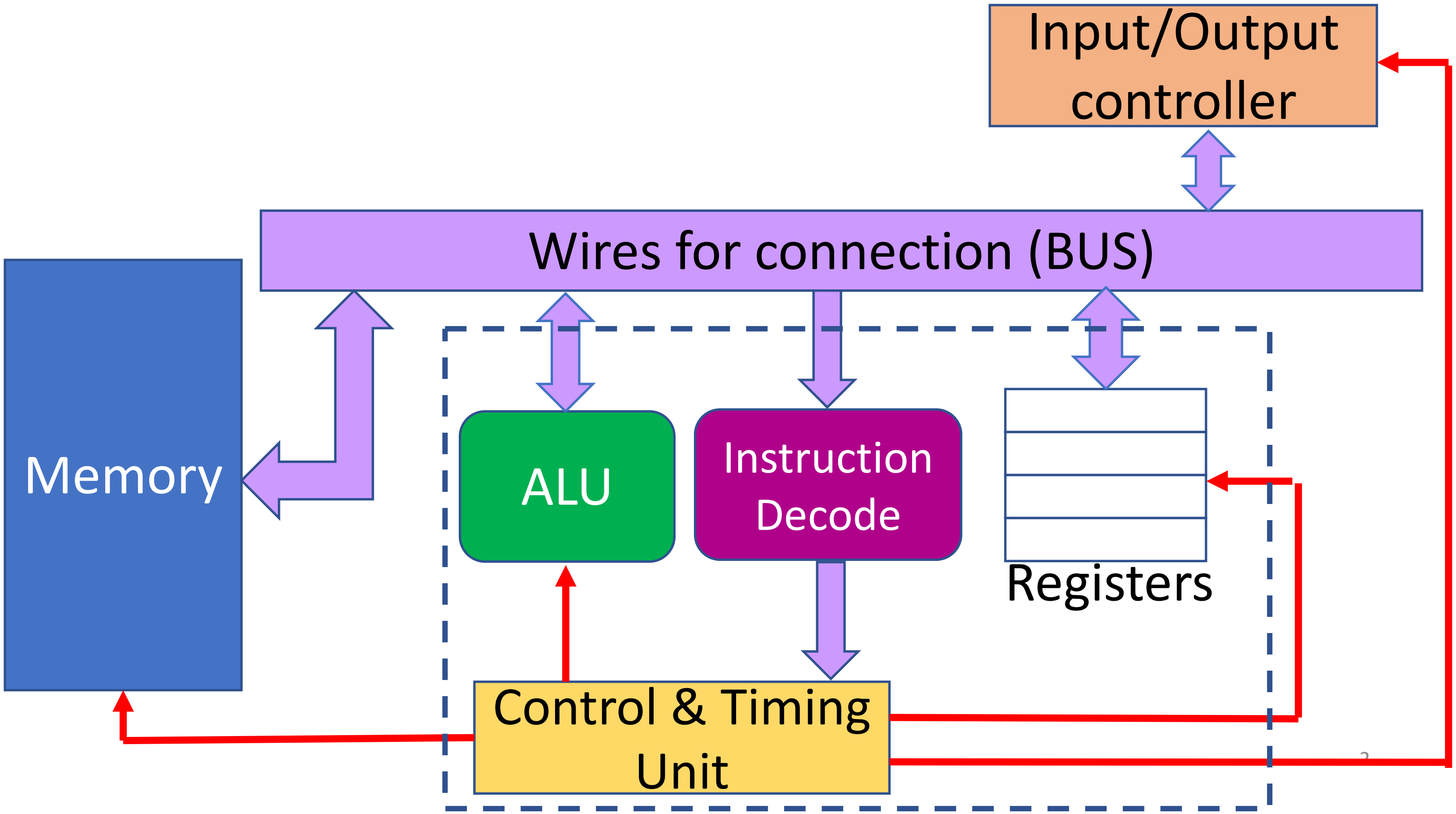
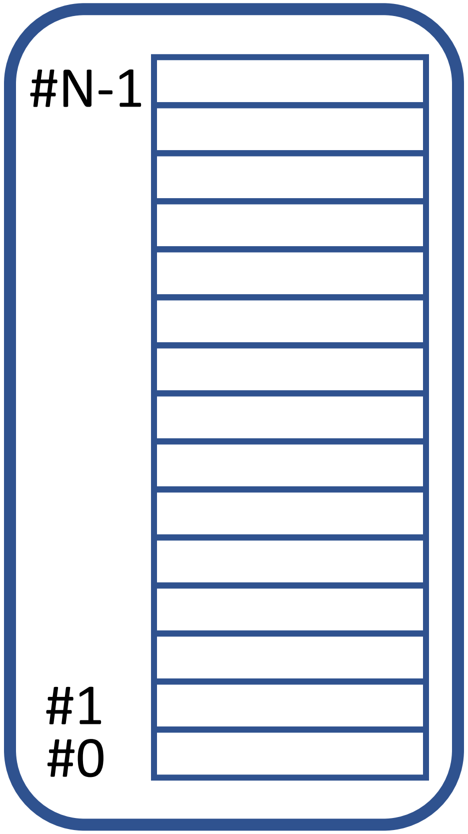
# Computer Architecture: Memory Hierarchy

Mohammed Bahja
School of Computer Science
University of Birmingham

# Recap: Computer organization



# Memory: Programmer's perspective



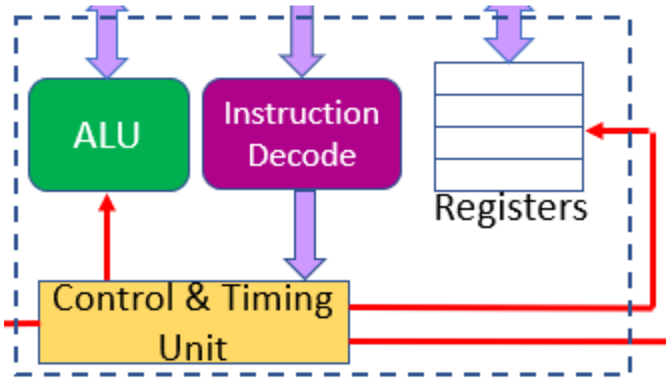

Memory

This shows a simplified functional view of Memory.

Our computers have more complex memory system!

- Memory consists of small 'locations'
- Each location can store a small information

# Different types of memory




Registers inside a Processor


DRAM (we call it ‘RAM’)


Solid State Disk


Hard Disk

# Different types of memory

| Memory Tech. | Capacity | Data speed (time) | Cost/GB |
|---|---|---|---|
| Registers | ~1000s bits | 10 ps | ££££££££ |
| SRAM | ~10 KB-10 MB | 1-10 ns | ~£1000 |
| DRAM | ~10 GB | 100 ns | ~£10 |
| SSD | ~100 GB | 100 us | ~£1 |
| Hard Disk | ~1 TB | 10 ms | ~£0.10 |

Different memory technologies have different **tradeoffs**.

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

Given these **tradeoffs:**

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

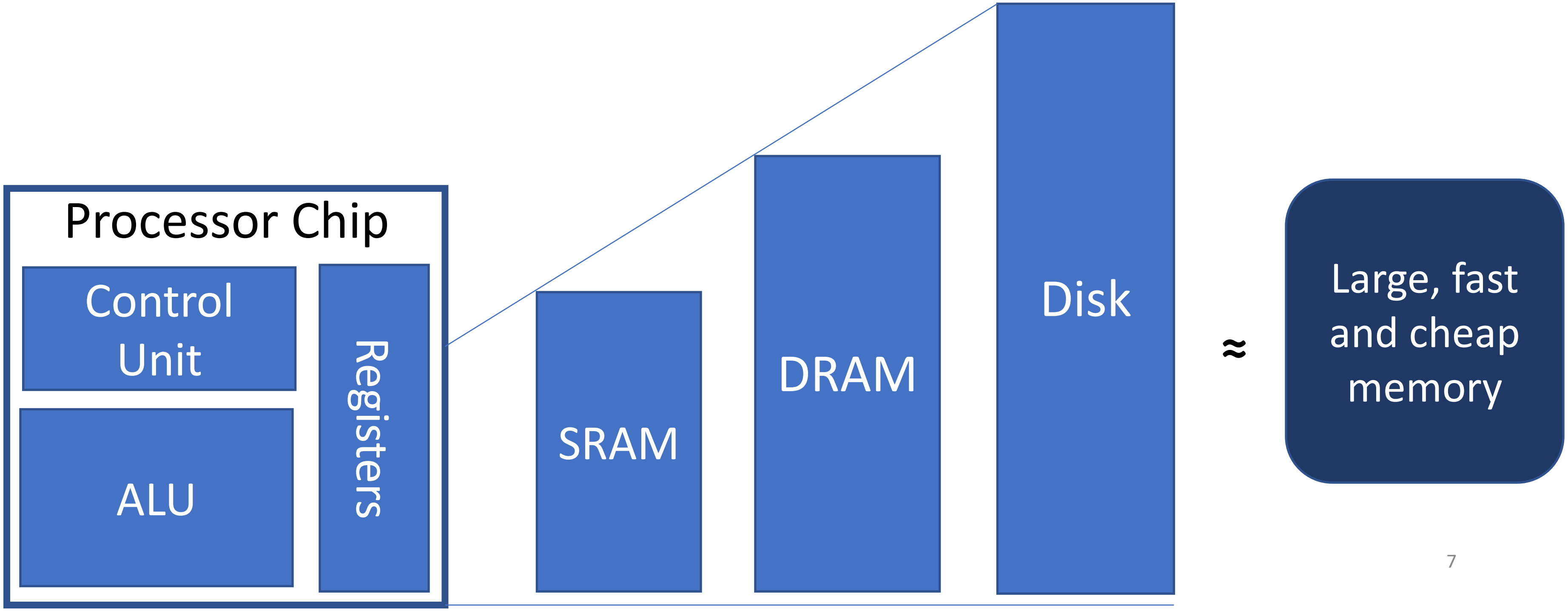
Can we have a computer which has '*large, fast and cheap*' memory?

