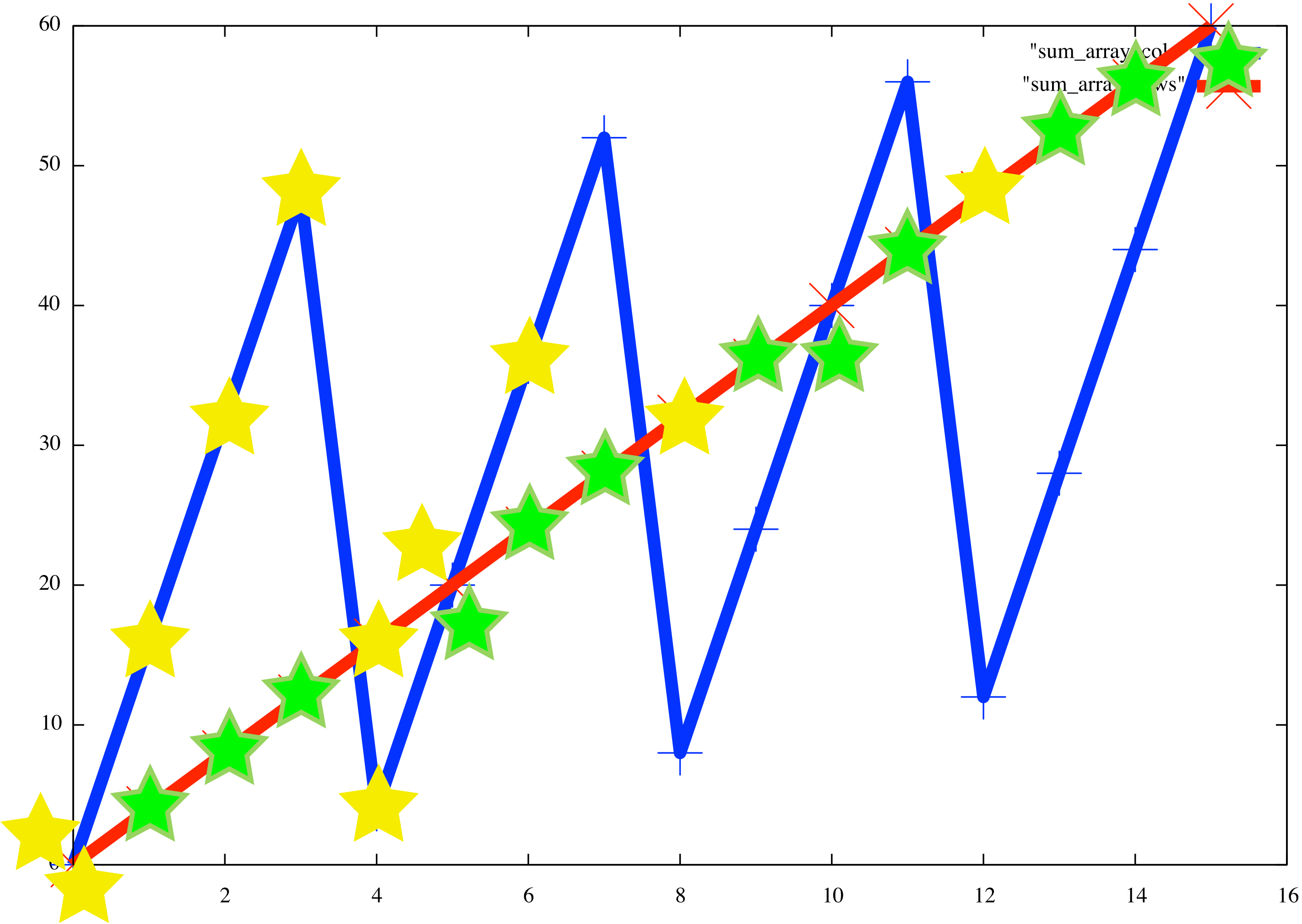


|  |  |  |  |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



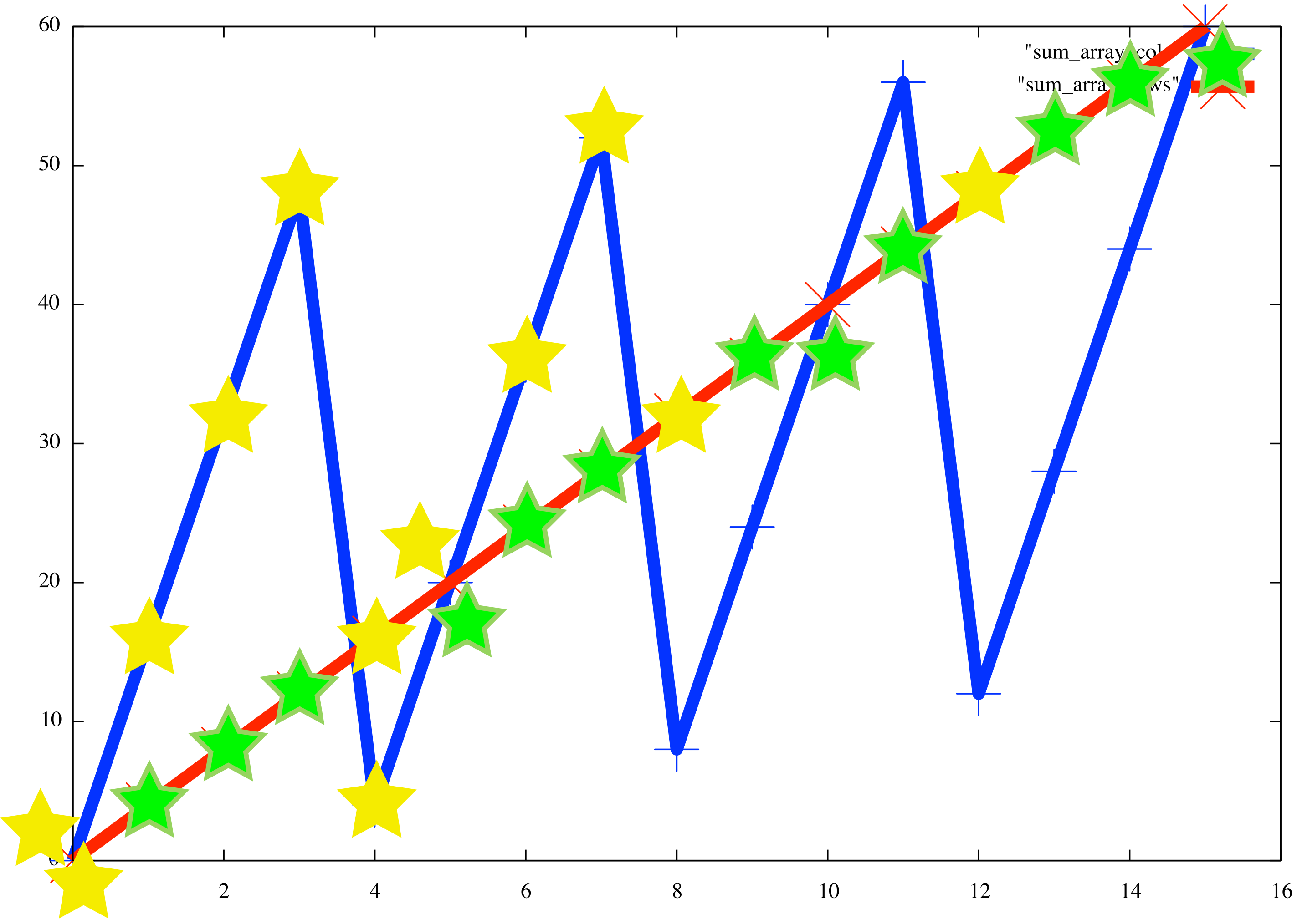
single line 16 byte cache

|  |  |  |  |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



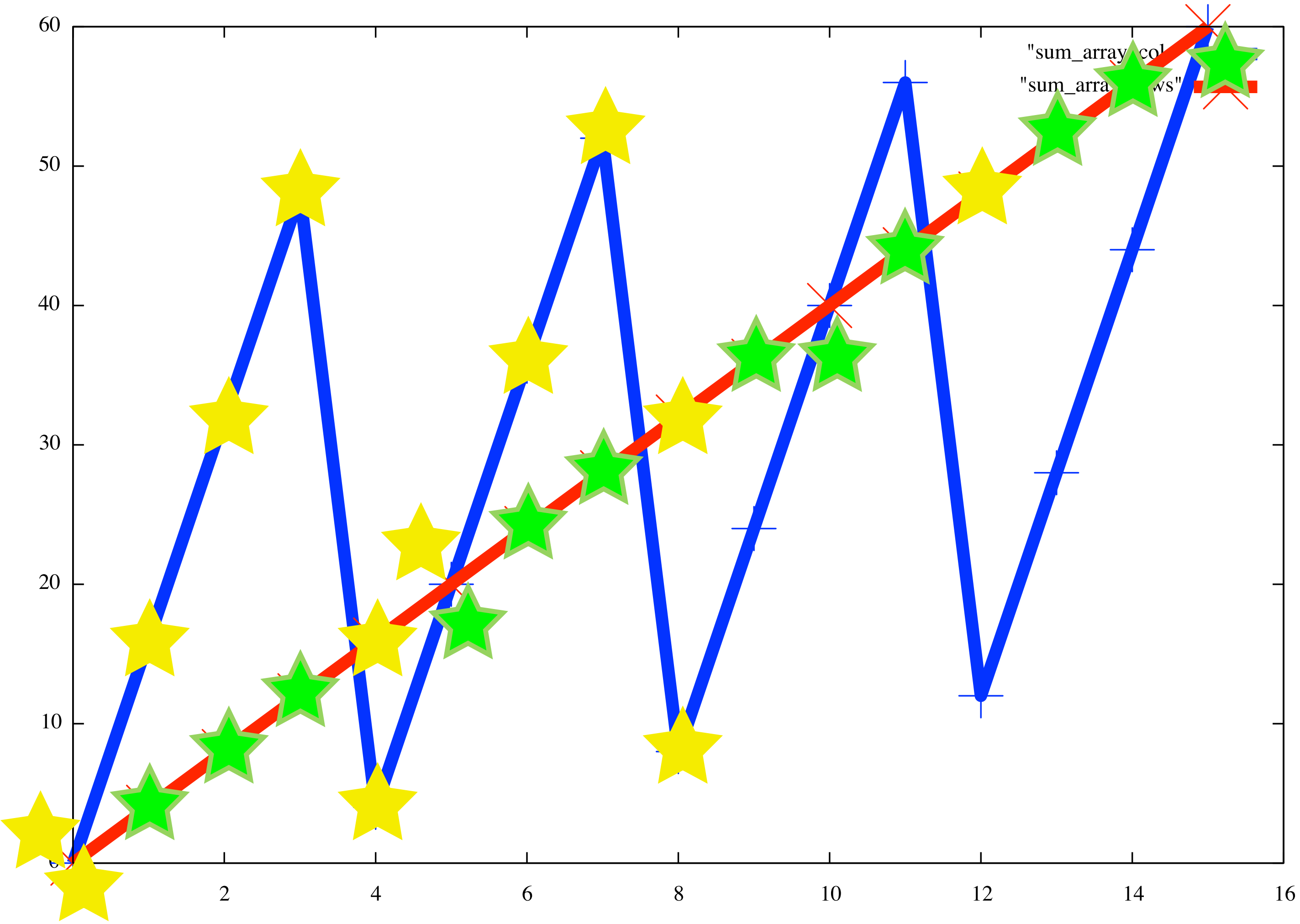
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