Given these **tradeoffs:**

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

Can we have a computer which has '*large, fast and cheap*' memory?

**Idea:** use hierarchy of memory elements to emulate a '*large, fast and cheap*' memory.



The next cartoon shows how to think of memory hierarchy.

I have study leave before exam.
At home, I want to prepare computer architecture.

I have study leave before exam.
At home, I want to prepare computer architecture.

Let’s collect the architecture book by Patterson and Hennessy from our Library.

Library (large storage of books)

Computer architecture books

There are several books on computer architecture. So, why not collect a few more of them?
May be later, I find some of them useful!

*home sweet home*

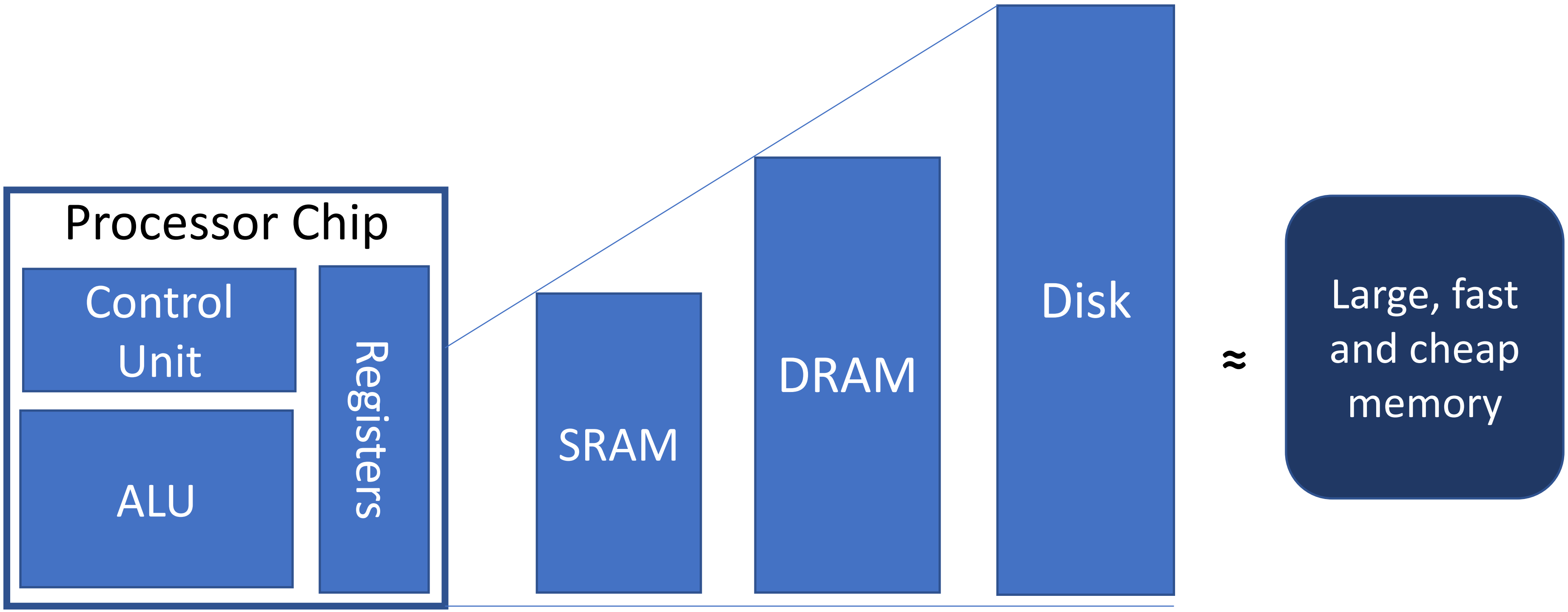
Luckily, I had brought these other books on Comp Architecture. Otherwise I would have required to visit the library again!

Luckily, I had brought these other books on Comp Architecture. Otherwise I would have required to visit the library again!

*home sweet home*

**Idea:** Keep the book that you wanted to study as well as a few extra books that might be useful, on your study desk.

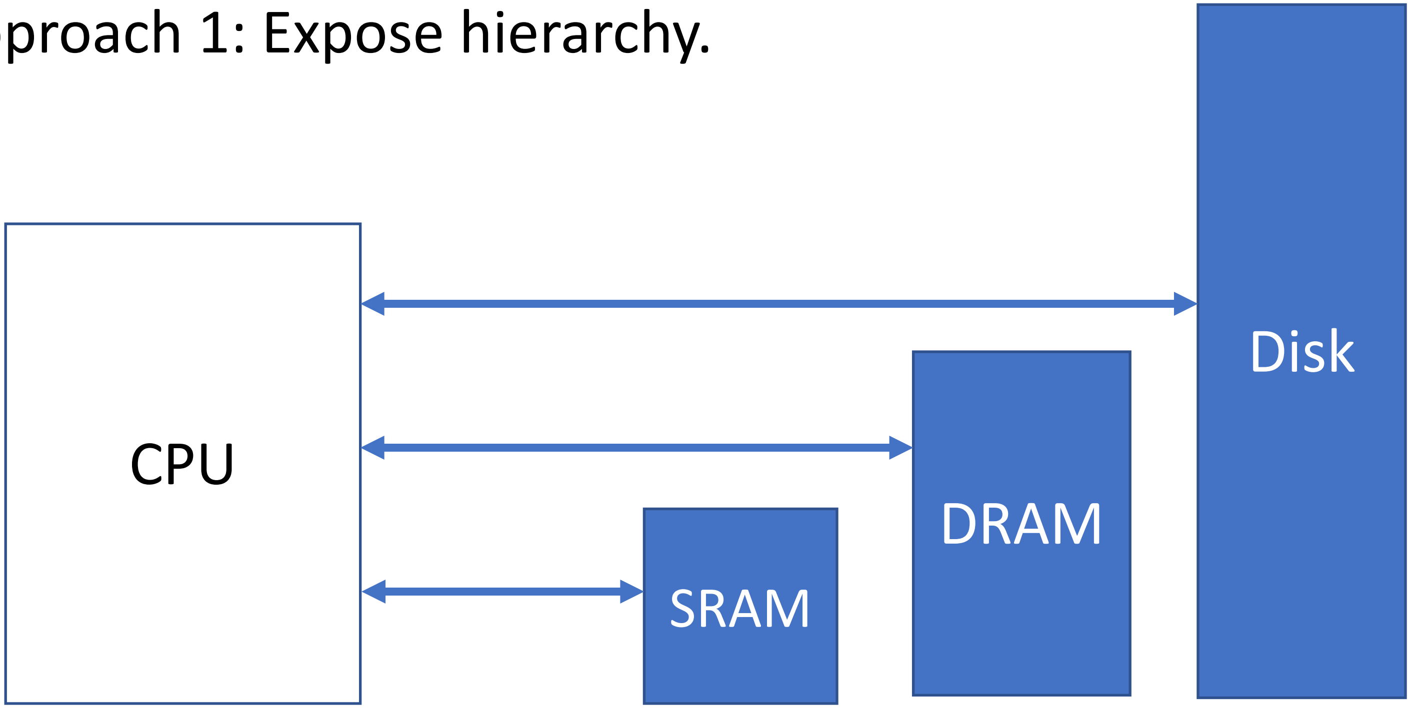
The same idea can be applied to a computer!




Idea: Keep the piece of data that the processor requires now, as well as some extra data that might useful next, close to the processor in the fast memory.

# How to interface memory hierarchy?

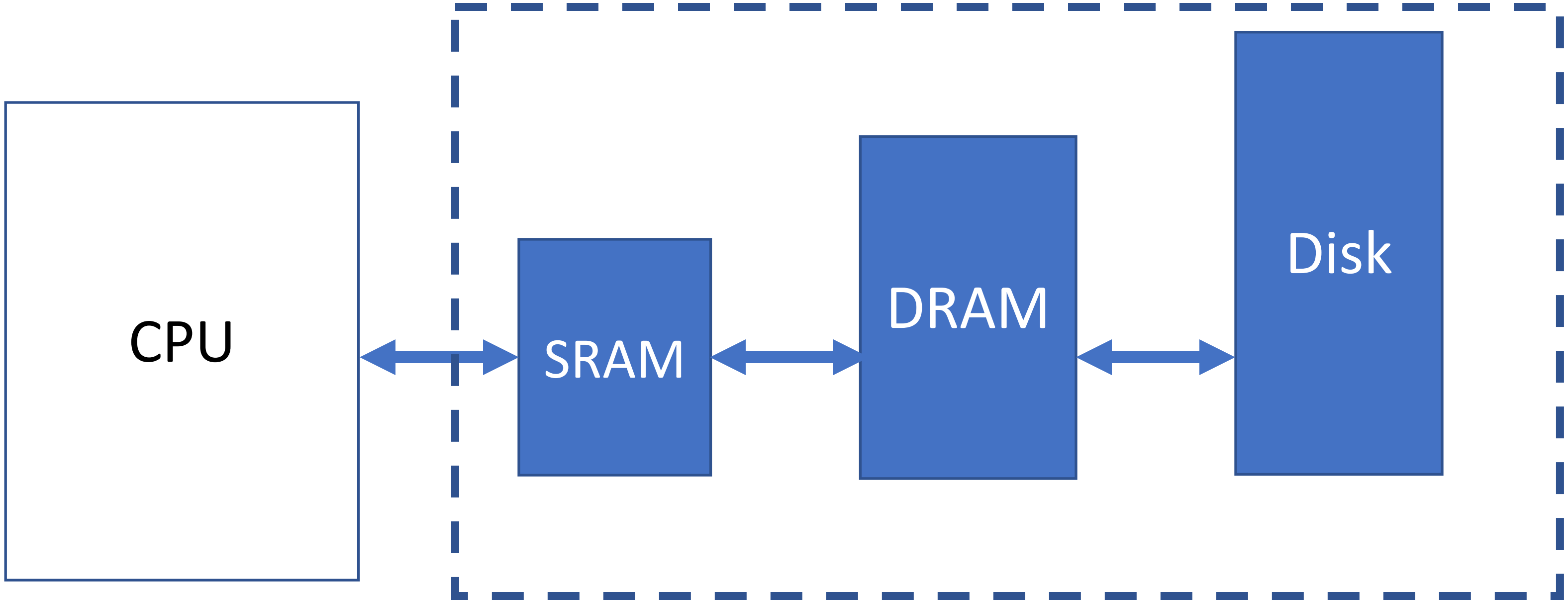
Approach 1: Expose hierarchy.




- CPU can access any level in the memory hierarchy directly
- Programmer should know the ‘details’ of the hierarchy and should write code 'cleverly’.

# How to interface memory hierarchy?

Approach 2: Hide hierarchy.




- Programmer's perspective: a single memory with single address space.
- Hardware takes the responsibility of storing data in fast or slow memory, depending on usage patterns.
- ‘Cleaver programmer’ writes code in such a way that the CPU gets its data from fast memory with high probability. [will learn this]

## How does a computer decide which data to keep in fast memory and which data to keep in slow memory?

Answer: computer uses locality of reference

# The locality of reference

- **Locality of reference**, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.

- Two kinds of locality: temporal and spatial locality.

- Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration.

- Spatial locality refers to the use of data elements within relatively close storage locations.

# The locality of reference: example

Example of computing the sum of an array

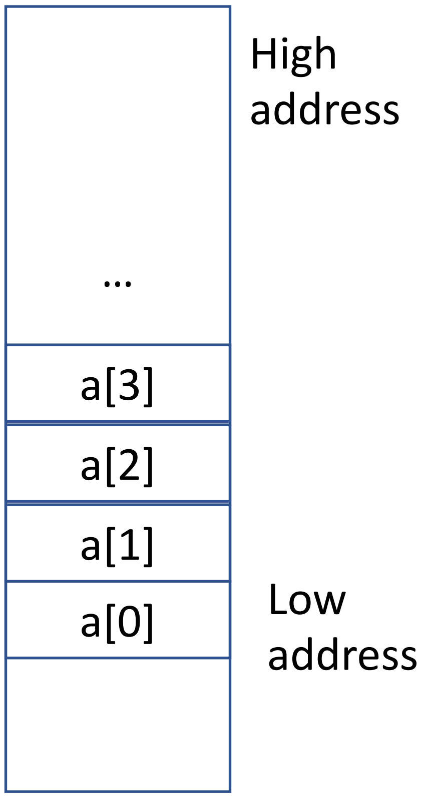
```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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

Do you see any locality in the program?

# The locality of reference: example

Example of computing the sum of an array

```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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




Locality of reference in the above program:

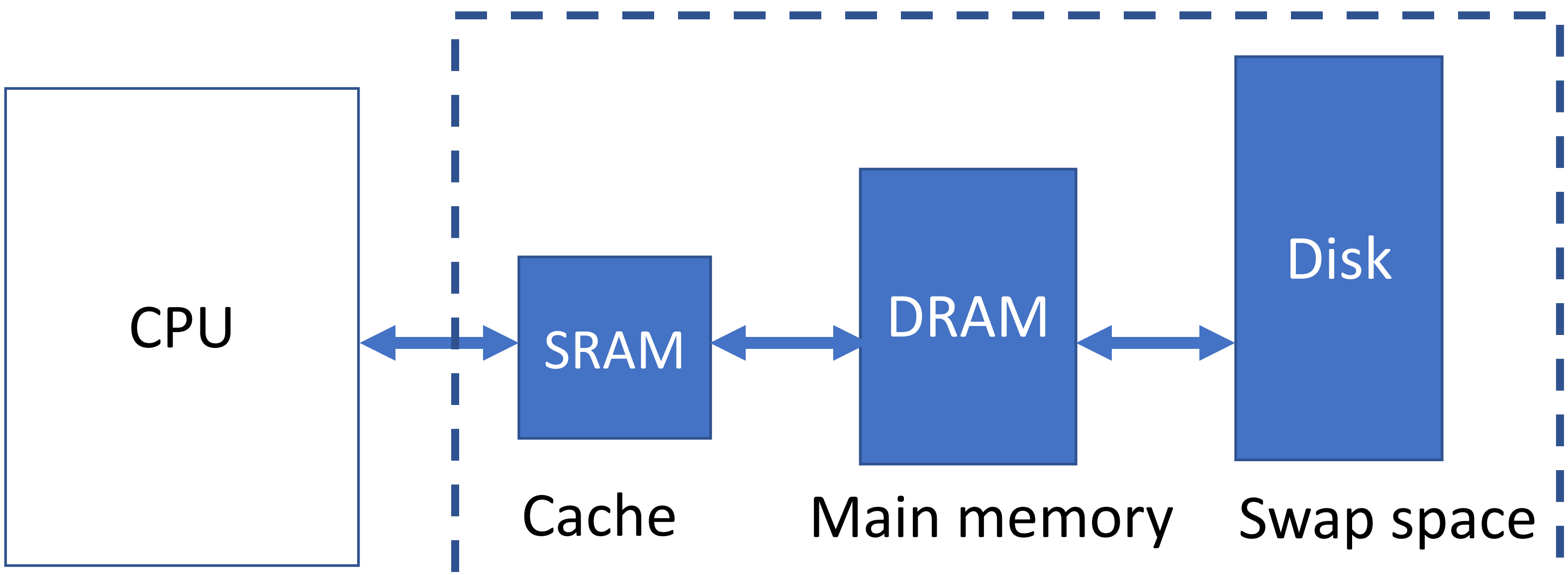
- The object 'sum' satisfies temporal locality.
  It is used again and again.

- Array elements satisfy spatial locality.
  → If `a[i]` is used, then use of `a[i+1]`, `a[i-1]` etc. is highly probable.

# Computer tries to increase locality of reference

Computer tries to:

- Keep the most-frequently used data in a fast (but small) SRAM. This SRAM is called 'cache' as it transparently retains (caches) data from recently accessed memory locations in DRAM.
- Refer to large (but slower) DRAM for data which is not present in cache. This DRAM is called 'main memory'
- Swap space is used only when data cannot be fit in main memory.

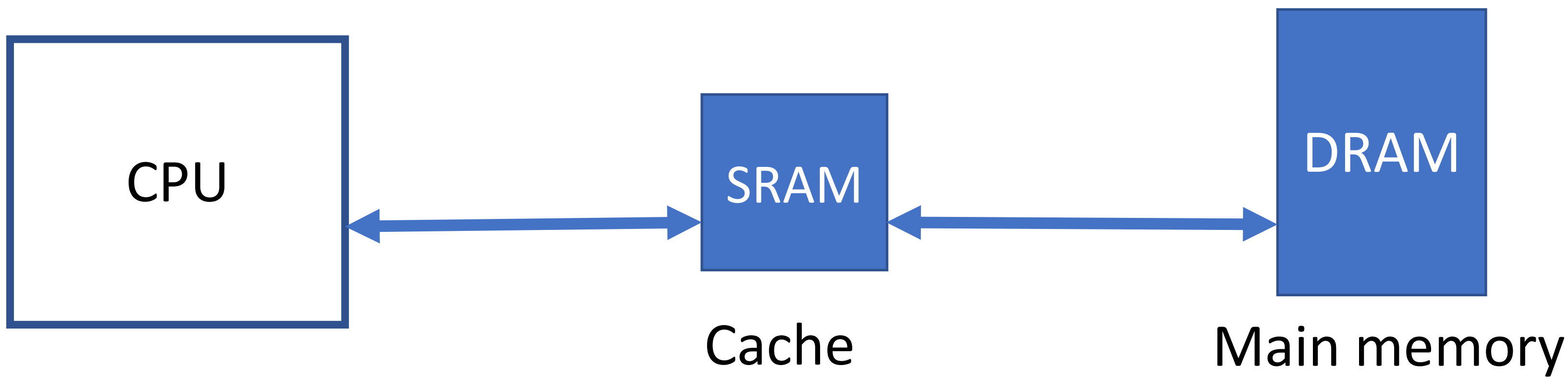


# A typical memory hierarchy

| | | Access time | Capacity |
|---|---|---|---|
| Attached to ALU | Registers | 1 cycle | 1000s of bits |
| Different levels of caches are present inside the processor chip. | Level 1 Cache | 4 cycles | 32 KB |
| | Level 2 Cache | 10 cycles | 256 KB |
| | Level 3 Cache | 40 cycles | 10 MB |
| Present outside the processor chip. | Main Memory | 200 cycles | 8-16 GB |
| | SSD | 10-100 us | 256 GB |
| Slow mechanical device | Hard Disk | 10 ms | 1 TB |