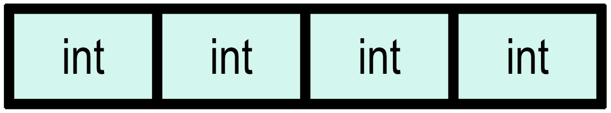
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
          sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
      for (i = 0; i < M; i++)
          sum += a[i][j];
    return sum;
}
```



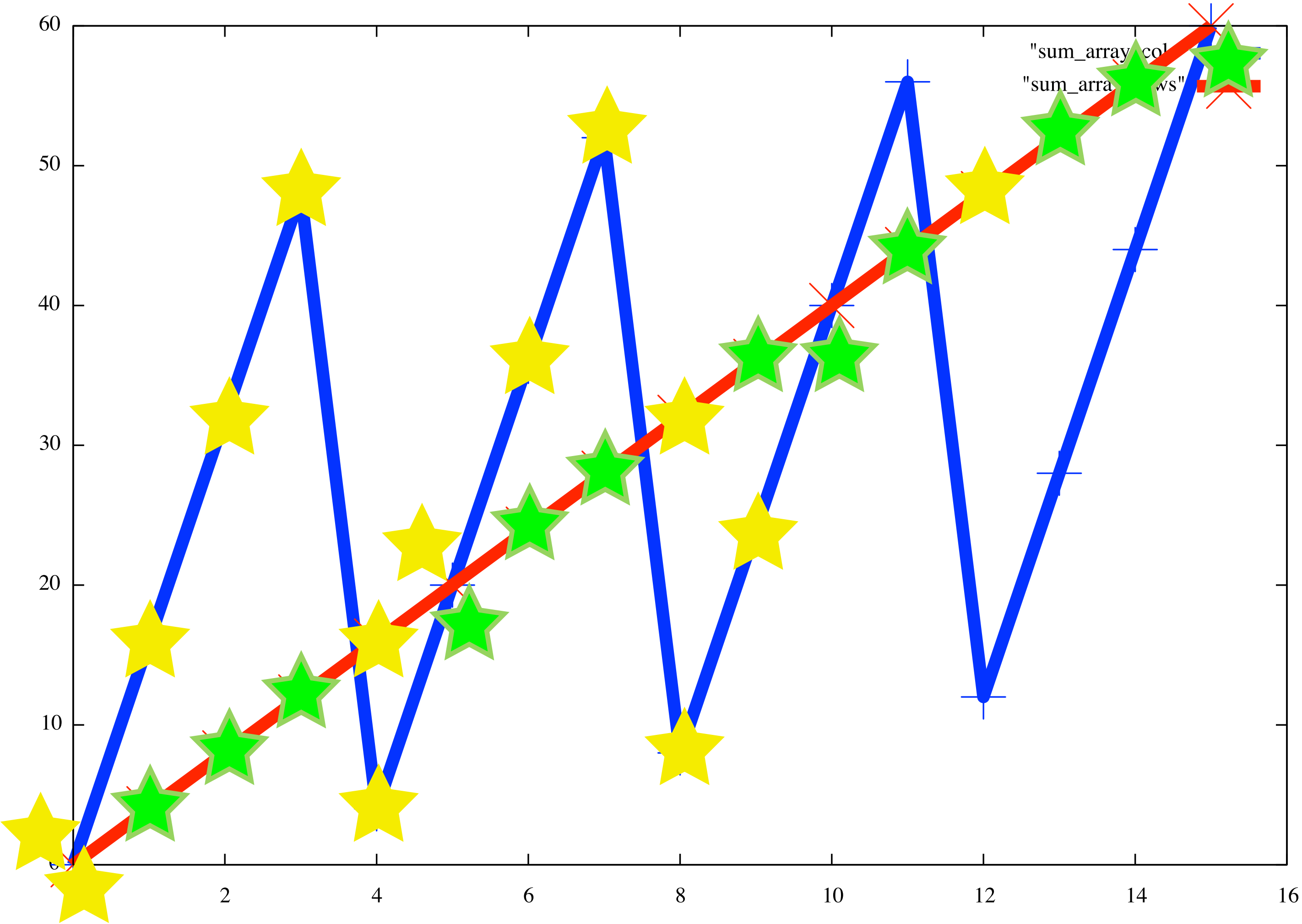
single line 16 byte cache



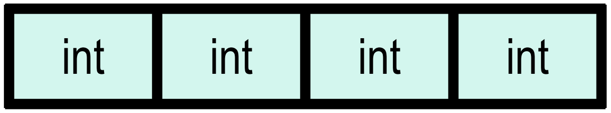
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

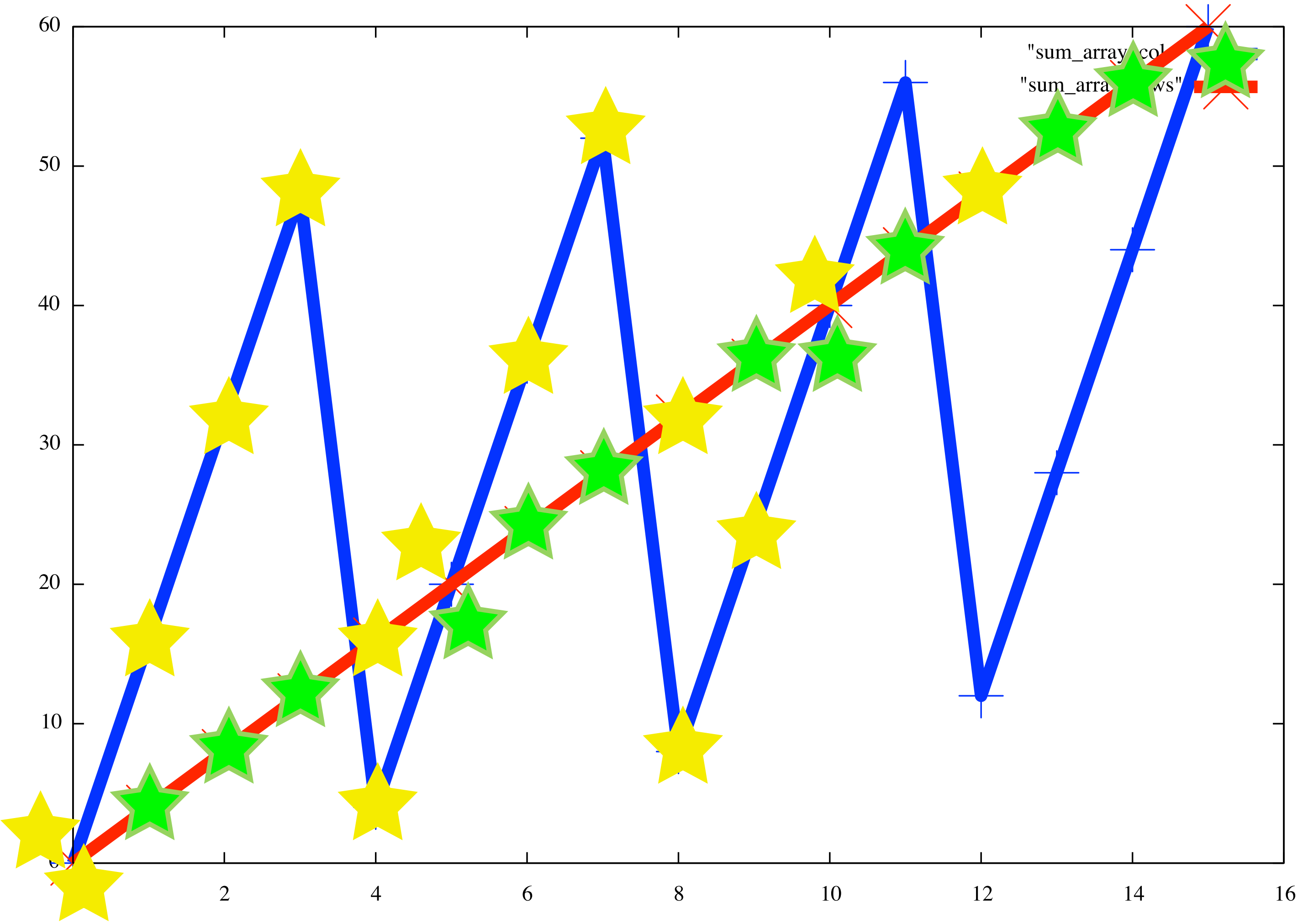


| | | | |
|---|---|---|---|
| int | int | int | int |

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



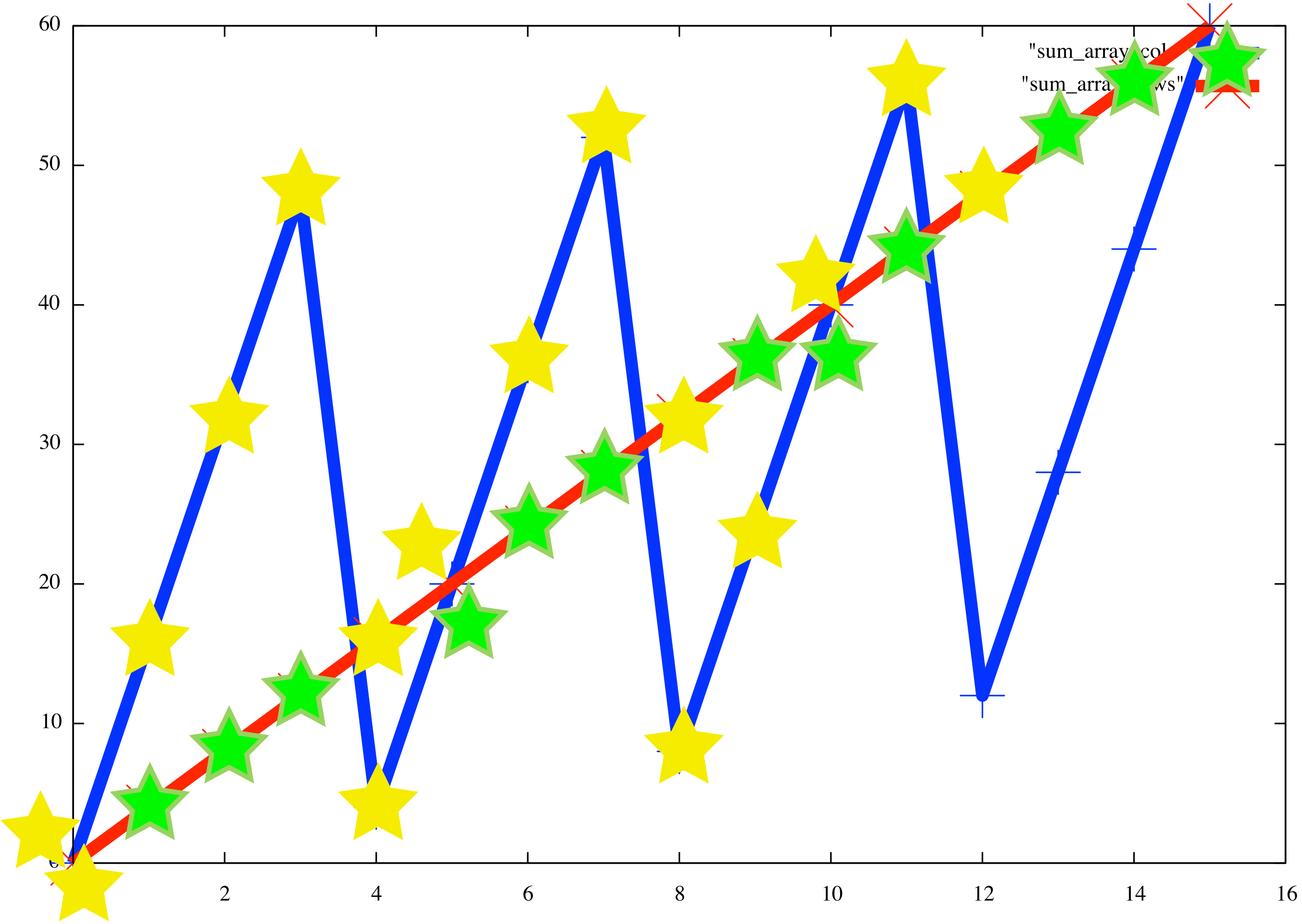
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

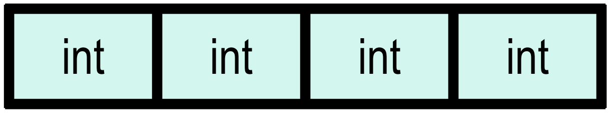
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



single line 16 byte cache

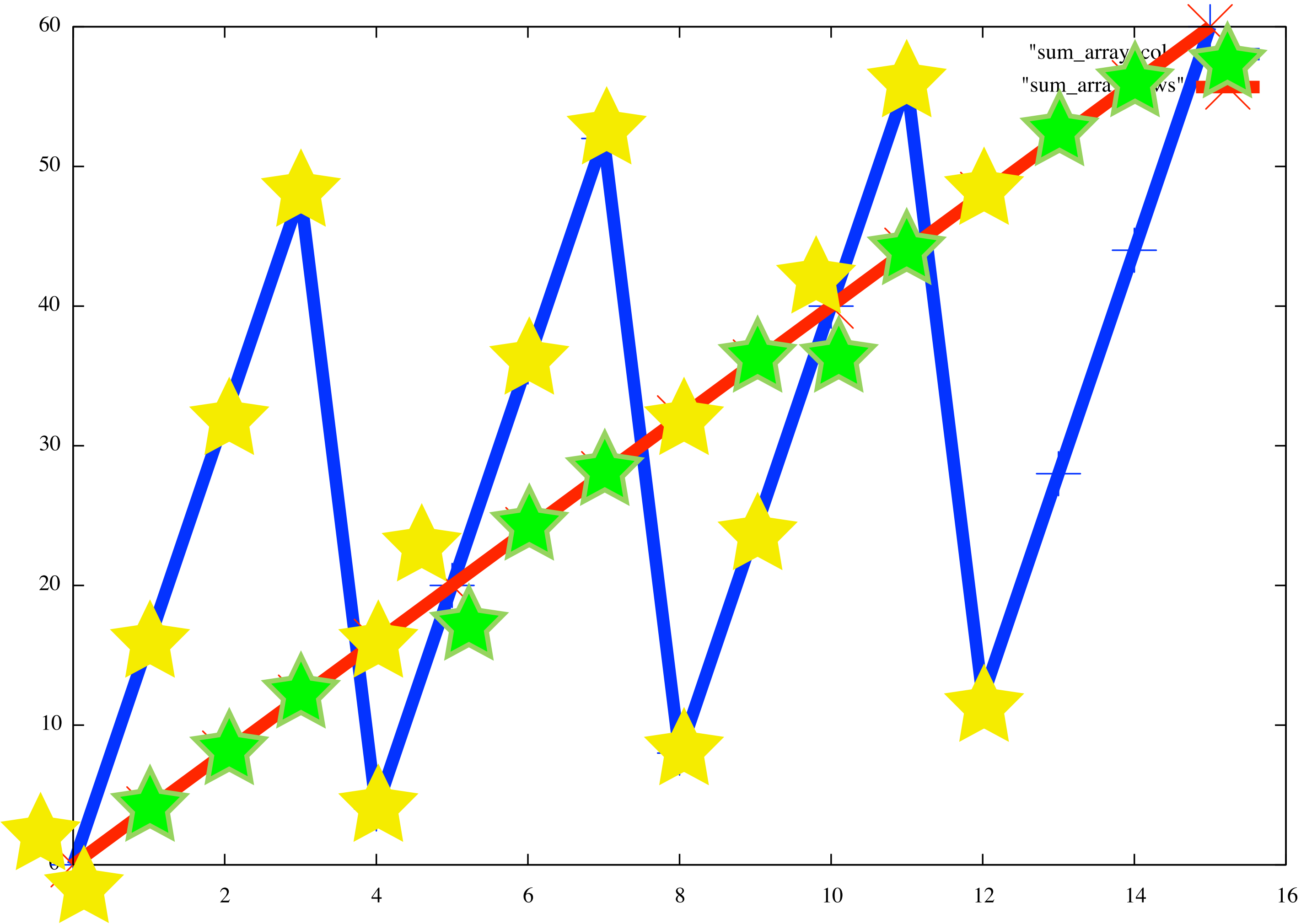


| int | int | int | int |
|---|---|---|---|

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



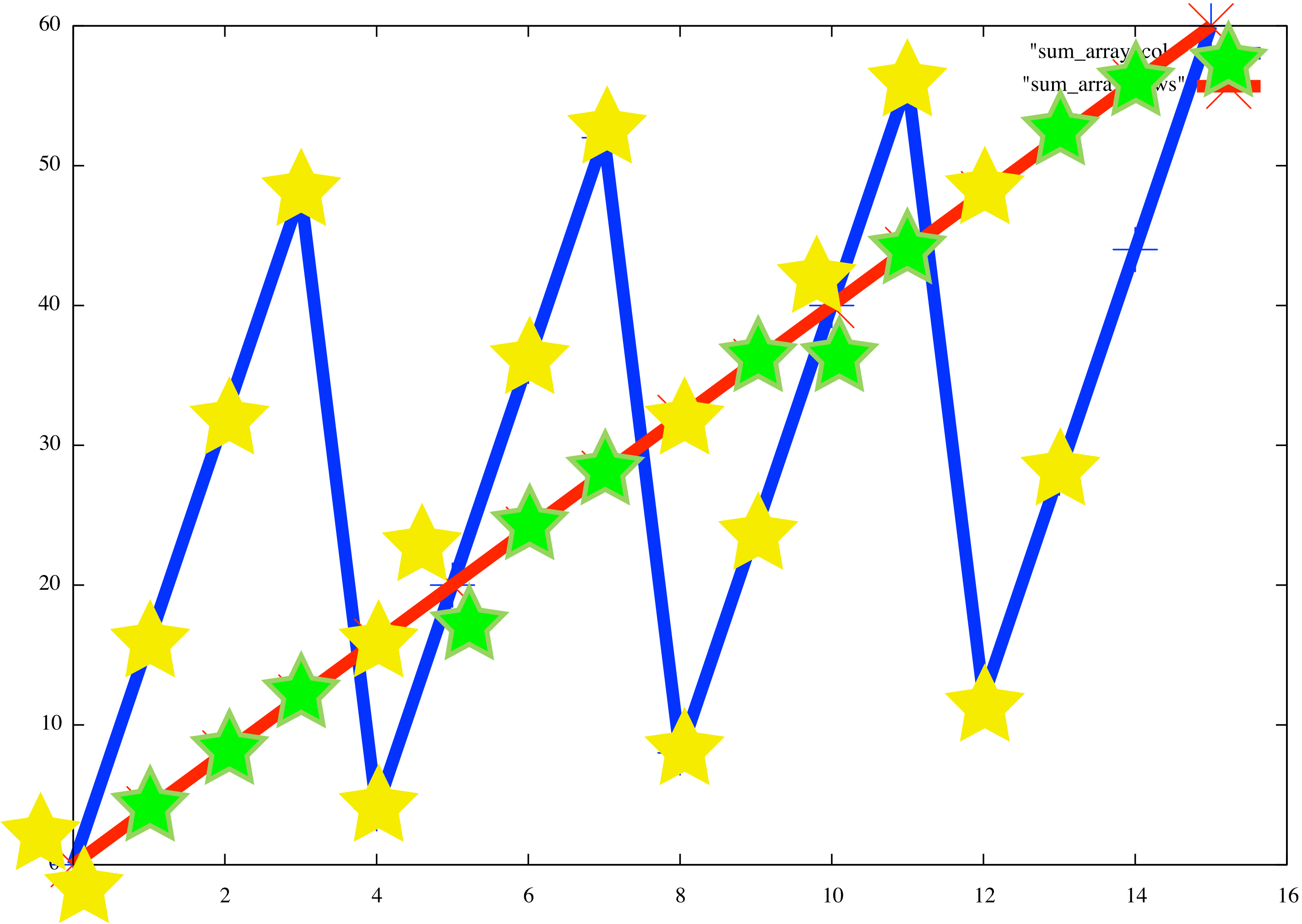
single line 16 byte cache

| | | | |
|---|---|---|---|
| int | int | int | int |

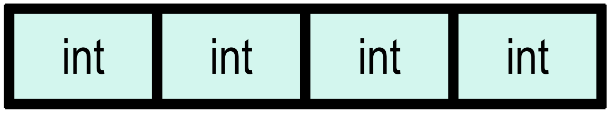
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
               {
        int i, j, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
              sum += a[i][j];
            return sum;
               }
```

```
int sum_array_cols(int a[M][N])
               {
        int i, j, sum = 0;

      for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
              sum += a[i][j];
            return sum;
               }
```