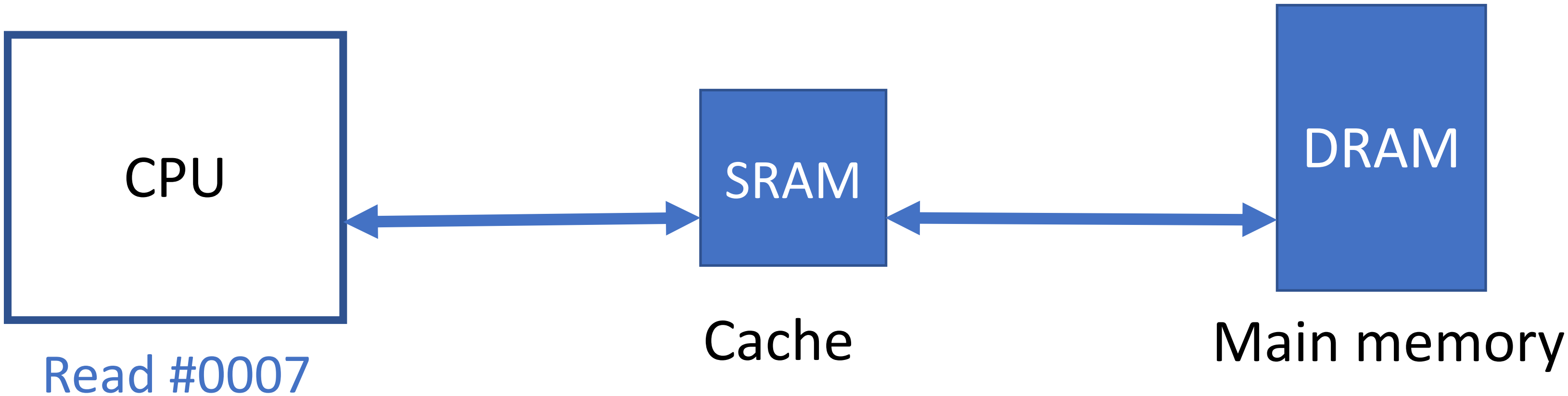
# Cache access

Example: (Simplified) Computer with only Level 1 Cache and Main memory



# Cache access

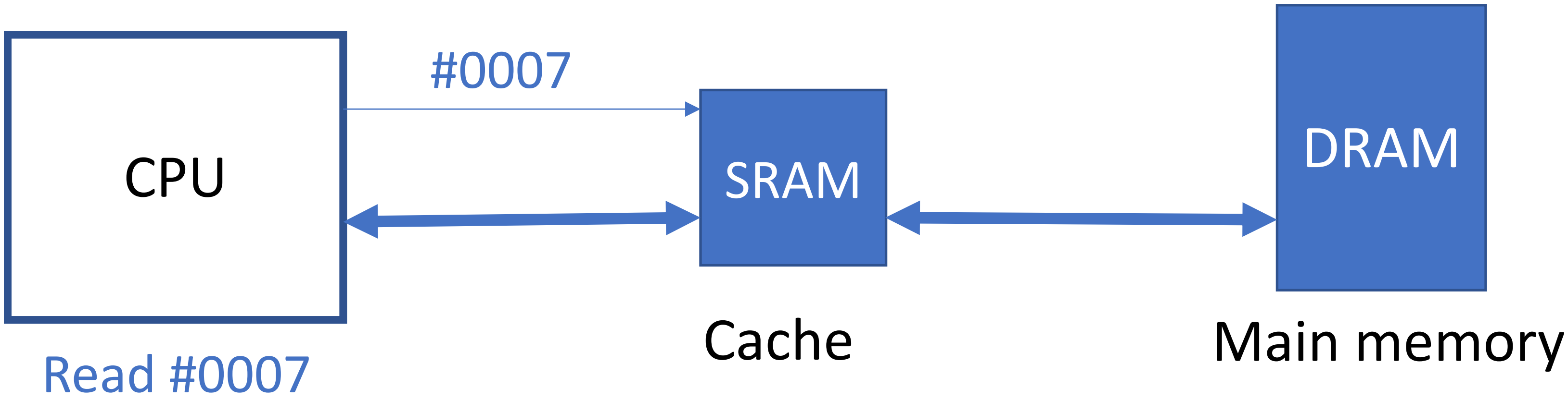
Example: (Simplified) Computer with only Level 1 Cache and Main memory




1. Suppose CPU wants to read from address #0007

# Cache access

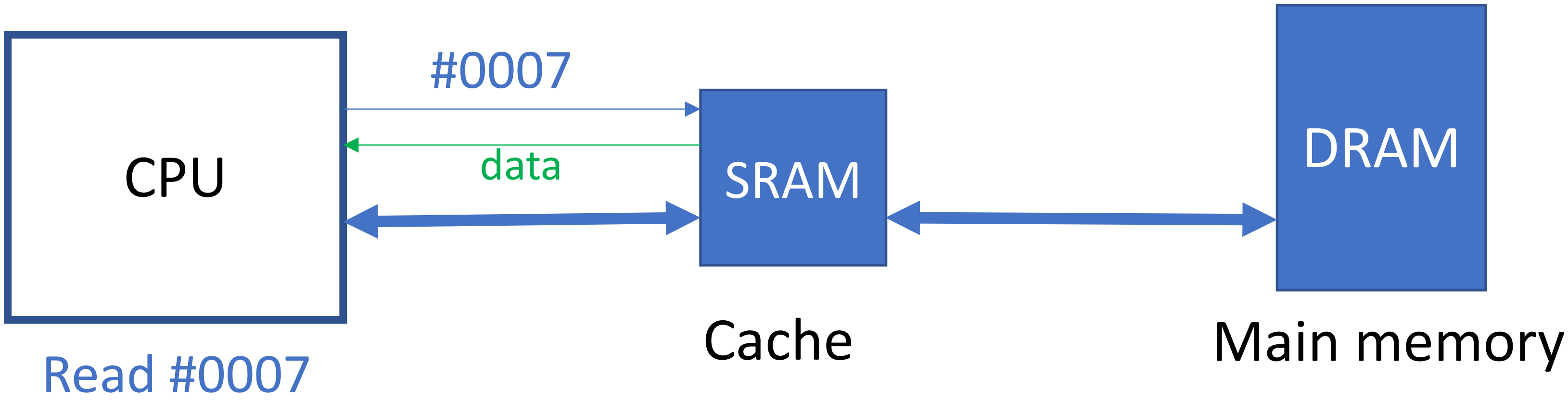
Example: (Simplified) Computer with only Level 1 Cache and Main memory




1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.

# Cache access

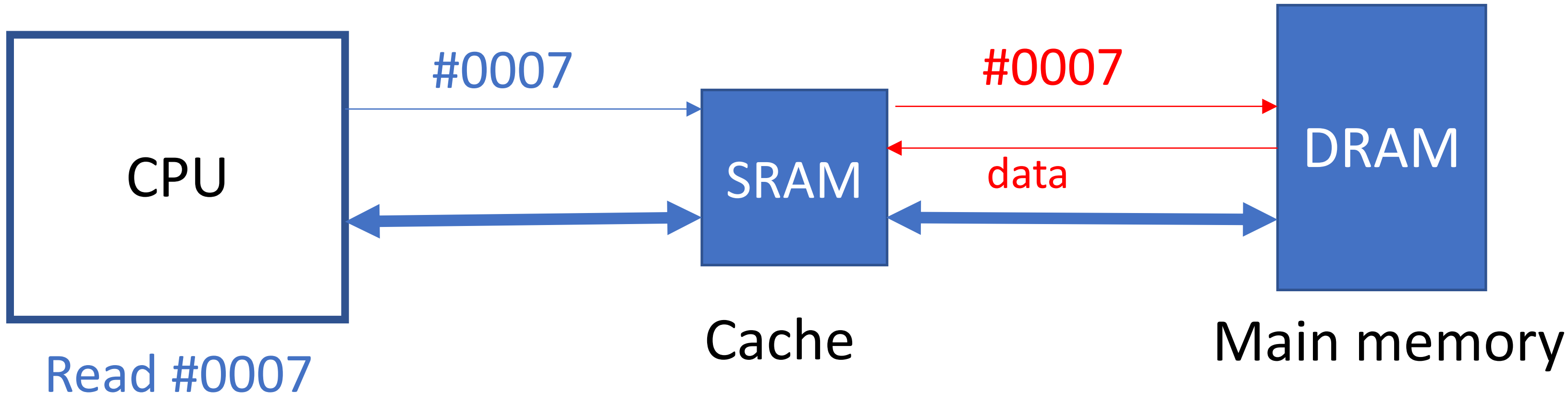
Example: (Simplified) Computer with only Level 1 Cache and Main memory




1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
   - Cache hit: The required data is present in Cache.
     So, the data is returned quickly from Cache.

# Cache access

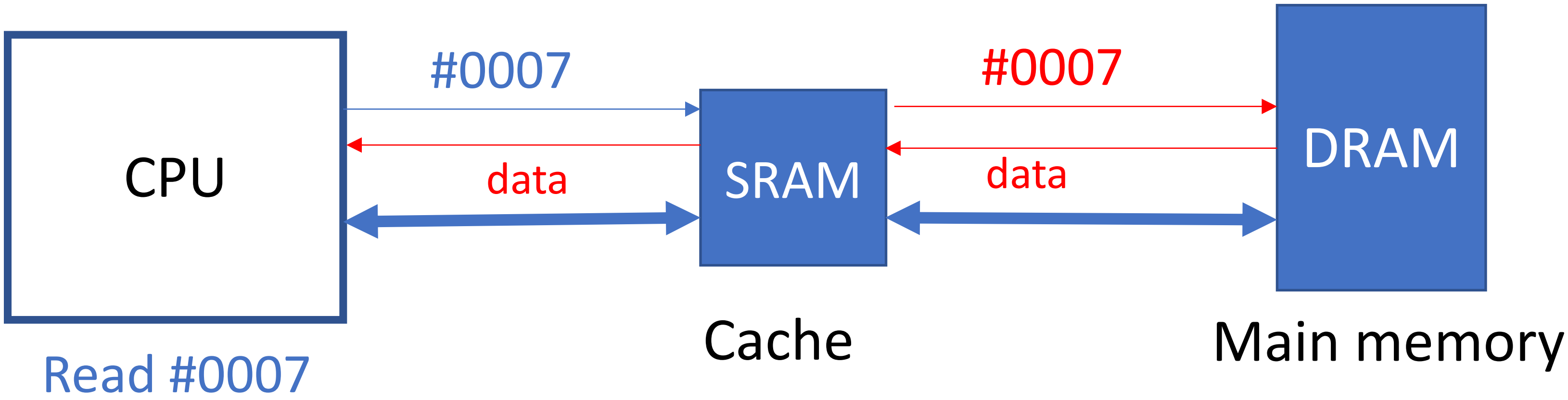
Example: (Simplified) Computer with only Level 1 Cache and Main memory




1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
   - Cache hit: The required data is present in Cache. So, the data is returned quickly from Cache.
   - Cache miss: The data is not in cache. So, get it from Main Memory and bring it to Cache

# Cache access

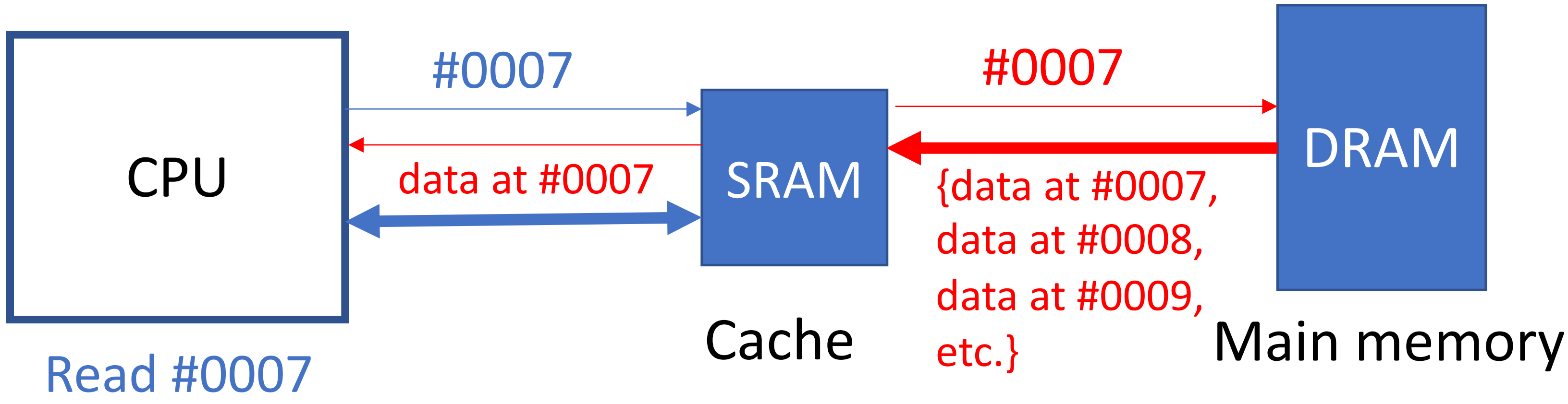
Example: (Simplified) Computer with only Level 1 Cache and Main memory




1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
   - Cache hit: The required data is present in Cache. So, the data is returned quickly from Cache.
   - Cache miss: The data is not in cache. So, get it from Main Memory and bring it to Cache and finally provide it to CPU. → There is a performance penalty ☹

# Cache access: Spatial locality

Example: (Simplified) Computer with only Level 1 Cache and Main memory

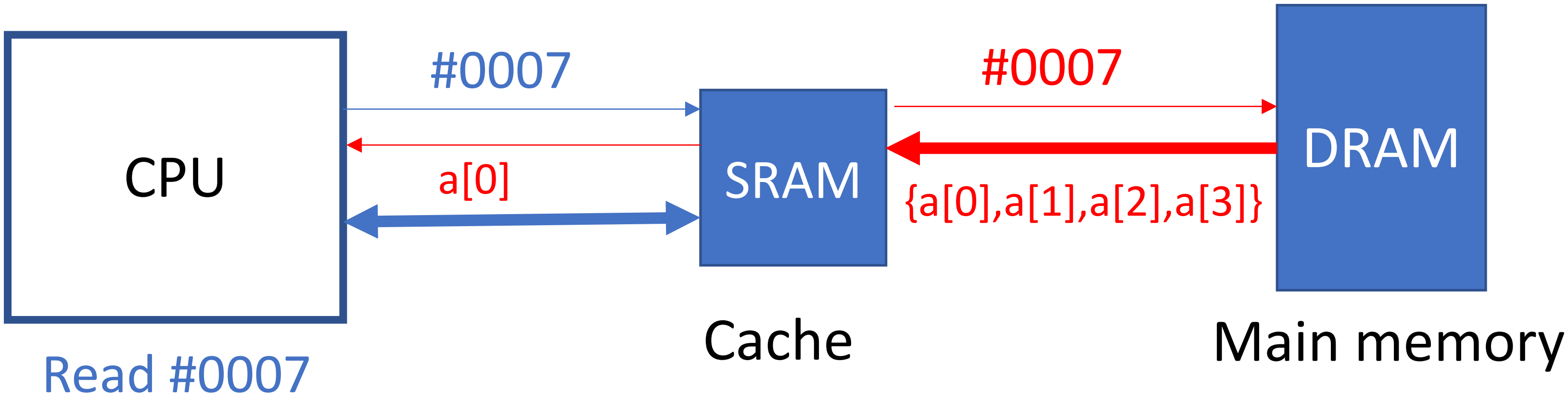



When the required piece of data is loaded from Main Memory to Cache, other nearby data blocks are also copied into the Cache.

Example: The data requested by CPU is at #0007 in Main memory.
When it is copied into Cache, data blocks from #0008, #0009, etc. are also copied into Cache.
This increases the chances of 'Cache Hit' in the future.

# Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



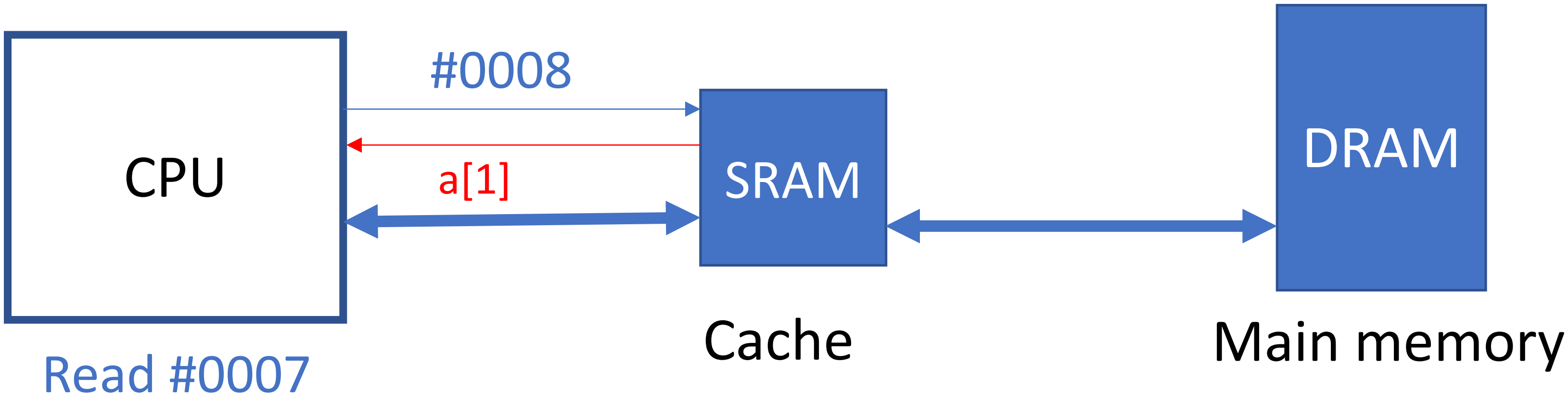

```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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

Assume the array starts from #0007. Initially all data is in DRAM.

CPU wants a[0]: Data is initially not in Cache. So, a[0] along with a[1], a[2], a[3] are fetched from Main memory and brought to Cache.

# Cache access: Spatial locality : Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



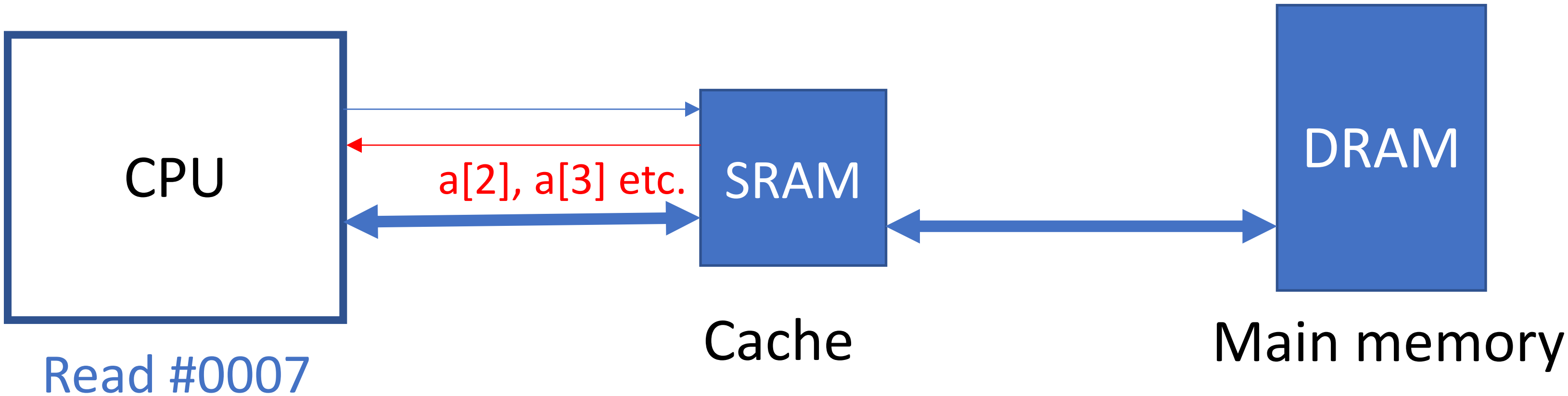

```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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

Assume the array starts from #0007.