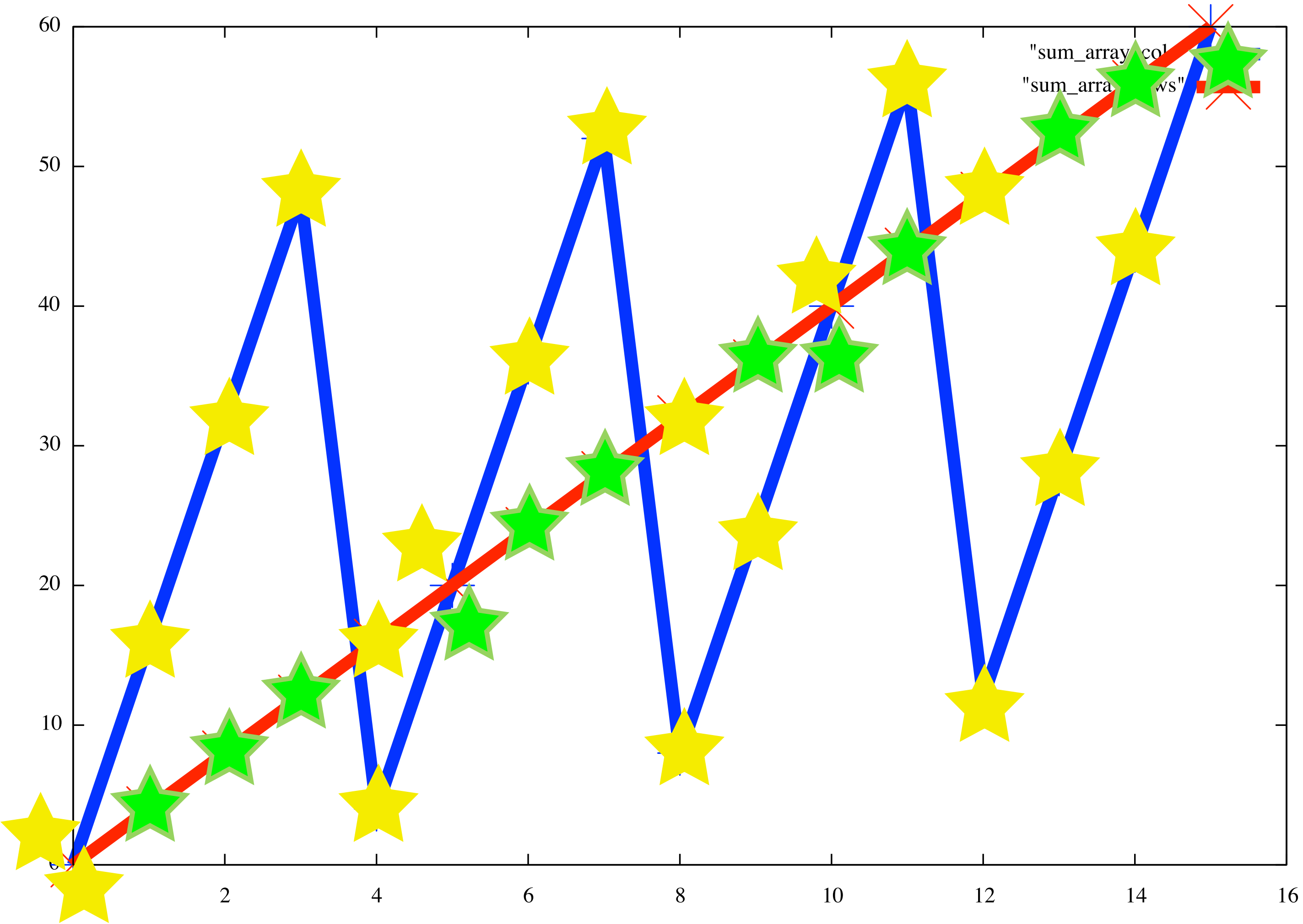
single line 16 byte cache



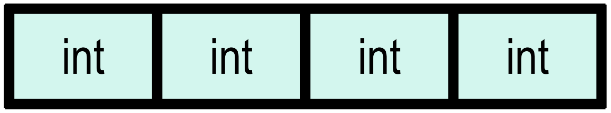
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
            sum += a[i][j];
        return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
      for (i = 0; i < M; i++)
            sum += a[i][j];
        return sum;
}
```



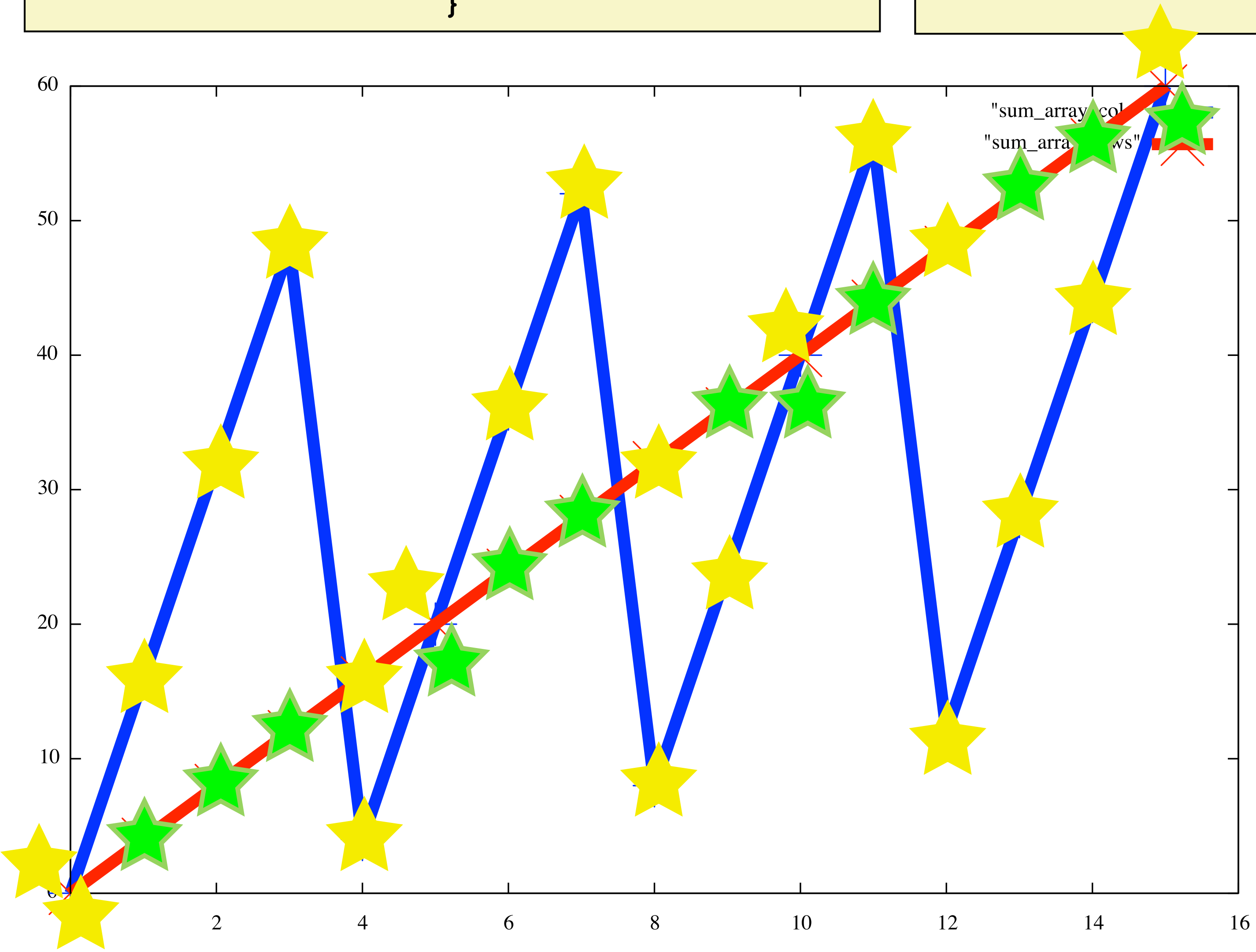
single line 16 byte cache



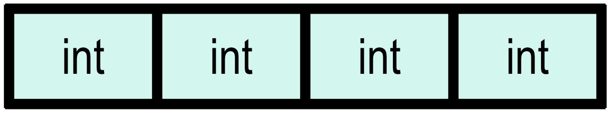
# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

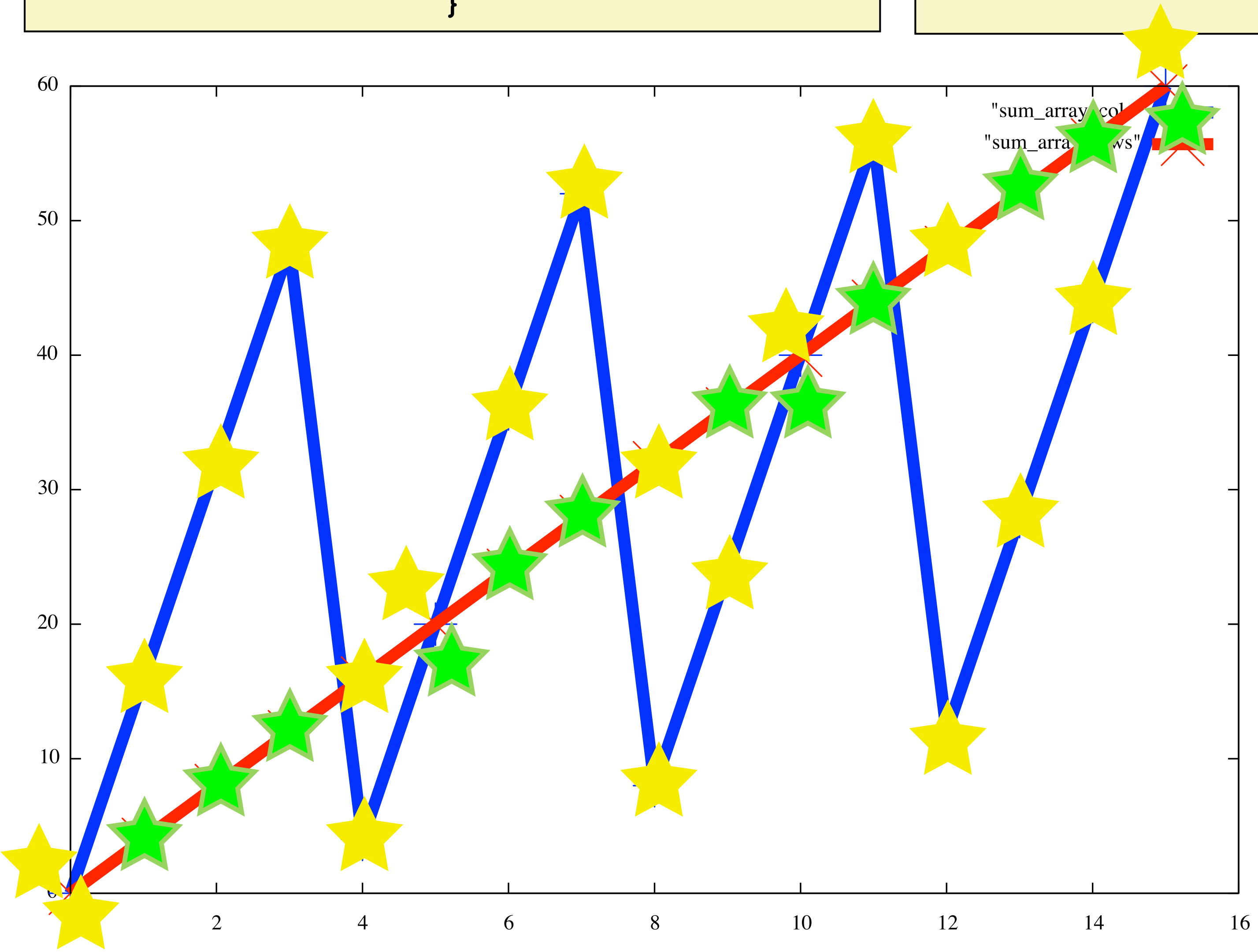


int int int int

# Which one do you prefer?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



single line 16 byte cache

| int | int | int | int |
|---|---|---|---|

|  | sum array rows | sum array cols |
|---|---|---|
| HITS | 12 | 0 |
| MISSES | 4 | 16 |

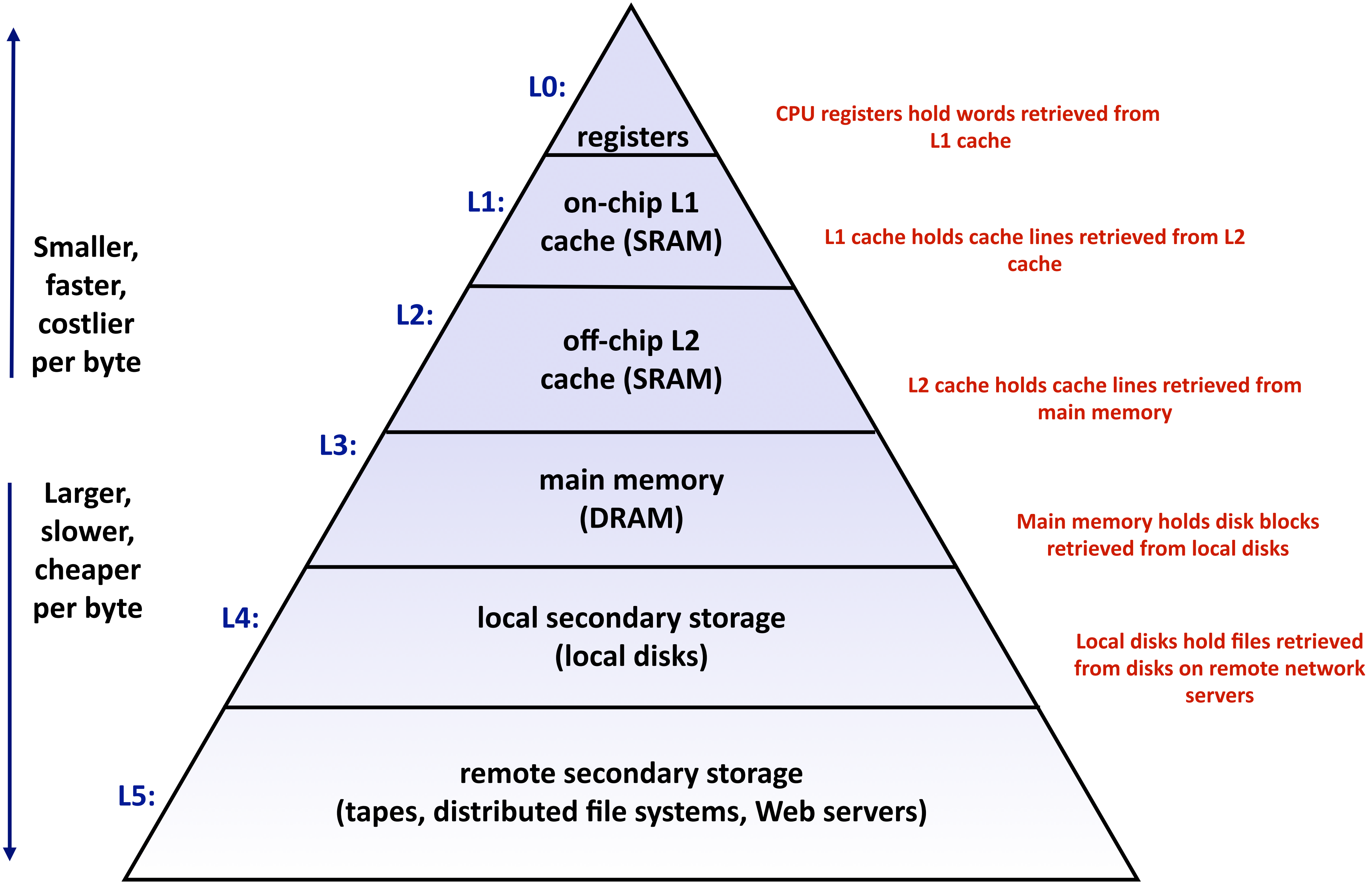
# Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software systems:**
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True of registers ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality

- **These properties complement each other beautifully**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy**

# An Example Memory Hierarchy



Smaller,
faster,
costlier
per byte

Larger,
slower,
cheaper
per byte

L0: registers

L1: on-chip L1 cache (SRAM)

L2: off-chip L2 cache (SRAM)

L3: main memory (DRAM)

L4: local secondary storage (local disks)

L5: remote secondary storage (tapes, distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

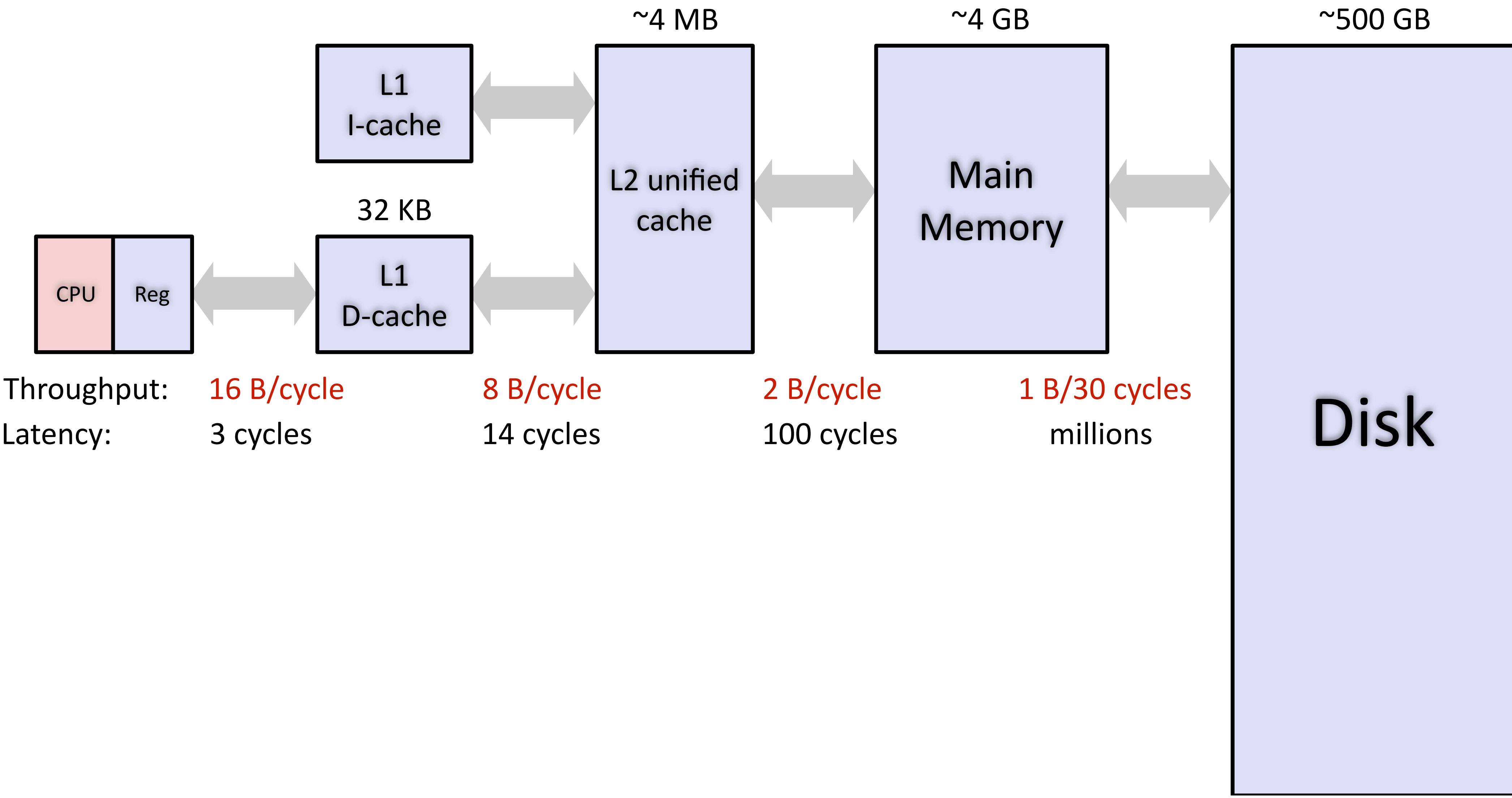
# Examples of Caching in the Hierarchy

| Cache Type | What is Cached? | Where is it Cached? | Latency (cycles) | Managed By |
|---|---|---|---|---|
| Registers | 4-byte words | CPU core | 0 | Compiler |
| TLB | Address translations | On-Chip TLB | 0 | Hardware |
| L1 cache | 64-bytes block | On-Chip L1 | 1 | Hardware |
| L2 cache | 64-bytes block | Off-Chip L2 | 10 | Hardware |
| Virtual Memory | 4-KB page | Main memory | 100 | Hardware+OS |
| Buffer cache | Parts of files | Main memory | 100 | OS |
| Network buffer cache | Parts of files | Local disk | 10,000,000 | AFS/NFS client |
| Browser cache | Web pages | Local disk | 10,000,000 | Web browser |
| Web cache | Web pages | Remote server disks | 1,000,000,000 | Web proxy server |

# Memory Hierarchy: Core 2 Duo

*Not drawn to scale*

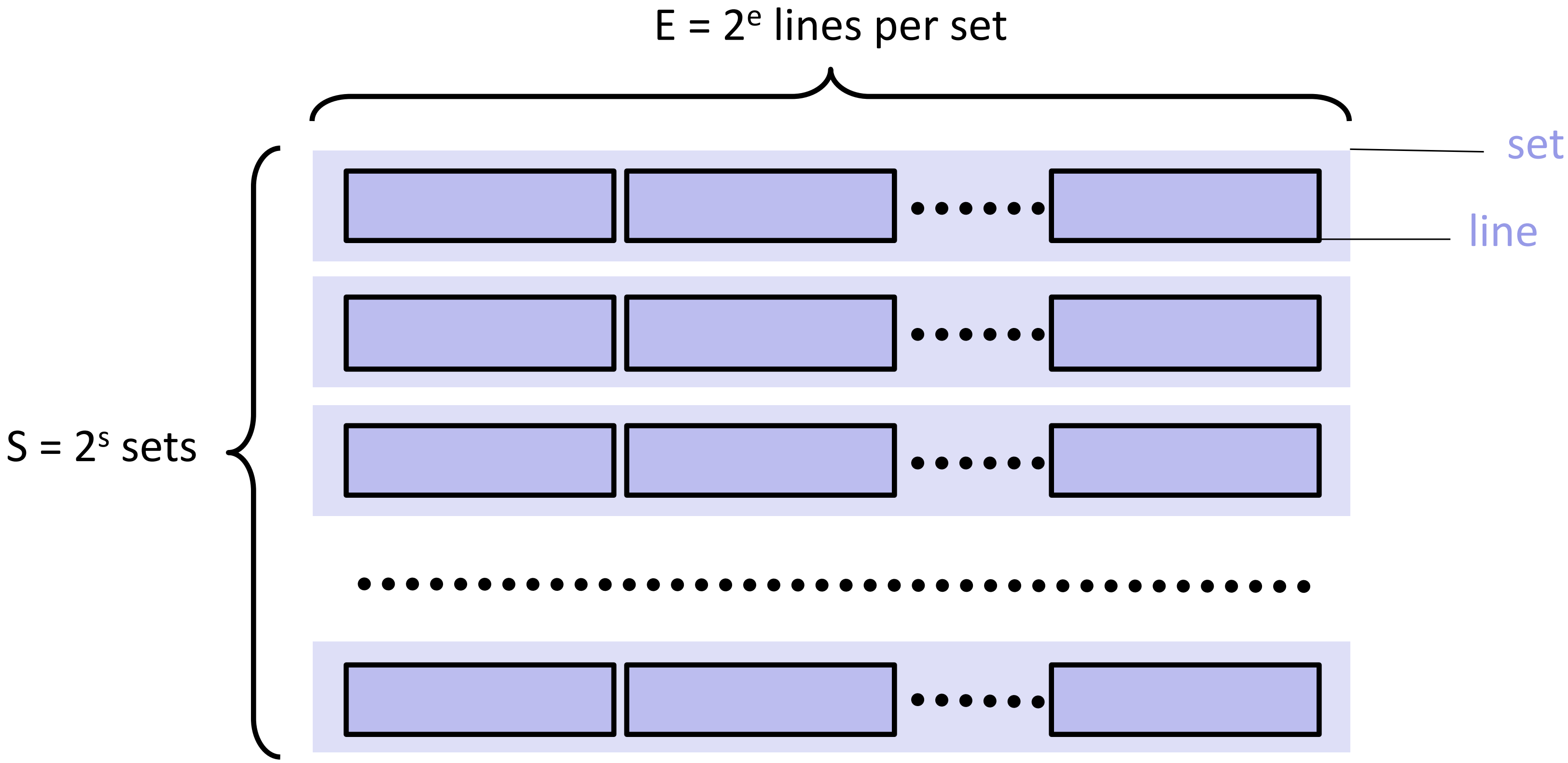
L1/L2 cache: 64 B blocks



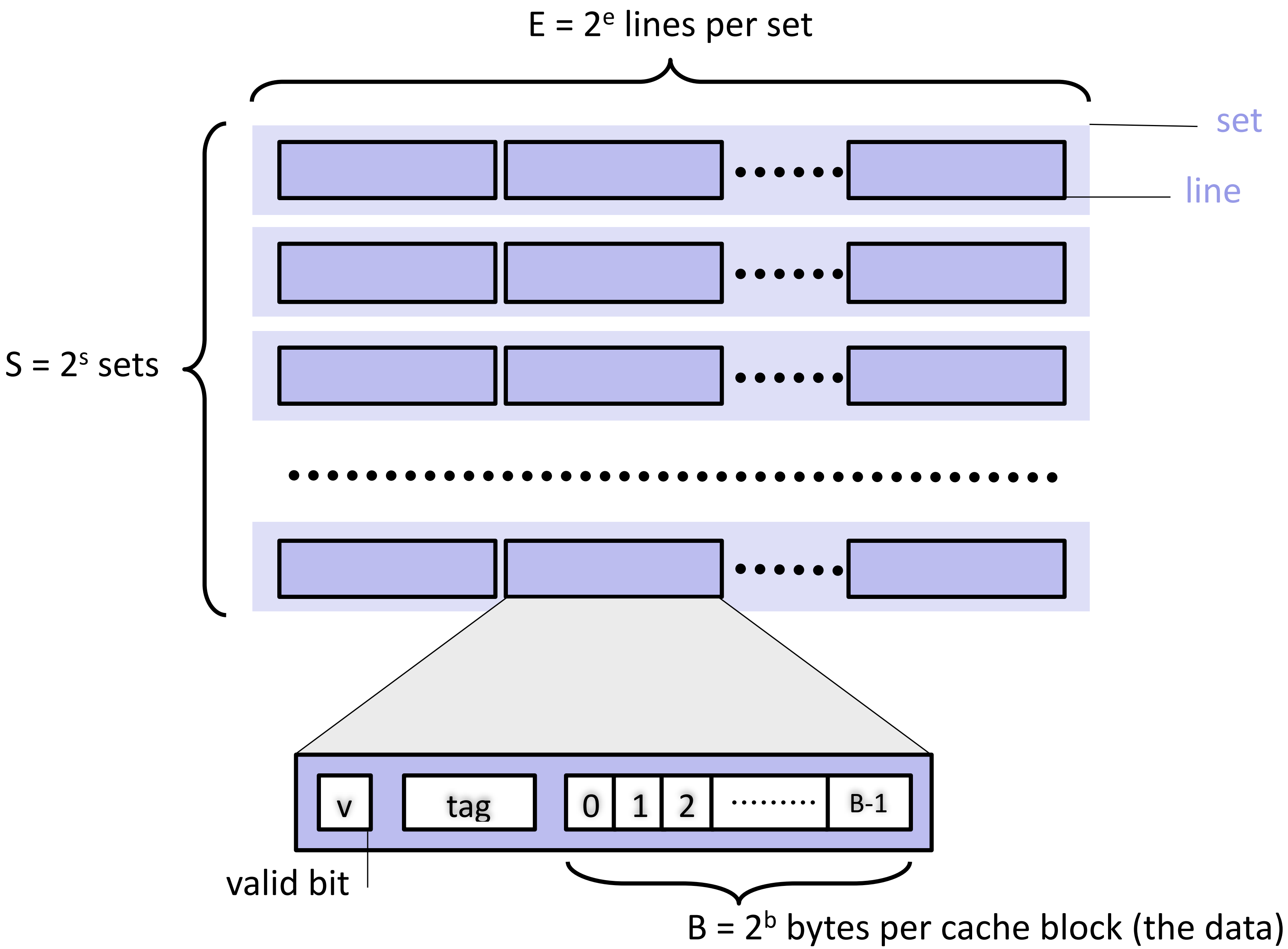
| | | | | |
|---|---|---|---|---|