CPU wants a[1]: Since a[1] is in Cache, there is a Cache Hit. Hence, a[1] is provided to CPU quickly.

# Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



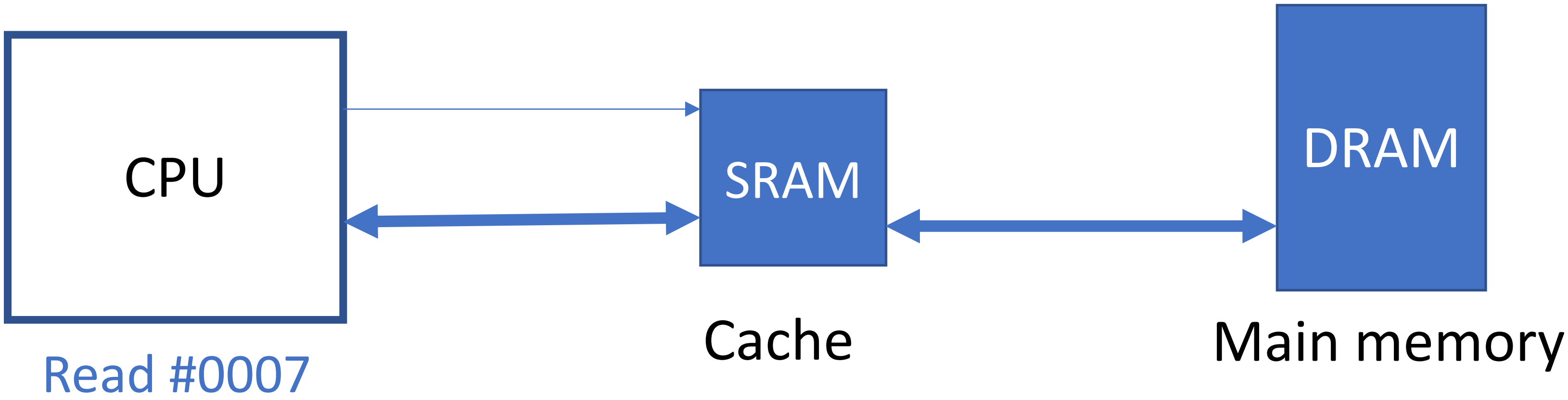

```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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

Assume the array starts from #0007.

Similarly, a[2] and a[3] are provide to the CPU from Cache.

# Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory




```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

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

Assume the array starts from #0007.

If there is not enough space left in Cache, then some old data is deleted from Cache to create space for new data.

# Locality example

Both functions compute the sum of the elements of an input 2D matrix. Which one has better locality?

```
int sum_2d_array1(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                        sum =   sum + a[i][j];
        return sum;
}
```

Computes the sum over a row.
a[0][0]+a[0][1]+…
+a[1][0]+a[1][1]+…

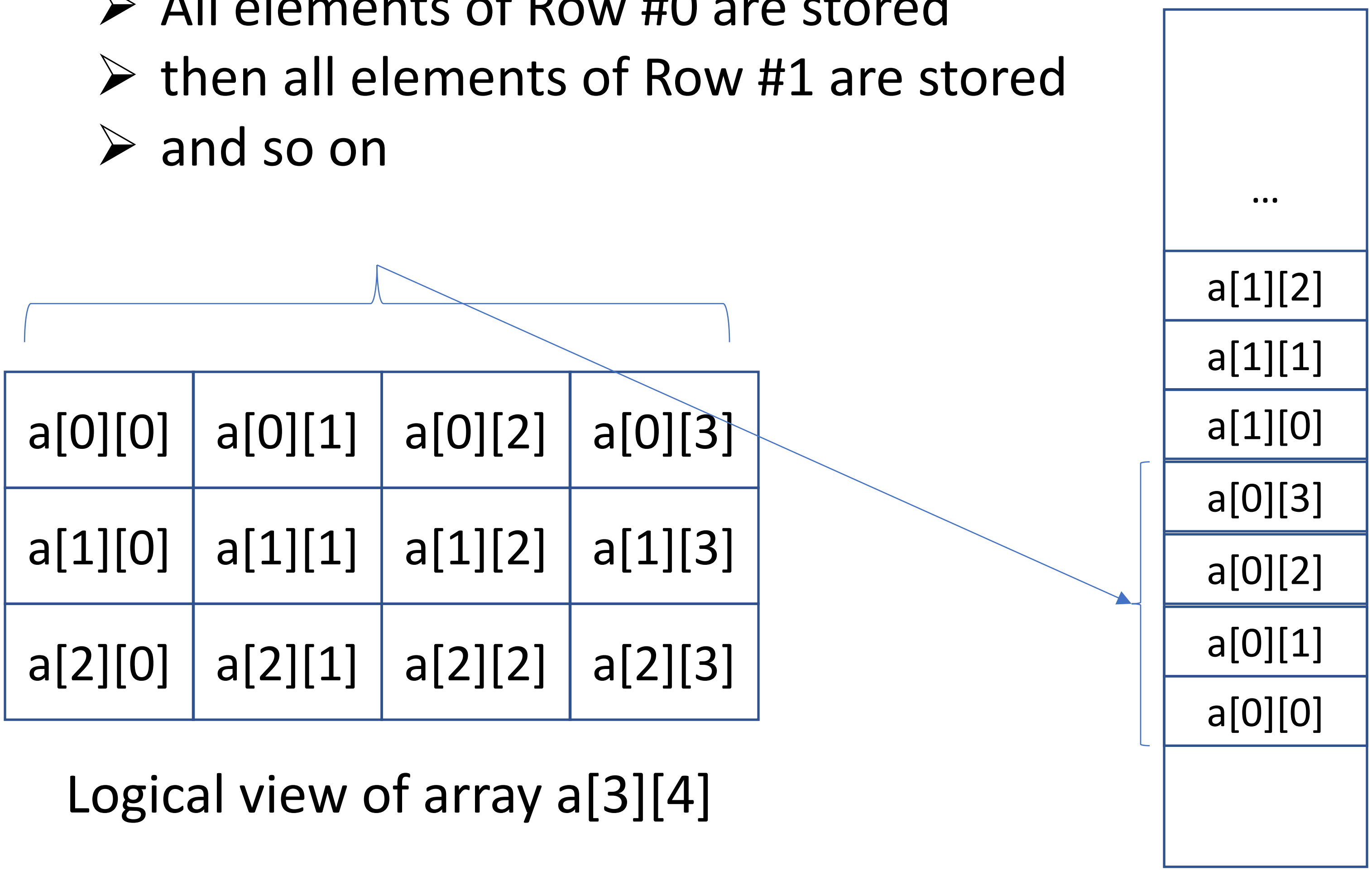
```
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

Computes the sum over a column.
a[0][0]+a[1][0]+…
+a[0][1]+a[1][1]+…

# Recap: Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on




Logical view of array a[3][4]

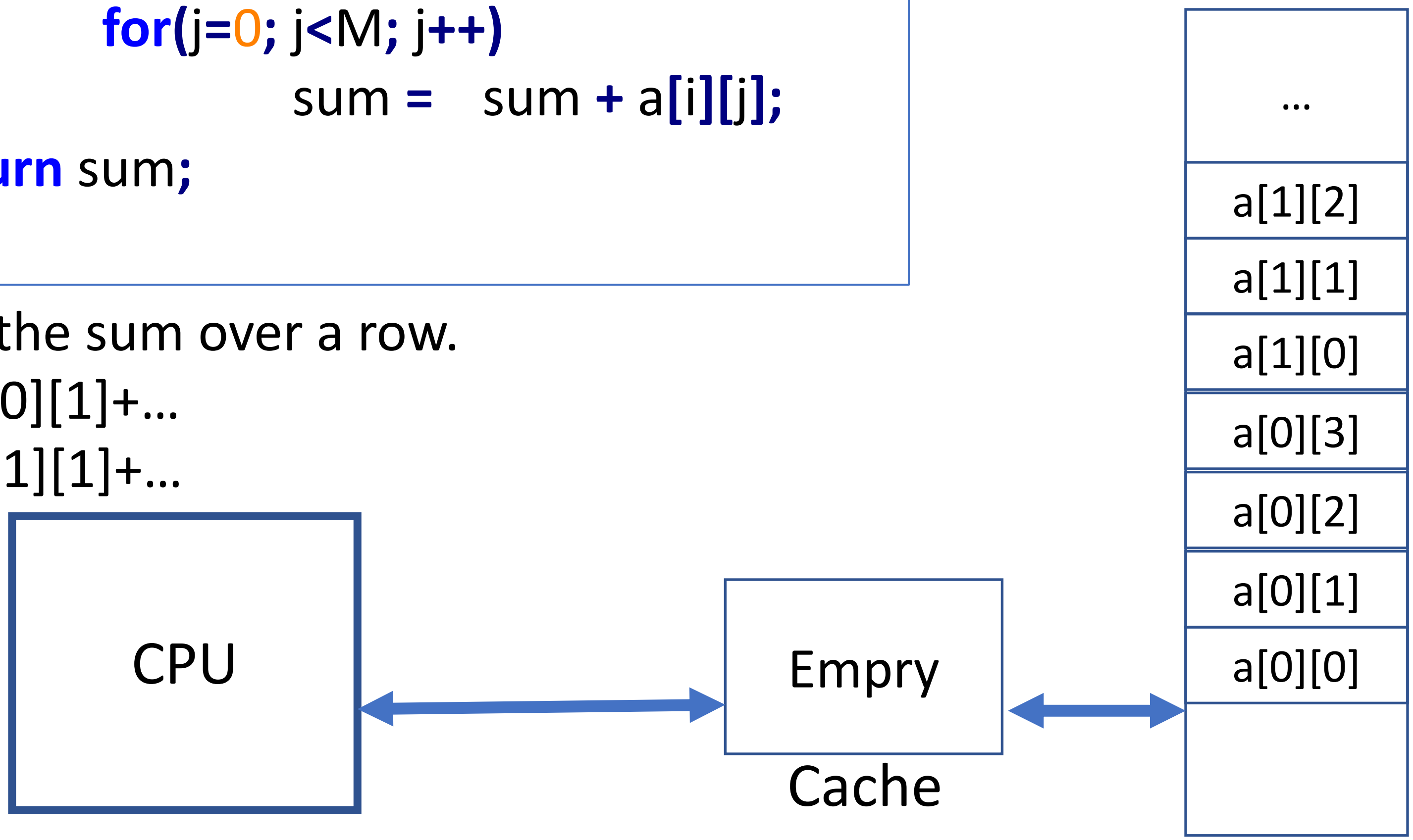
Memory layout

# Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                        sum =   sum + a[i][j];
        return sum;
}
```