| Throughput: | 16 B/cycle | 8 B/cycle | 2 B/cycle | 1 B/30 cycles |
| Latency: | 3 cycles | 14 cycles | 100 cycles | millions |

# CACHE ORGANIZATION

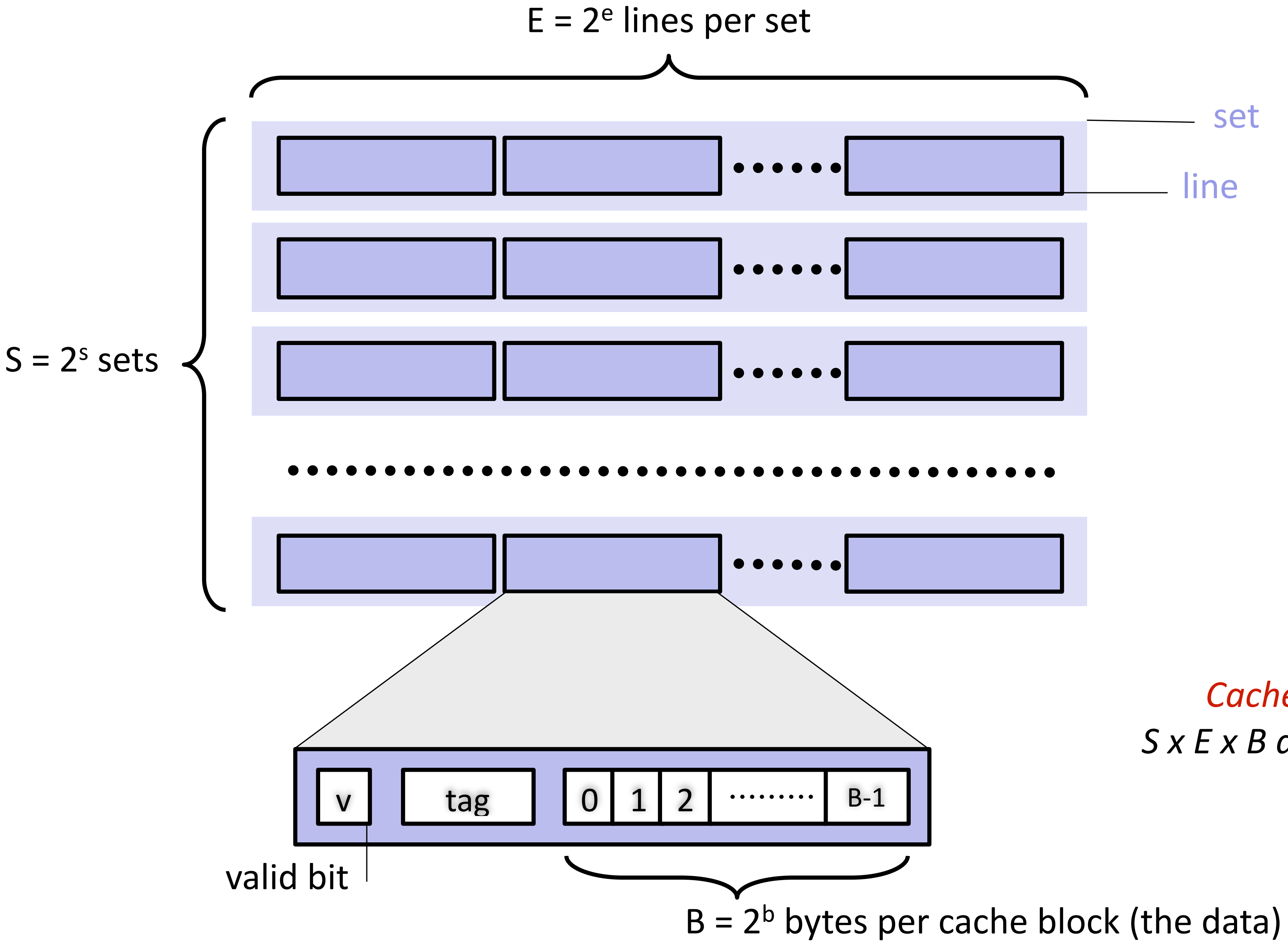
# General Cache Organization (S, E, B)



# General Cache Organization (S, E, B)



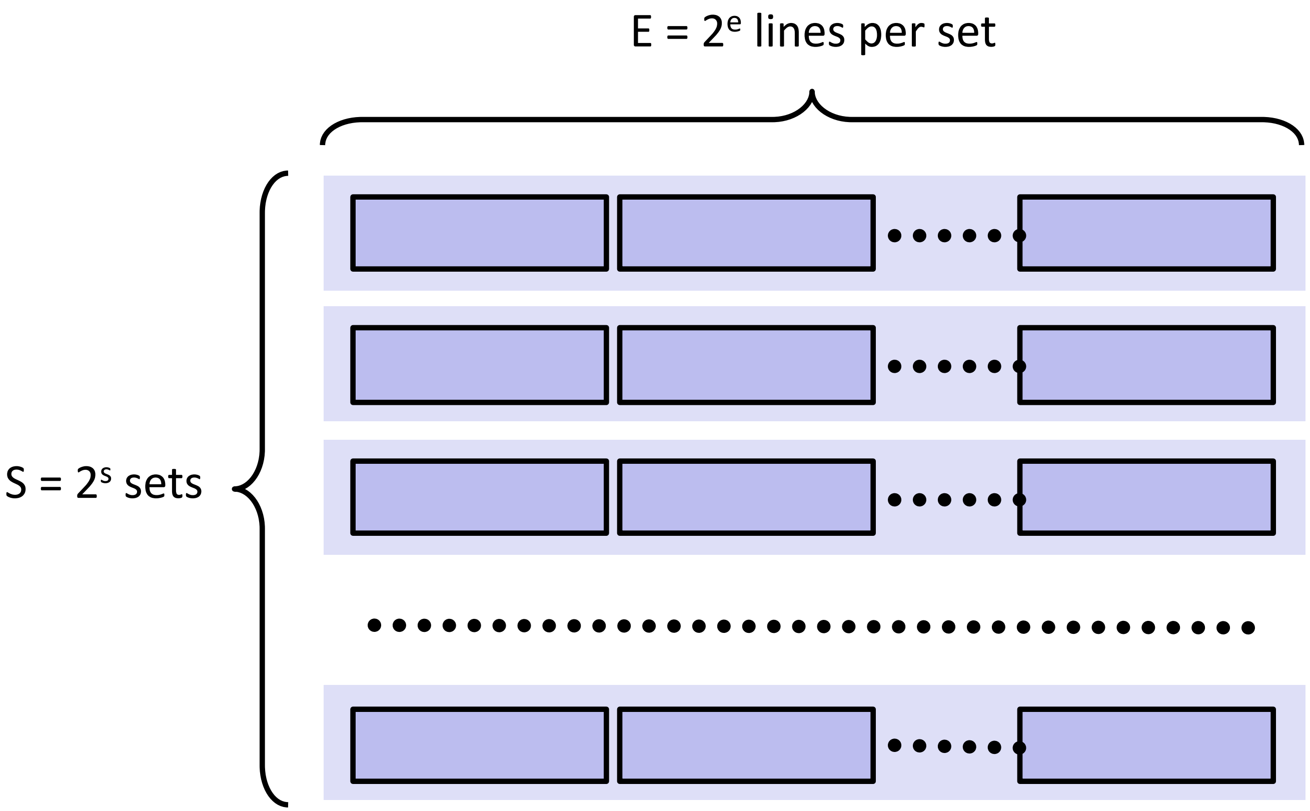
# General Cache Organization (S, E, B)



*Cache size:*

*S x E x B data bytes*

# Cache Read



E = $2^e$ lines per set

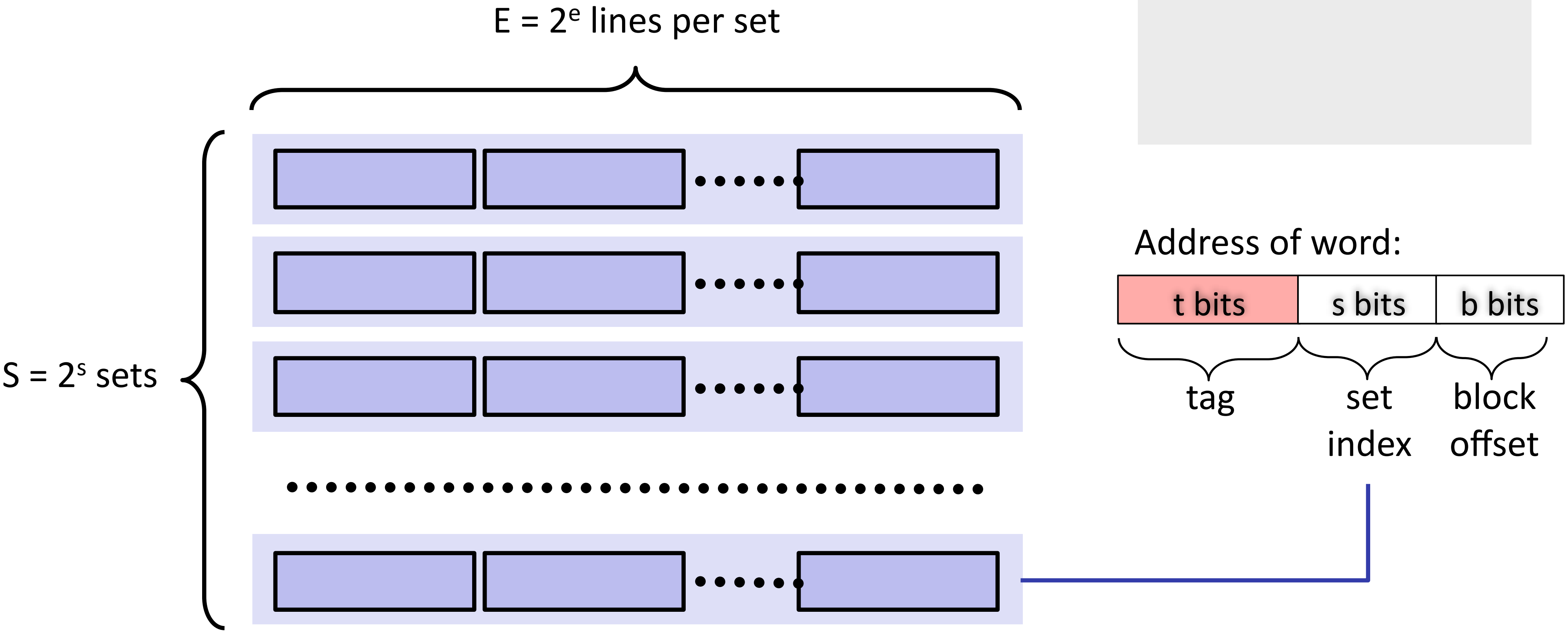
S = $2^s$ sets

Address of word:

| t bits | s bits | b bits |
|---|---|---|
| tag | set index | block offset |

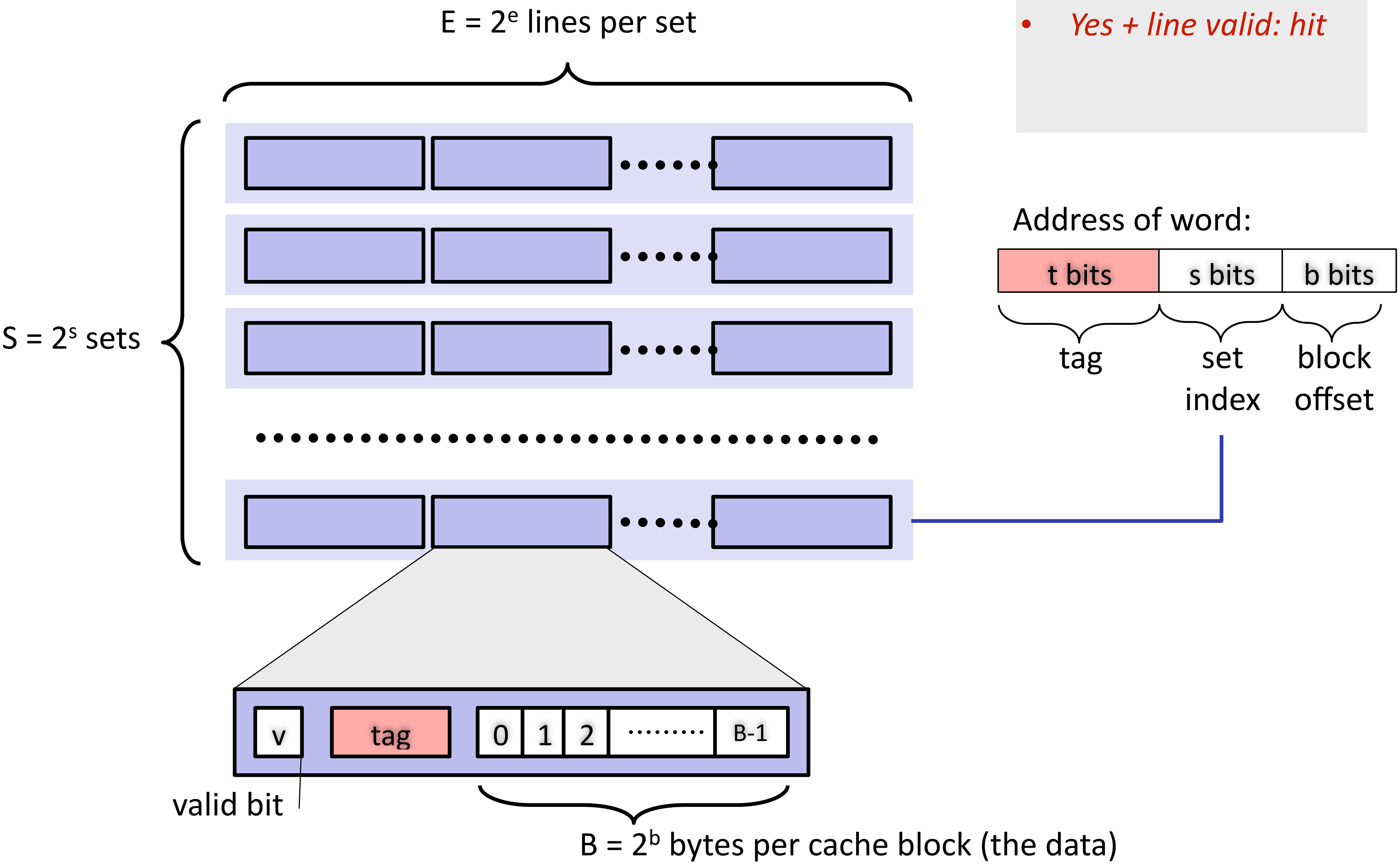
# Cache Read

- *Locate set*



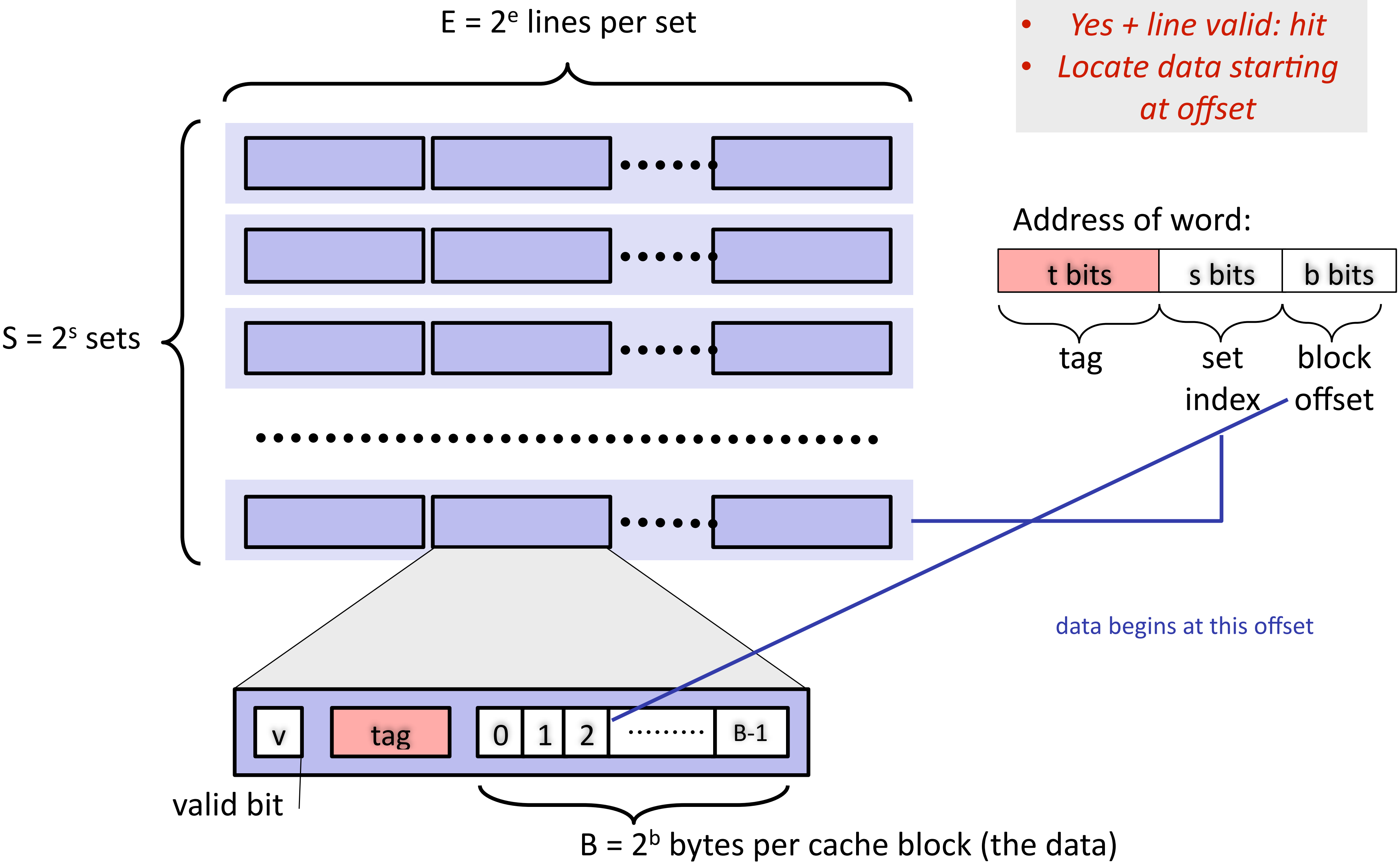
# Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*



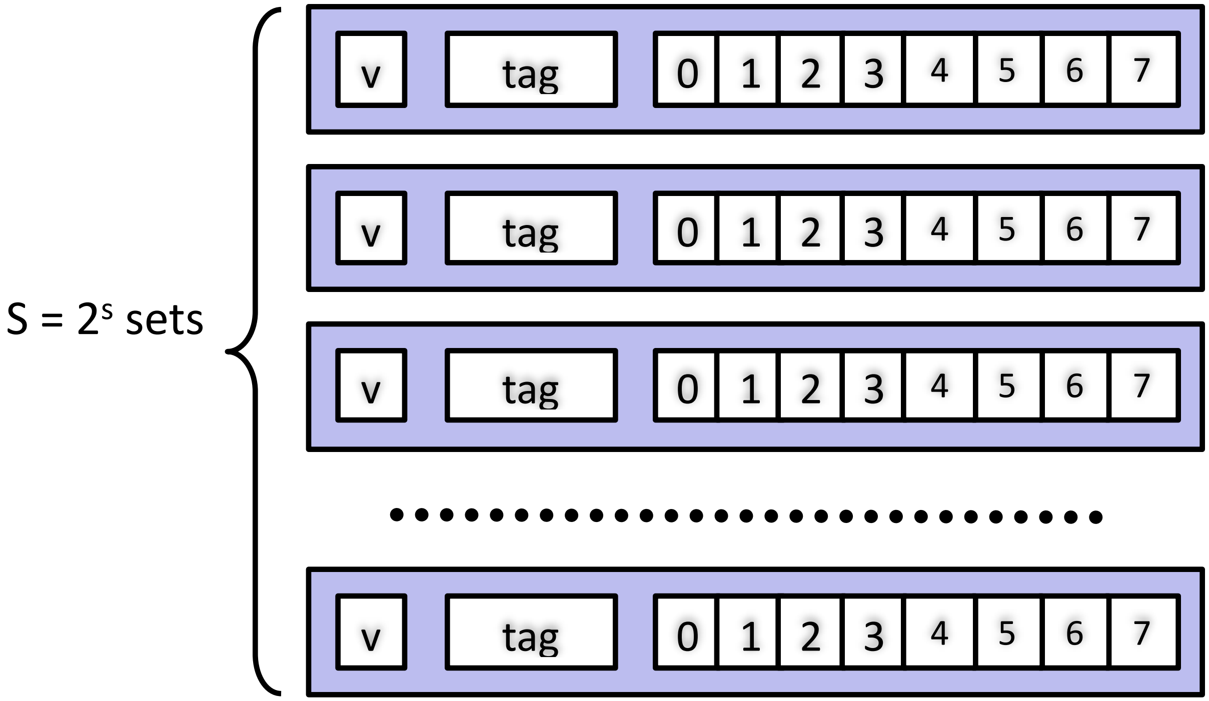
# Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

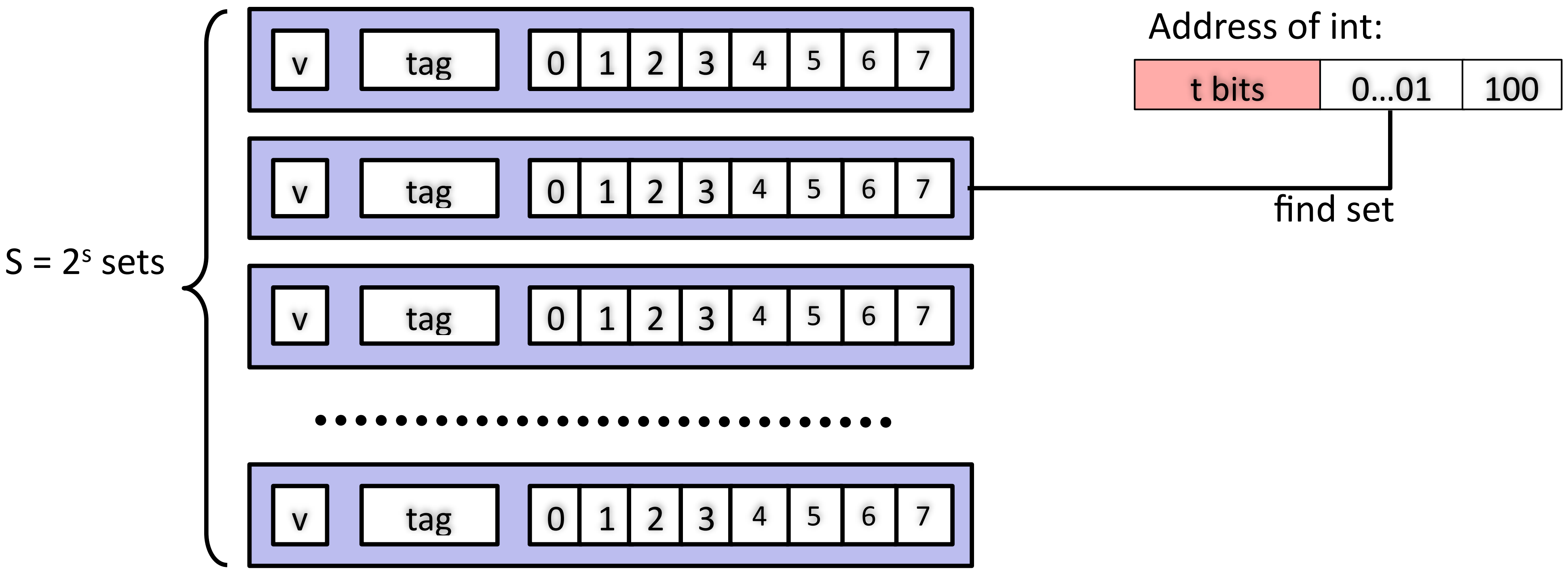


Address of int:

| t bits | 0…01 | 100 |
|---|---|---|

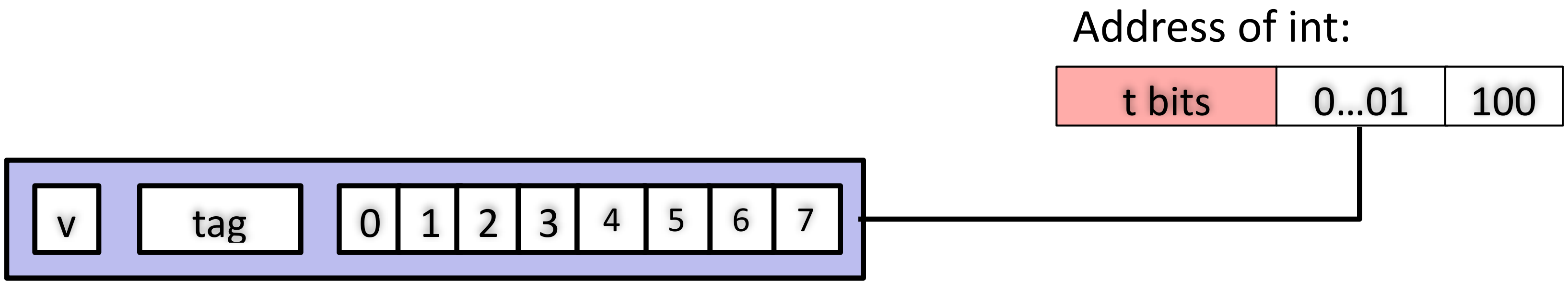
# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



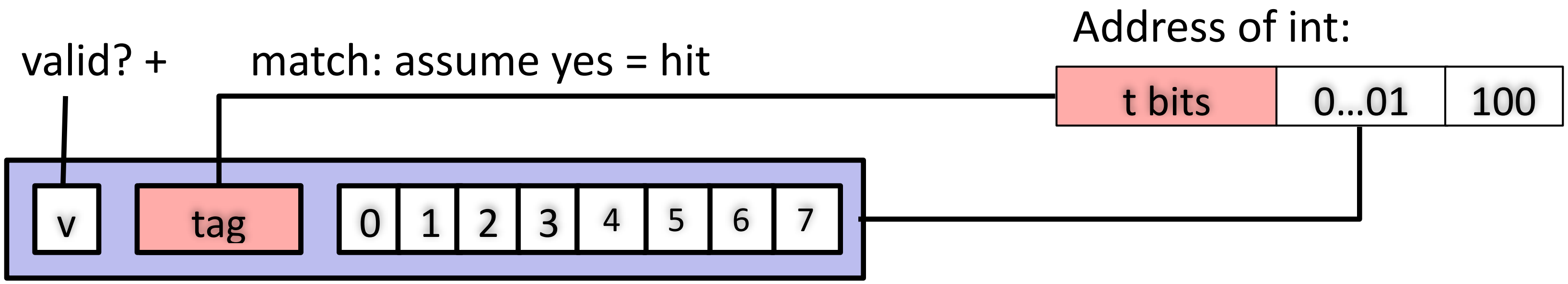
# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

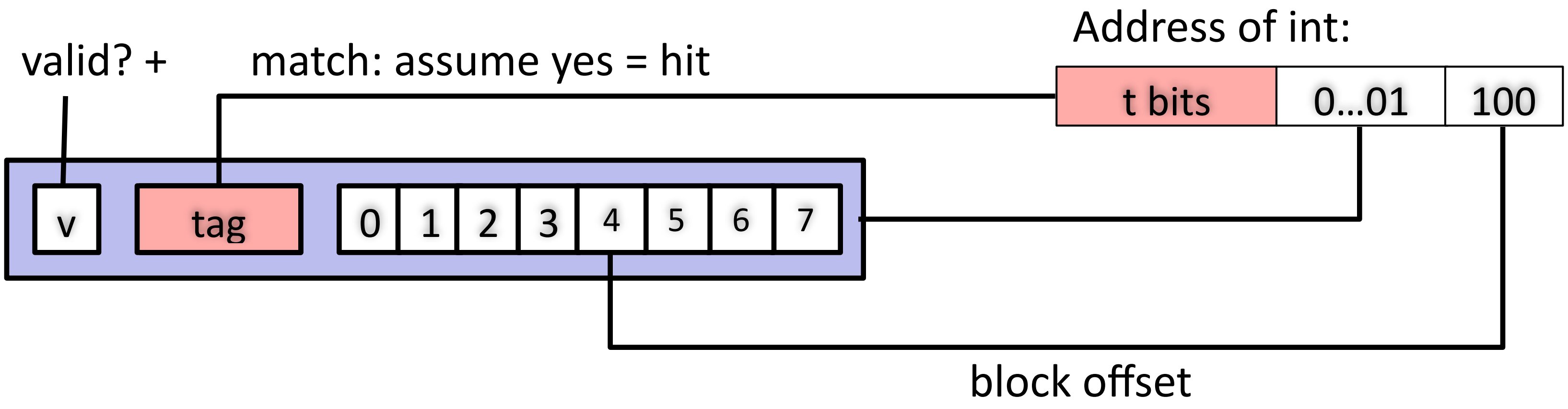
Direct mapped: One line per set
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

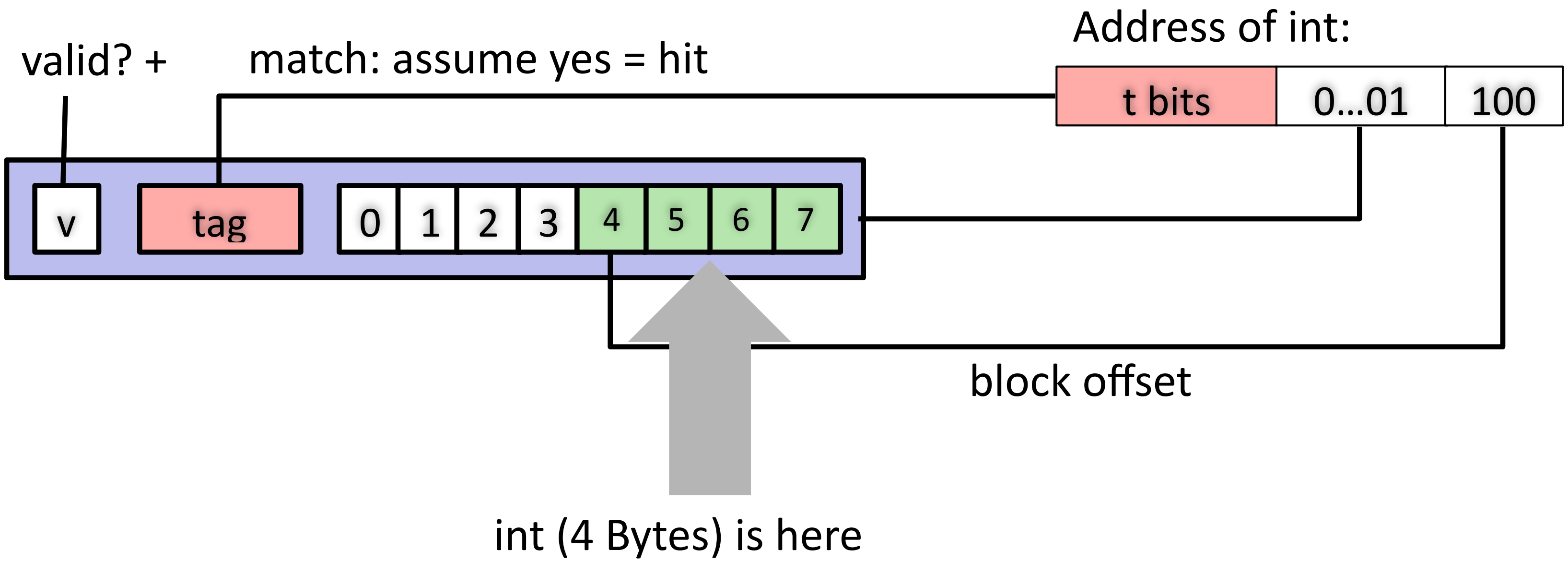
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size 8 bytes

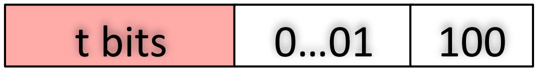


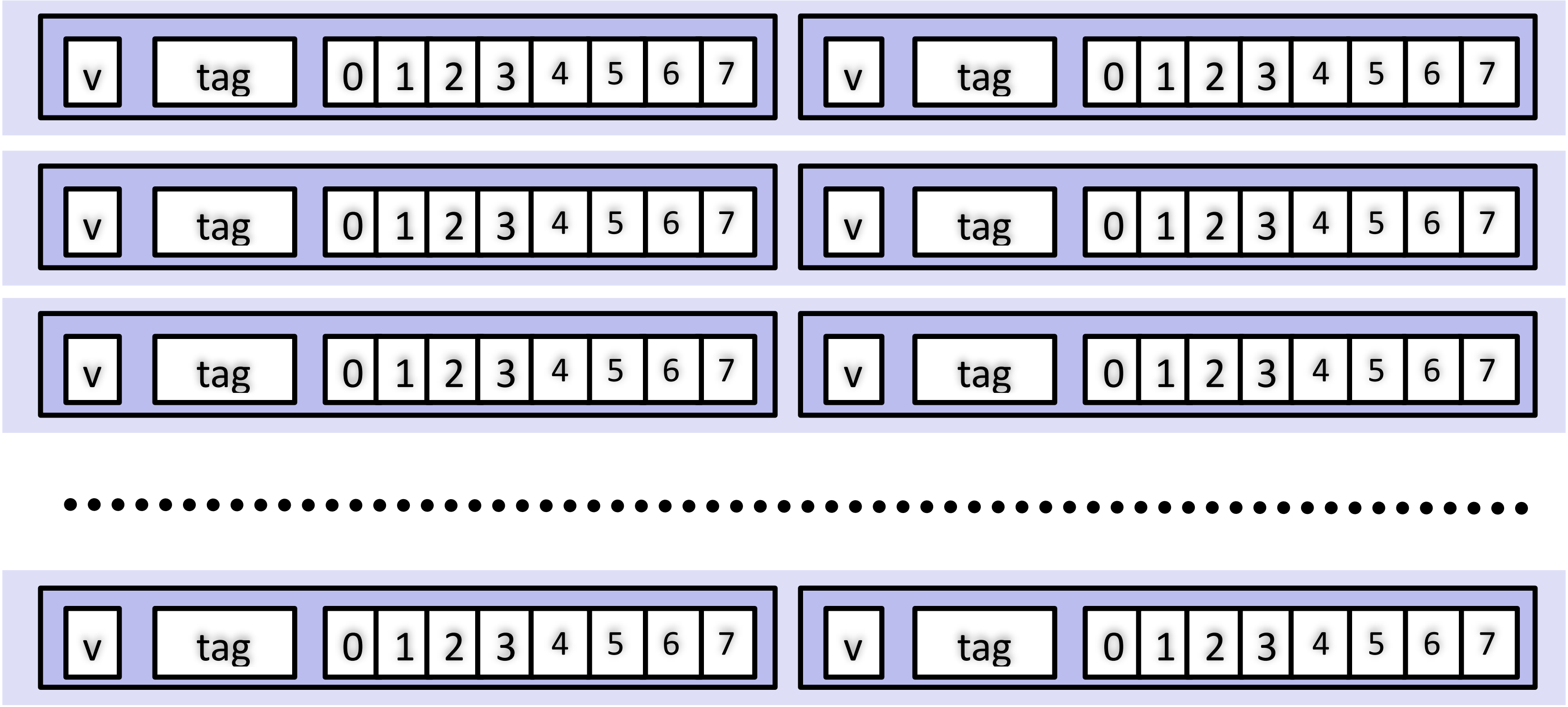
No match: old line is evicted and replaced

# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

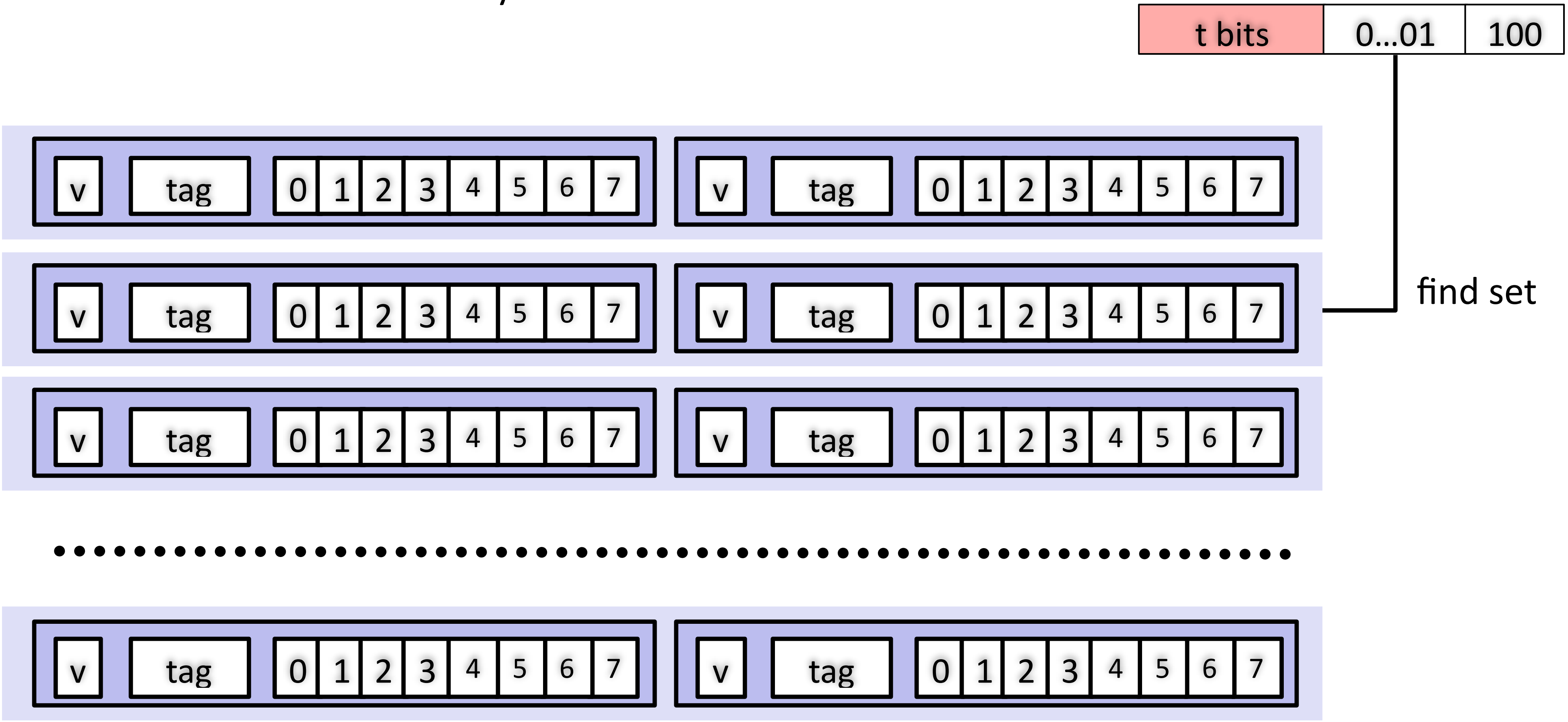




# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

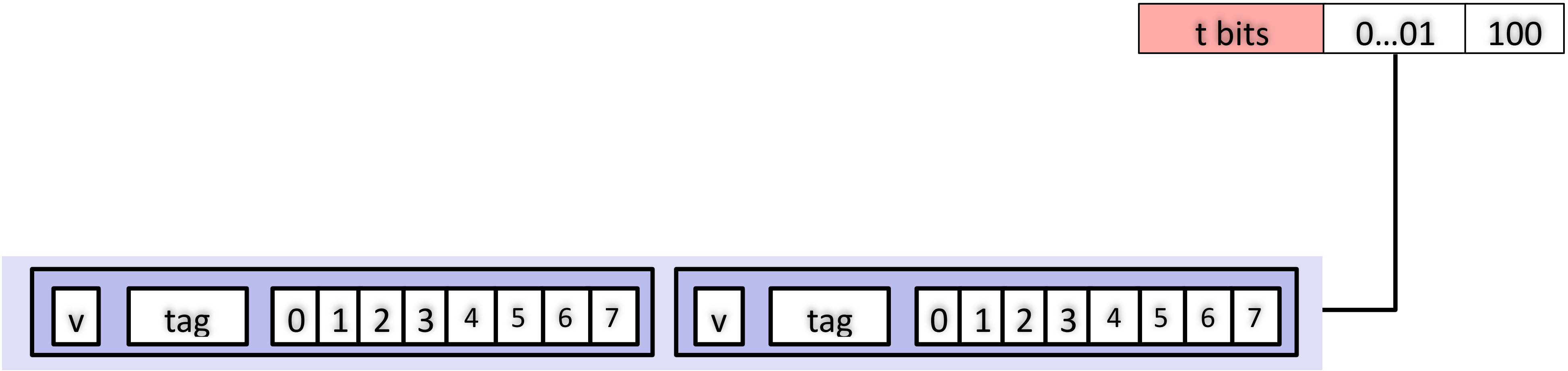
Address of short int:



# E-way Set Associative Cache (Here: E = 2)

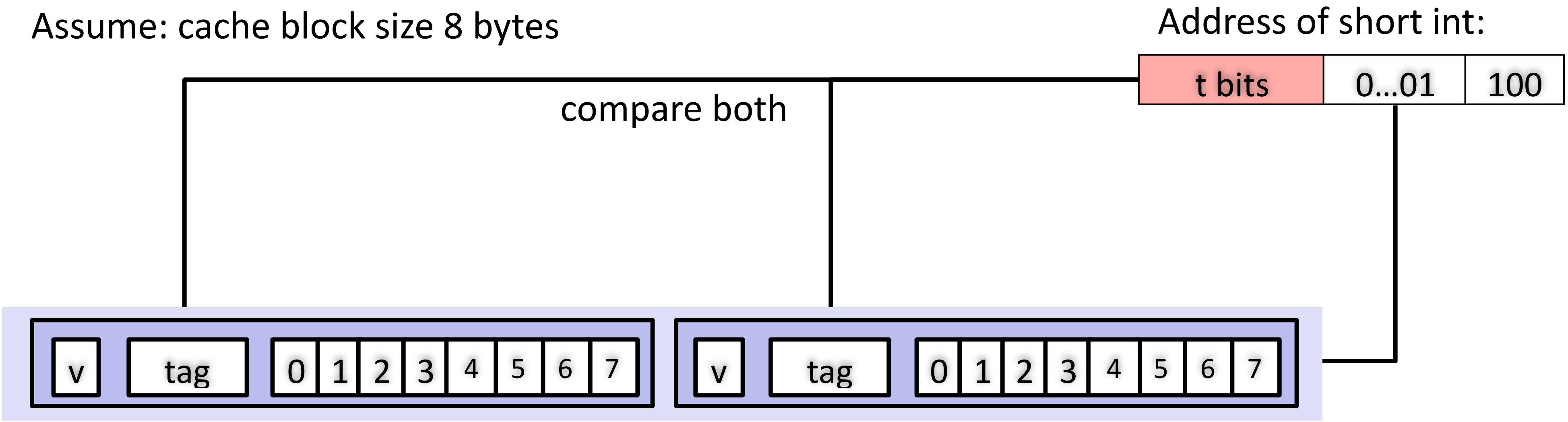
E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

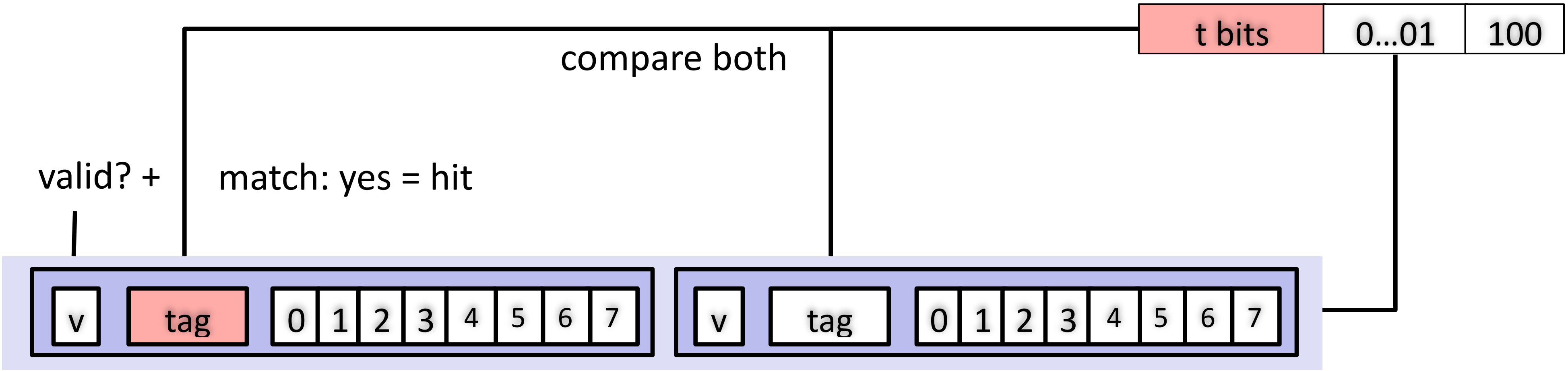


# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



compare both

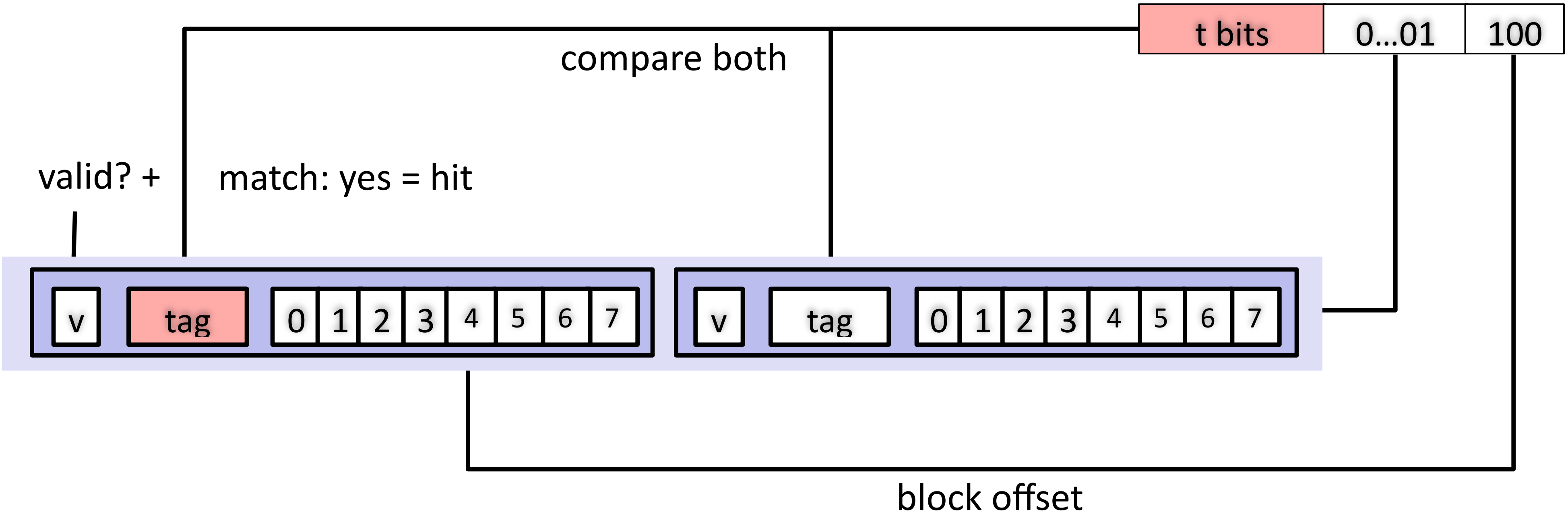
valid? + match: yes = hit

# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

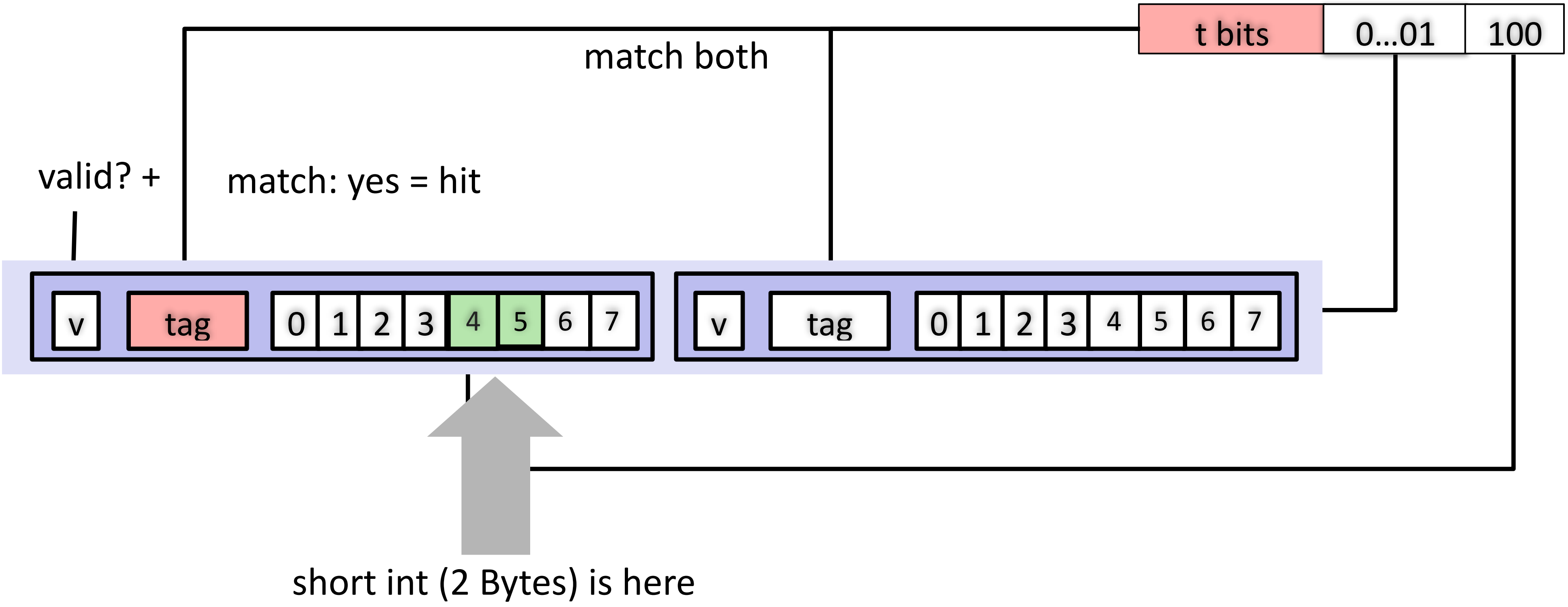


# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



match both

valid? +

match: yes = hit

short int (2 Bytes) is here

No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

# What about writes?

- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk
- **What to do one a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)

# What about writes?

- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk
- **What to do one a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)

# What about writes?

- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk
- **What to do one a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

# Software Caches are More Flexible

- **Examples**
  - File system buffer caches, web browser caches, etc.

- **Some design differences**
  - Almost always fully associative
    - so, no placement restrictions
    - index structures like hash tables are common
  - Often use complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single “block” transfers
    - may fetch or write-back in larger units, opportunistically

# CACHE OPTIMIZATIONS

# Optimizations for the Memory Hierarchy

# Optimizations for the Memory Hierarchy

- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

# Optimizations for the Memory Hierarchy

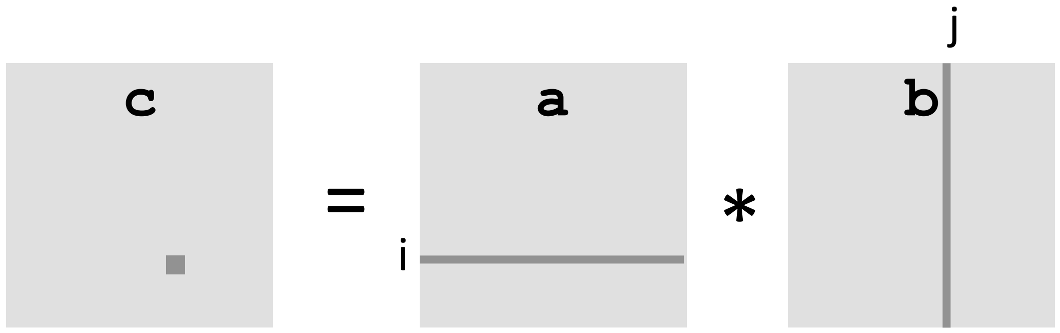
- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- **How to achieve?**
  - Proper choice of algorithm
  - Loop transformations

# Optimizations for the Memory Hierarchy

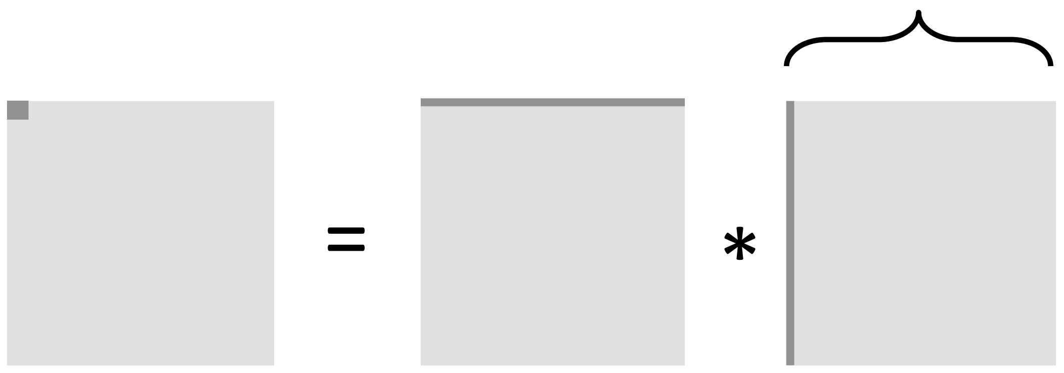
- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- **How to achieve?**
  - Proper choice of algorithm
  - Loop transformations
- **Cache versus register level optimization:**
  - In both cases locality desirable
  - Register space much smaller + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

# Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
             for (k = 0; k < n; k++)
              c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```

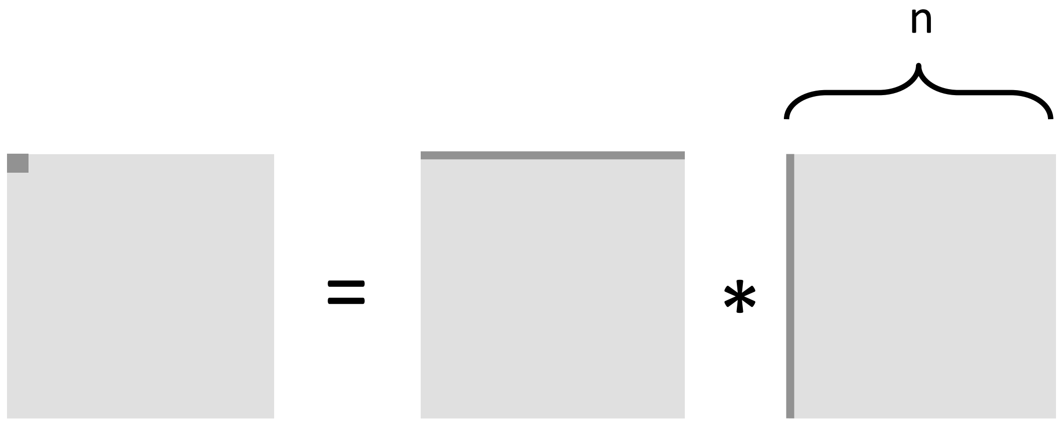


# Cache Miss Analysis



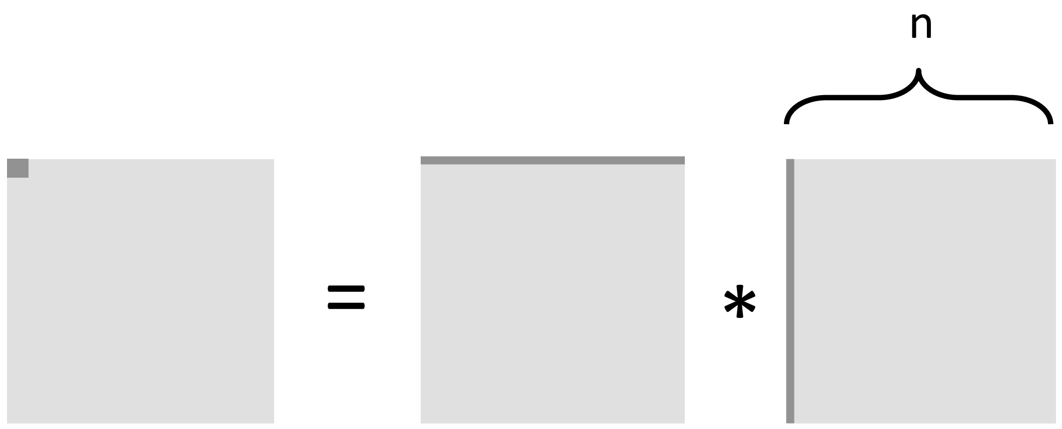
# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)



# Cache Miss Analysis

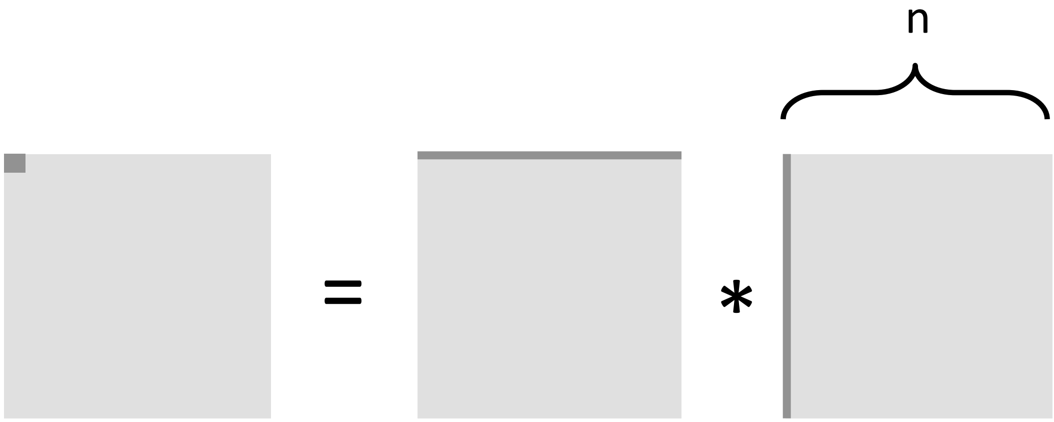
- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)



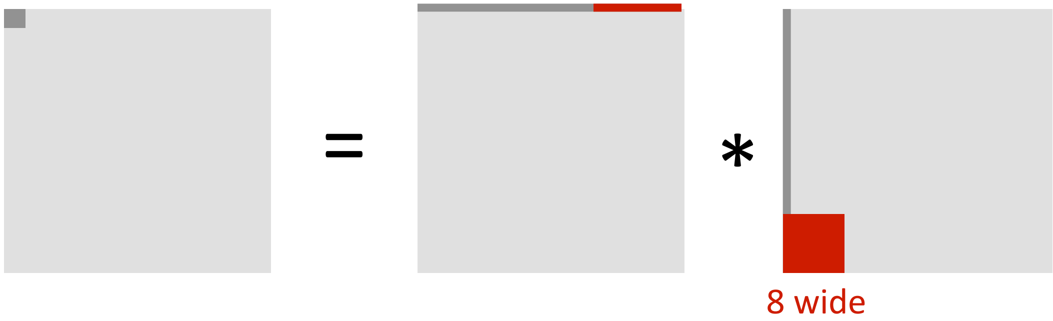
# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

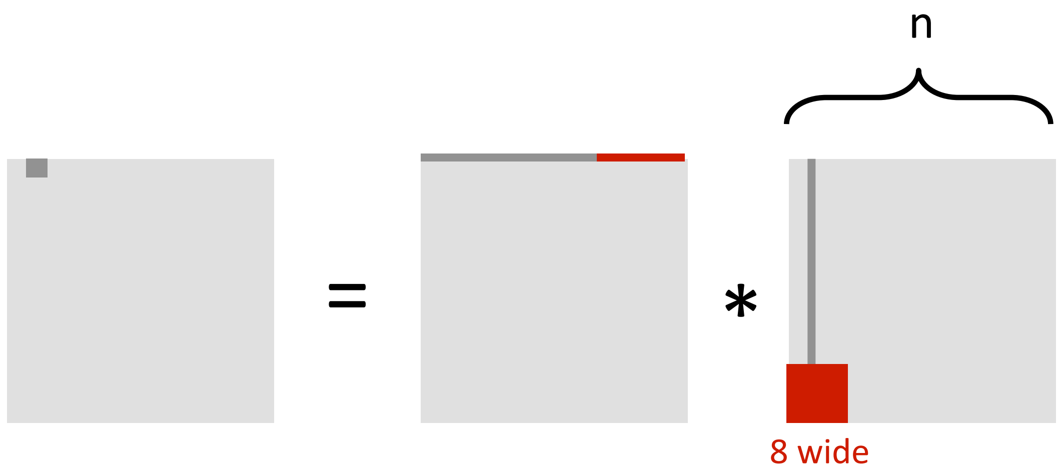
- **First iteration:**
  - n/8 + n = 9n/8 misses



  - Afterwards in cache:
(schematic)

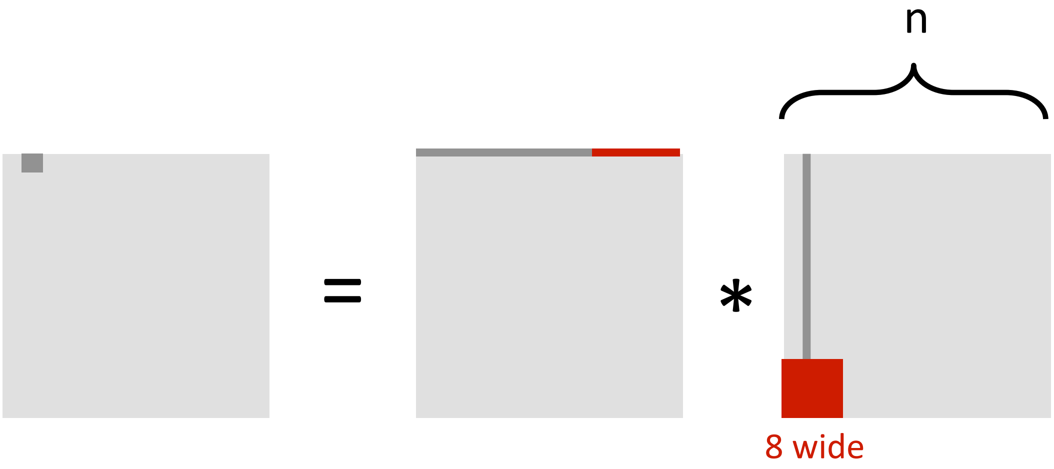


# Cache Miss Analysis



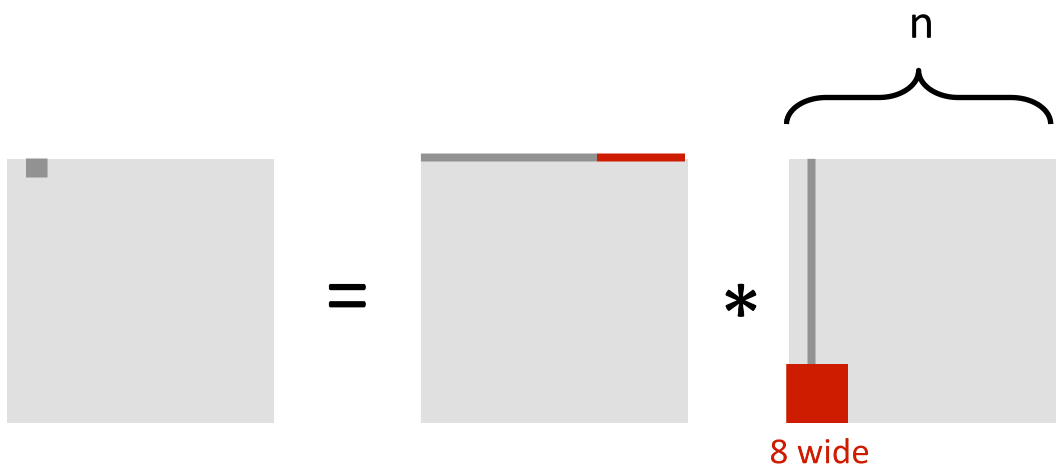
# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)



# Cache Miss Analysis

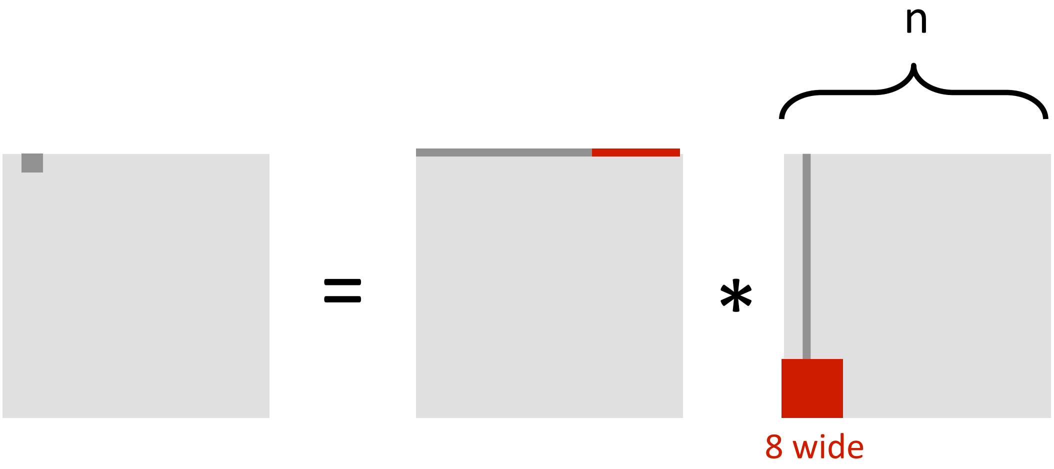
- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)



# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