Array elements are initially in the Main Memory.

Computes the sum over a row.
a[0][0]+a[0][1]+…
+a[1][0]+a[1][1]+…



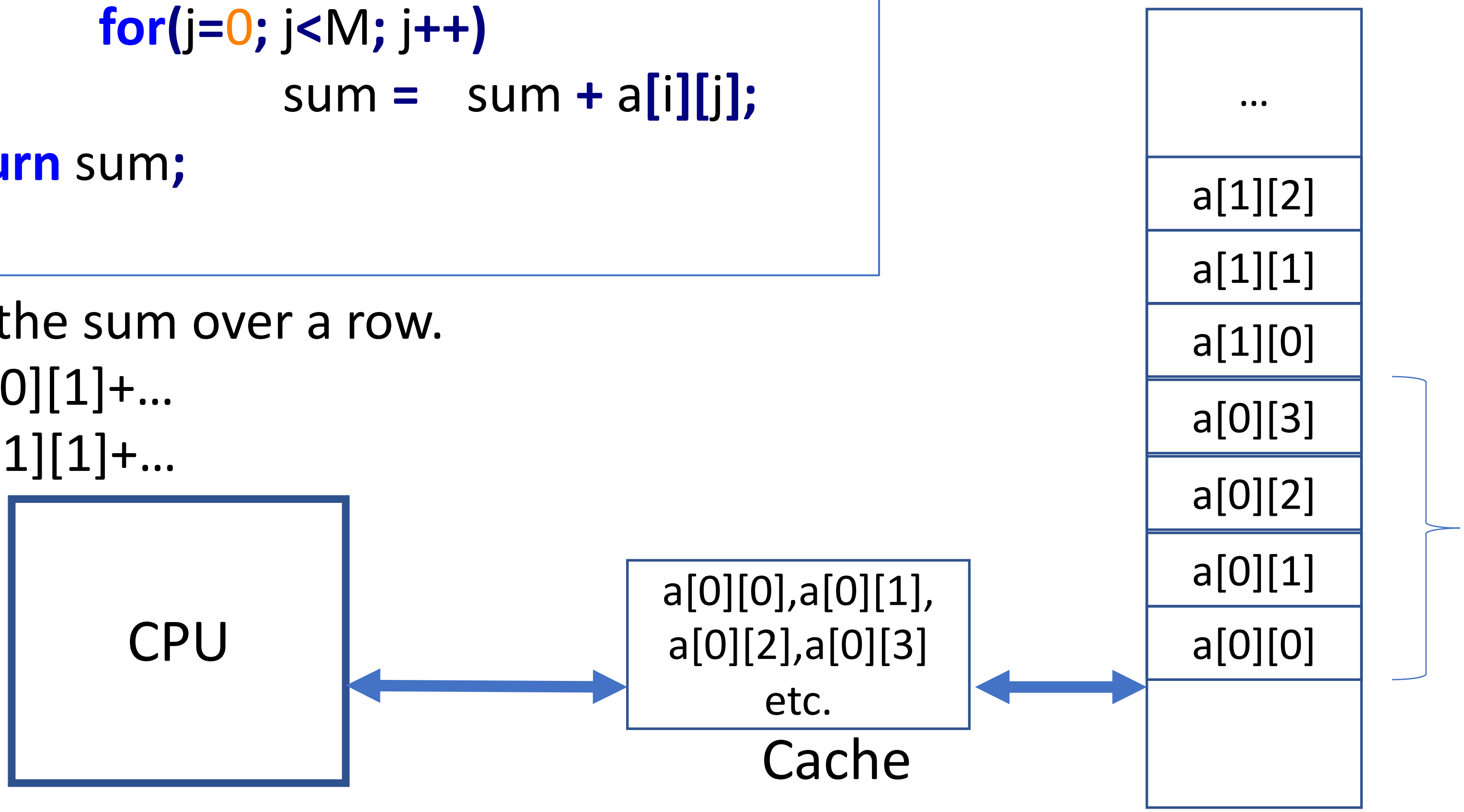

1. CPU requires a[0][0] to compute sum = sum + a[0][0]

# Locality example: the first case

```c
int sum_2d_array1(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                        sum =   sum + a[i][j];
        return sum;
}
```

Array elements are initially in the Main Memory.

Computes the sum over a row.
a[0][0]+a[0][1]+…
+a[1][0]+a[1][1]+…

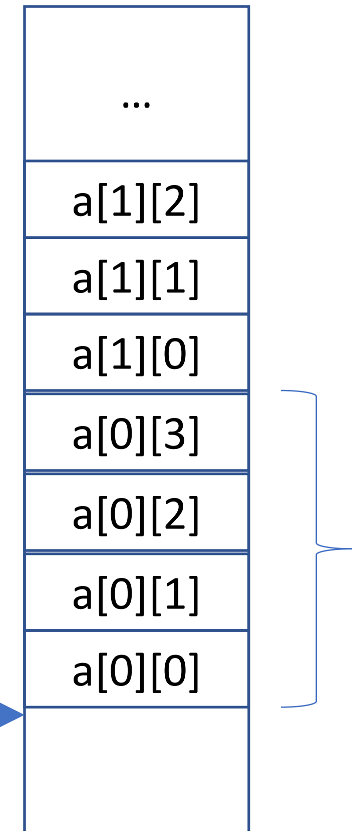



1. CPU requires a[0][0] to compute sum = sum + a[0][0]
2. Due to ‘spatial locality’, say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

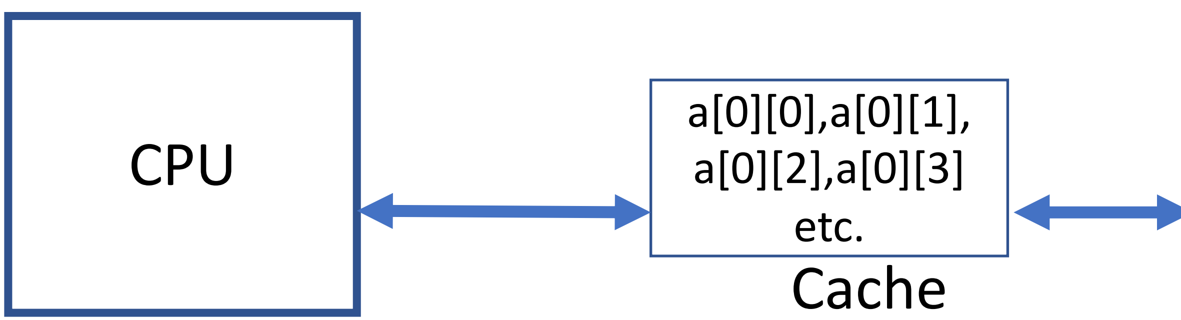
# Locality example: the first case

```c
int sum_2d_array1(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                        sum =   sum + a[i][j];
        return sum;
}
```

Array elements are initially in the Main Memory.