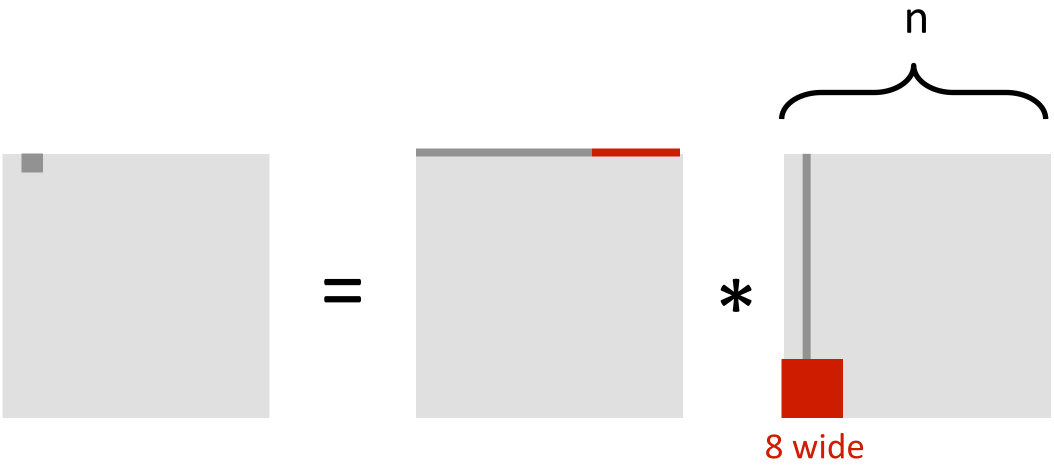
- **Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses



# Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

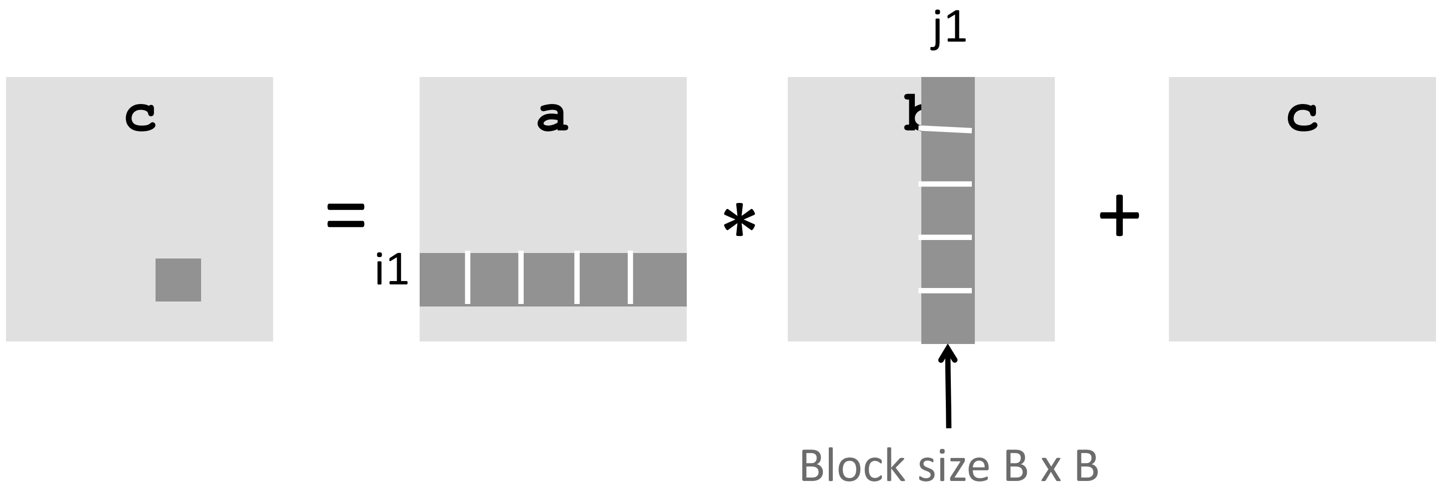
- **Second iteration:**
  - Again:  
    n/8 + n = 9n/8 misses



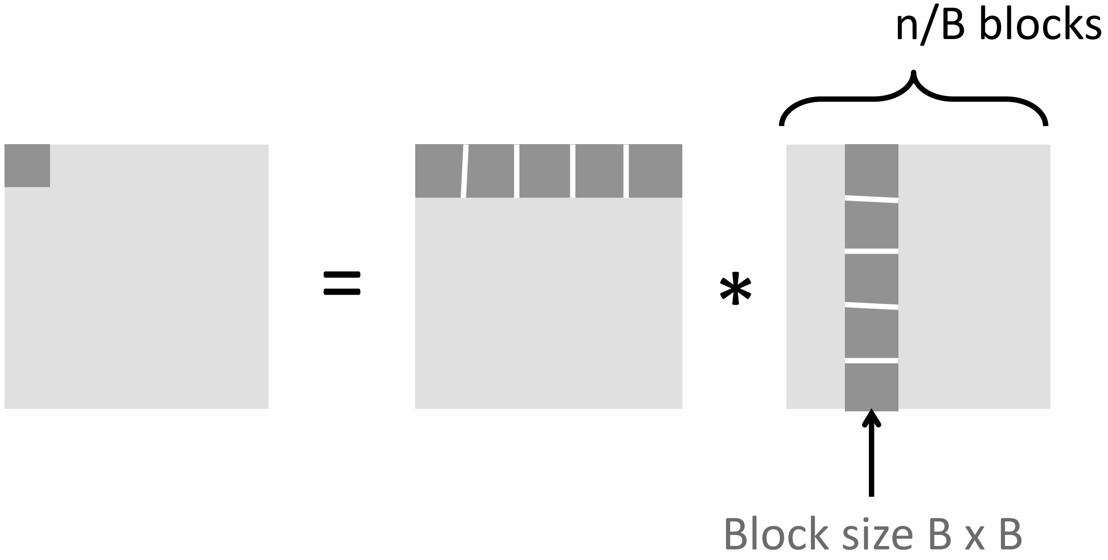
- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

# Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
         for (j = 0; j < n; j+=B)
             for (k = 0; k < n; k+=B)
                    /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i++)
                           for (j1 = j; j1 < j+B; j++)
                                   for (k1 = k; k1 < k+B; k++)
                               c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

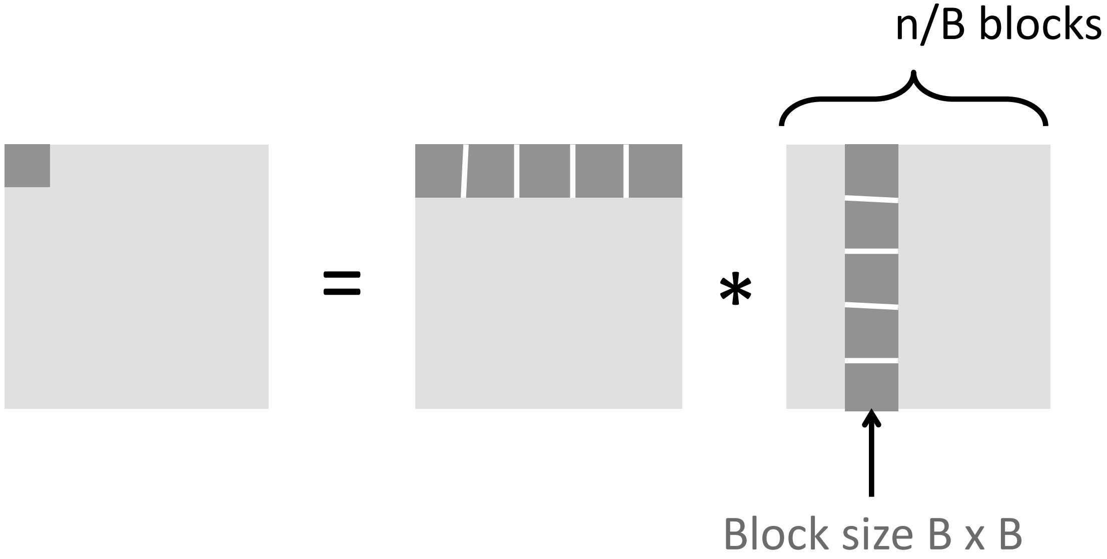


# Cache Miss Analysis



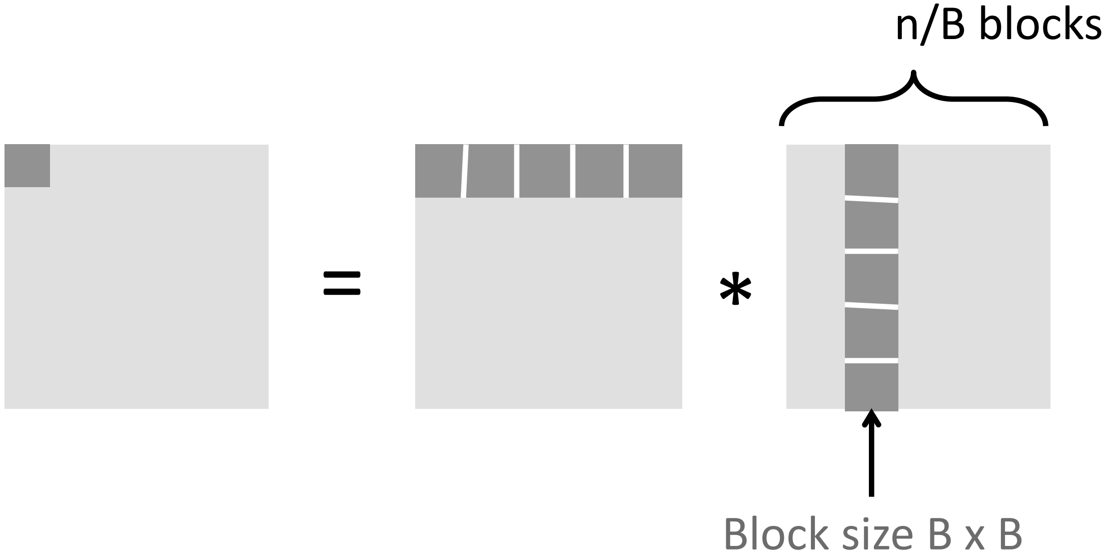
# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$



# Cache Miss Analysis

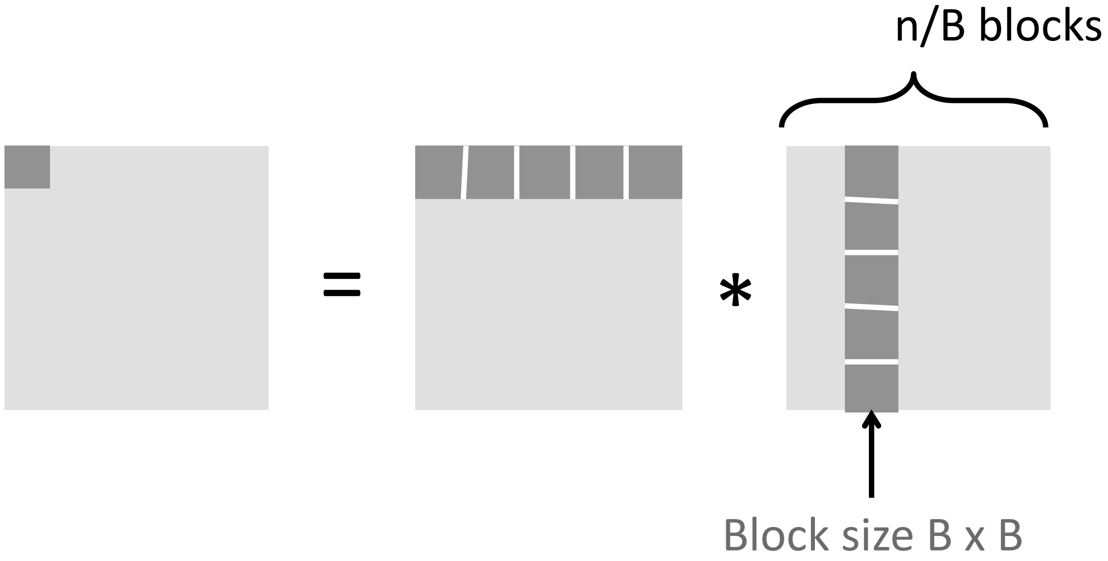
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$



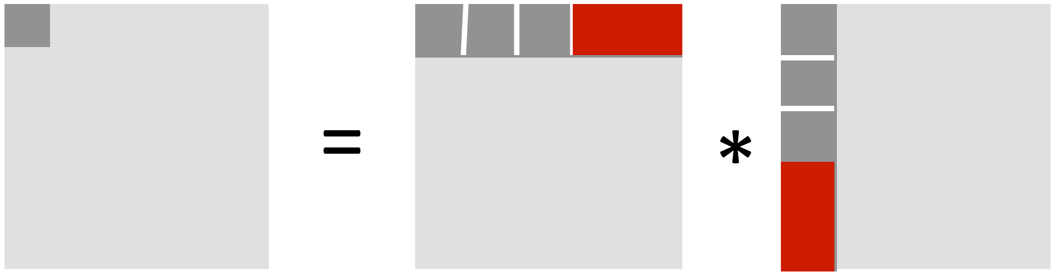
# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

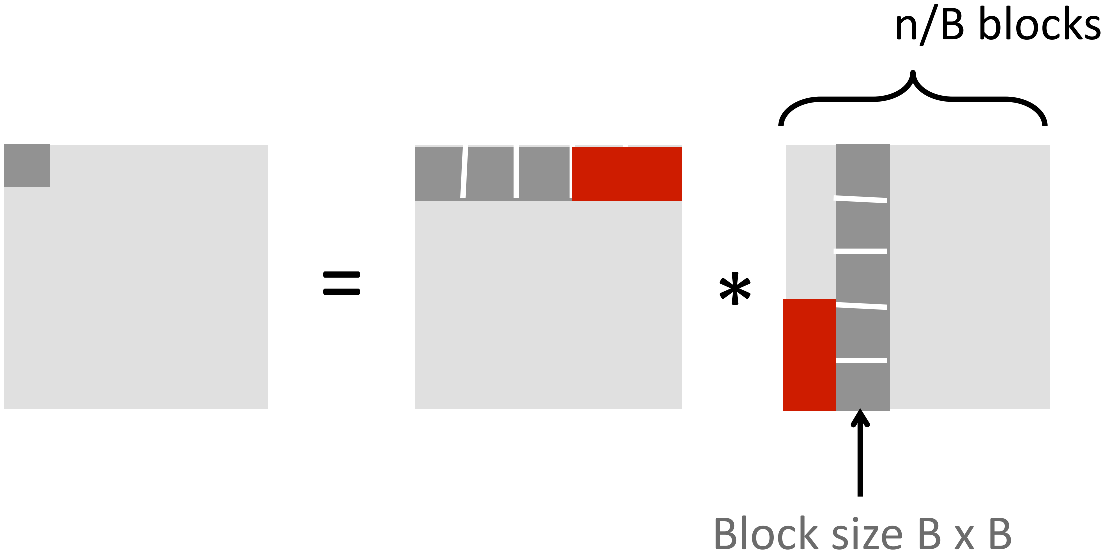
- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)



  - Afterwards in cache (schematic)

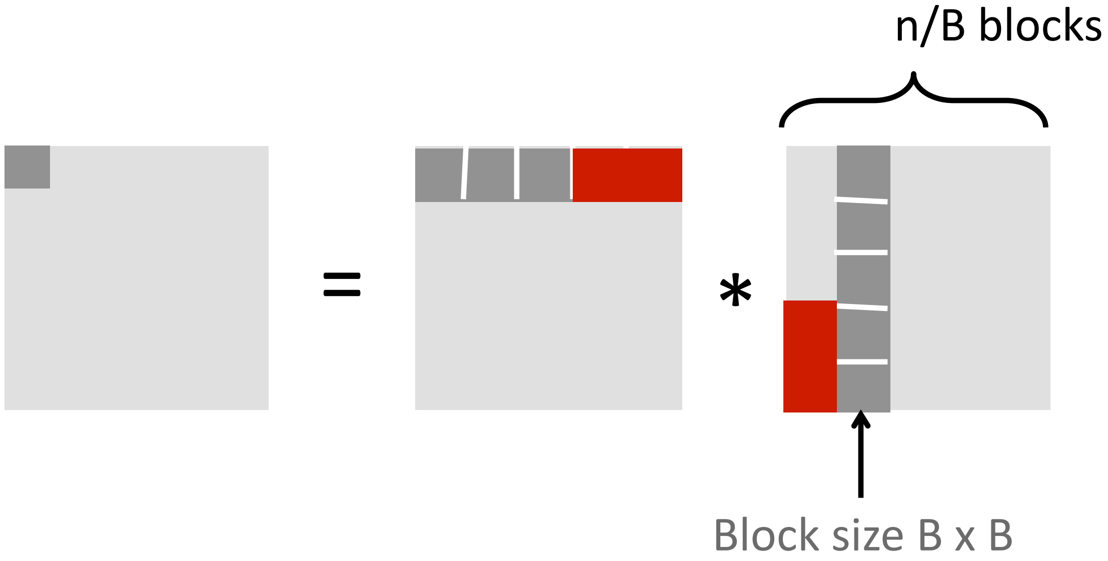


# Cache Miss Analysis



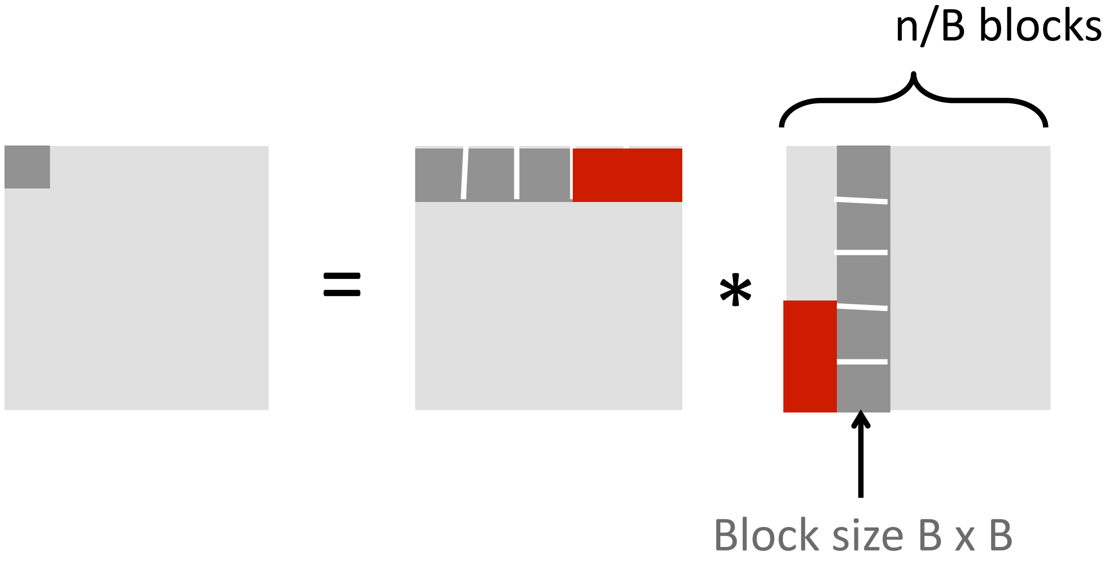
# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks      fit into cache: $3B^2 < C$



# Cache Miss Analysis

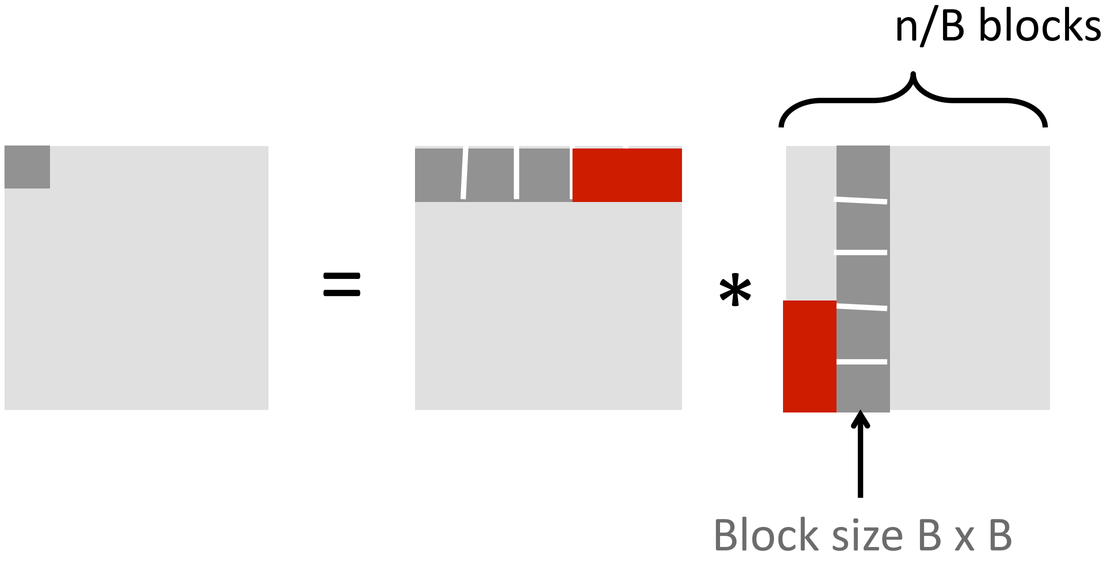
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$



# Cache Miss Analysis

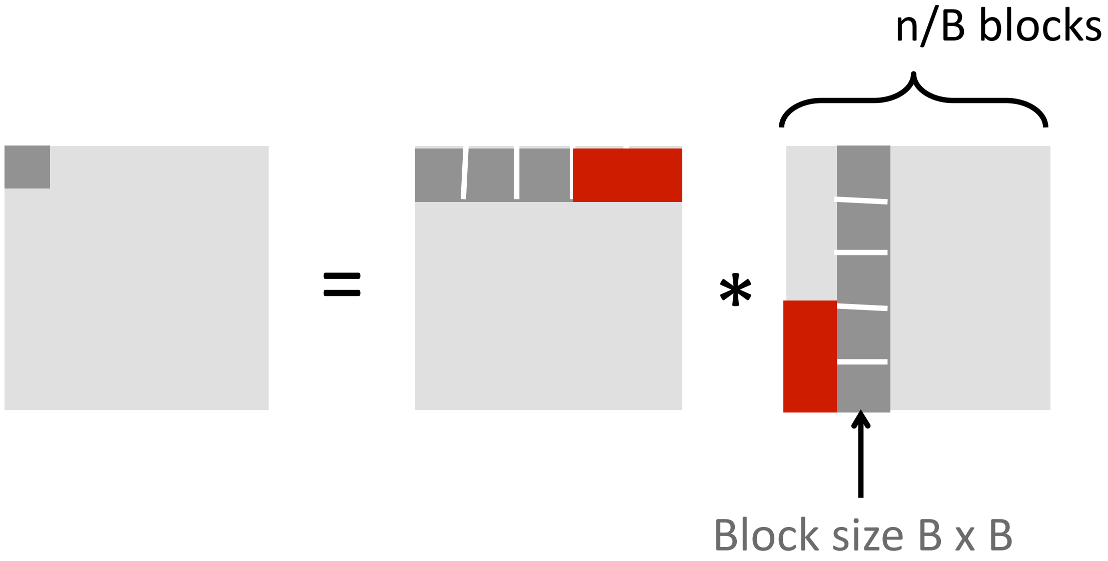
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

# Summary

- No blocking: (9/8) * $n^3$
- Blocking: 1/(4B) * $n^3$

# Summary

- **No blocking: (9/8) * $n^3$**
- **Blocking: 1/(4B) * $n^3$**

- **Suggest largest possible block size B, but limit $3B^2 < C$!**
  (can possibly be relaxed a bit, but there is a limit for B)

# Summary

- **No blocking: (9/8) * $n^3$**
- **Blocking: 1/(4B) * $n^3$**

- **Suggest largest possible block size B, but limit $3B^2 < C$!**
  (can possibly be relaxed a bit, but there is a limit for B)

# Summary

- **No blocking: (9/8) * $n^3$**
- **Blocking: 1/(4B) * $n^3$**

- **Suggest largest possible block size B, but limit $3B^2$ < C!**
  (can possibly be relaxed a bit, but there is a limit for B)

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

# Locality Example #3

```
int sum_array_3d(int a[M][N][N])
                {
        int i, j, k, sum = 0;

      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
          for (k = 0; k < N; k++)
               sum += a[k][i][j];
            return sum;
                }
```

- **How can it be fixed?**

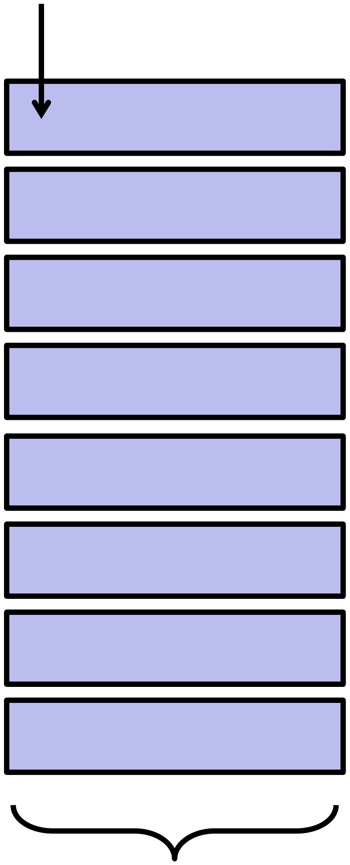
*Ignore the variables sum, i, j*

# Example

```
int sum_array_rows(double a[16][16])
                  {
                int i, j;
             double sum = 0;

        for (i = 0; i < 16; i++)
          for (j = 0; j < 16; j++)
                 sum += a[i][j];
               return sum;
                  }
```

```
int sum_array_cols(double a[16][16])
                  {
                int i, j;
             double sum = 0;

        for (j = 0; i < 16; i++)
          for (i = 0; j < 16; j++)
                 sum += a[i][j];
               return sum;
                  }
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

blackboard

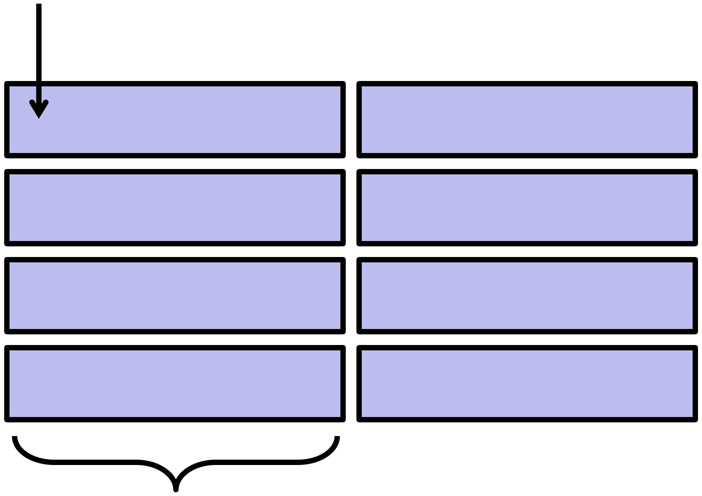
# Example

*Ignore the variables sum, i, j*

```
int sum_array_rows(double a[16][16])
                  {
                 int i, j;
             double sum = 0;

        for (i = 0; i < 16; i++)
          for (j = 0; j < 16; j++)
                 sum += a[i][j];
               return sum;
                  }
```

```
int sum_array_rows(double a[16][16])
                  {
                 int i, j;
             double sum = 0;

        for (j = 0; i < 16; i++)
          for (i = 0; j < 16; j++)
                 sum += a[i][j];
               return sum;
                  }
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

blackboard