Computes the sum over a row.
a[0][0]+a[0][1]+…
+a[1][0]+a[1][1]+…



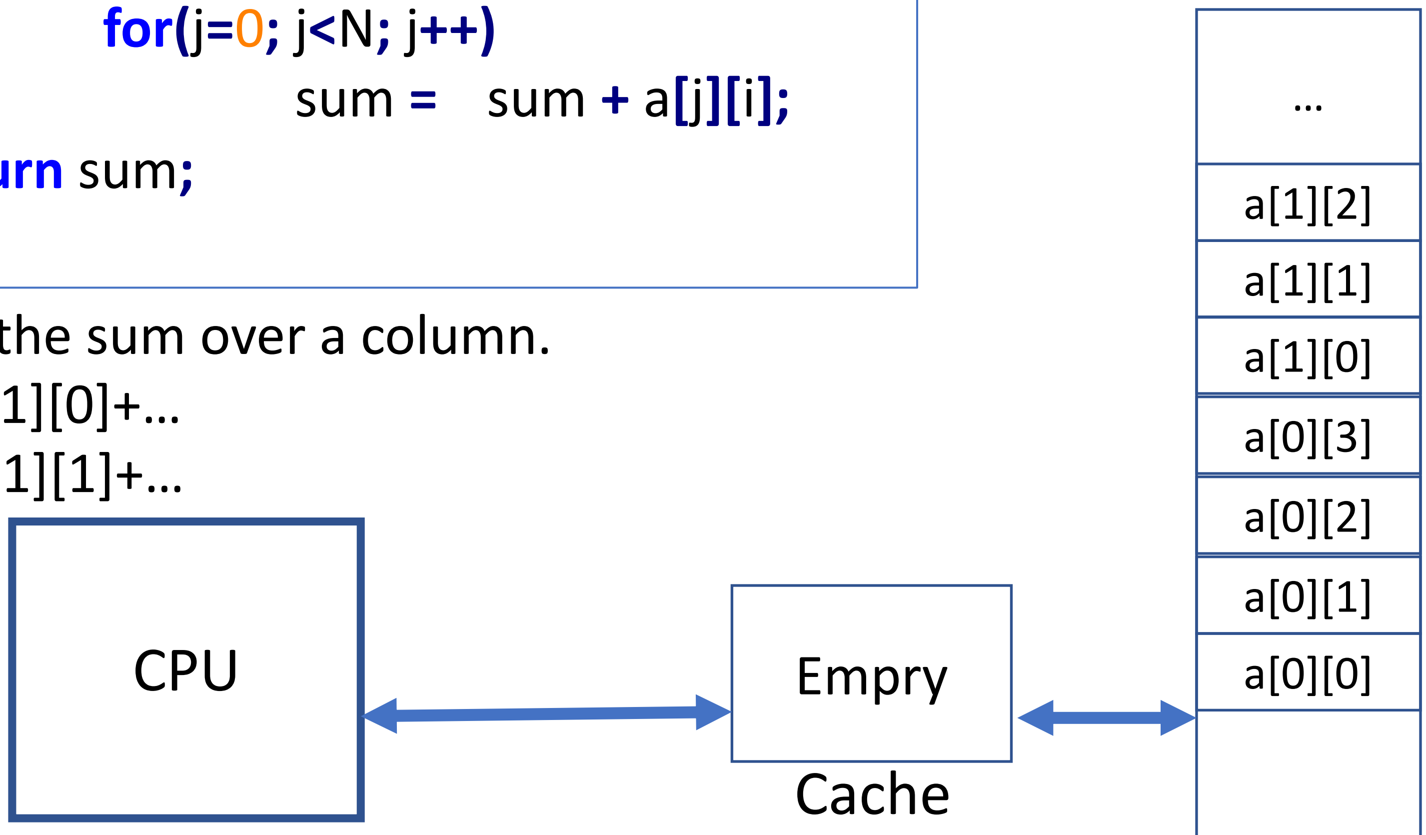

3. CPU computes sum=a[0][0]+a[0][1]+a[0][2] … by reading the elements from Cache. [Each access takes 4 cycles]
4. **Advantage: Cache hit happens most of the times.**

# Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

Array elements are initially in the Main Memory.

Computes the sum over a column.
a[0][0]+a[1][0]+…
+a[0][1]+a[1][1]+…



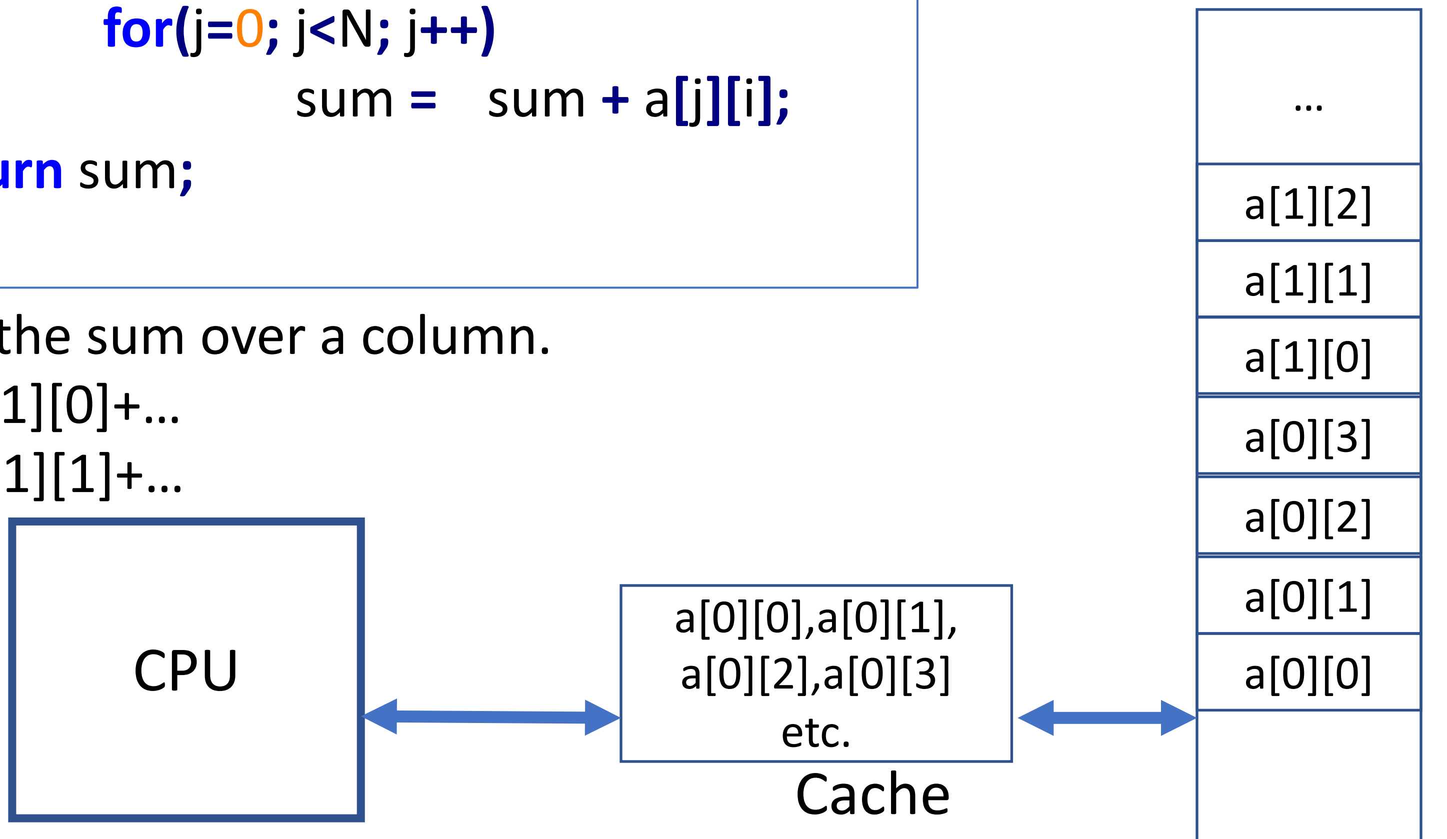

1. CPU requires a[0][0] to compute sum = sum + a[0][0]

# Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

Array elements are initially in the Main Memory.

Computes the sum over a column.
a[0][0]+a[1][0]+…
+a[0][1]+a[1][1]+…

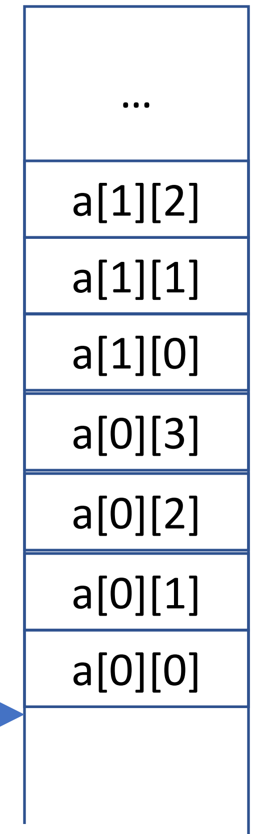



1. CPU requires a[0][0] to compute sum = sum + a[0][0]
2. Due to ‘spatial locality’, say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

# Locality example: the second case

```c
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

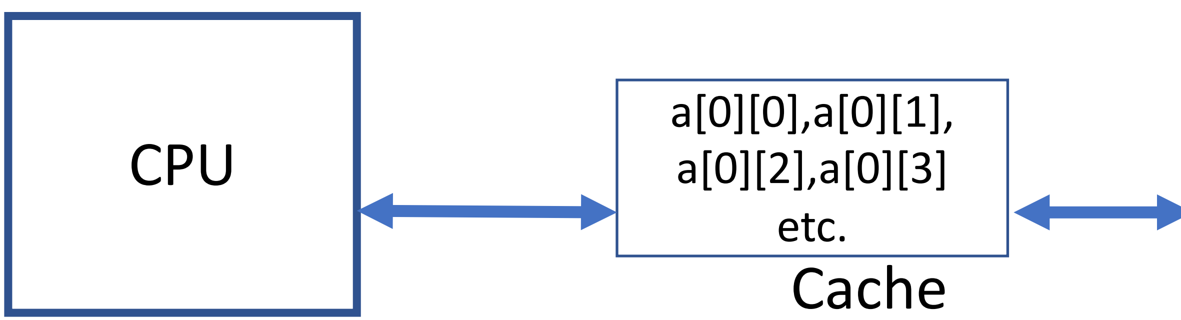
Array elements are initially in the Main Memory.




Computes the sum over a column.

a[0][0]+a[1][0]+…

+a[0][1]+a[1][1]+…




3. However, CPU to computes sum=sum+a[0][0]+a[1][0]+a[2][0]+…
4. Since, none of {a[1][0], a[2][0], …} are in the Cache. Hence, Main Memory is accessed for each of them. [100 cycles for every access]

# Locality example: the second case

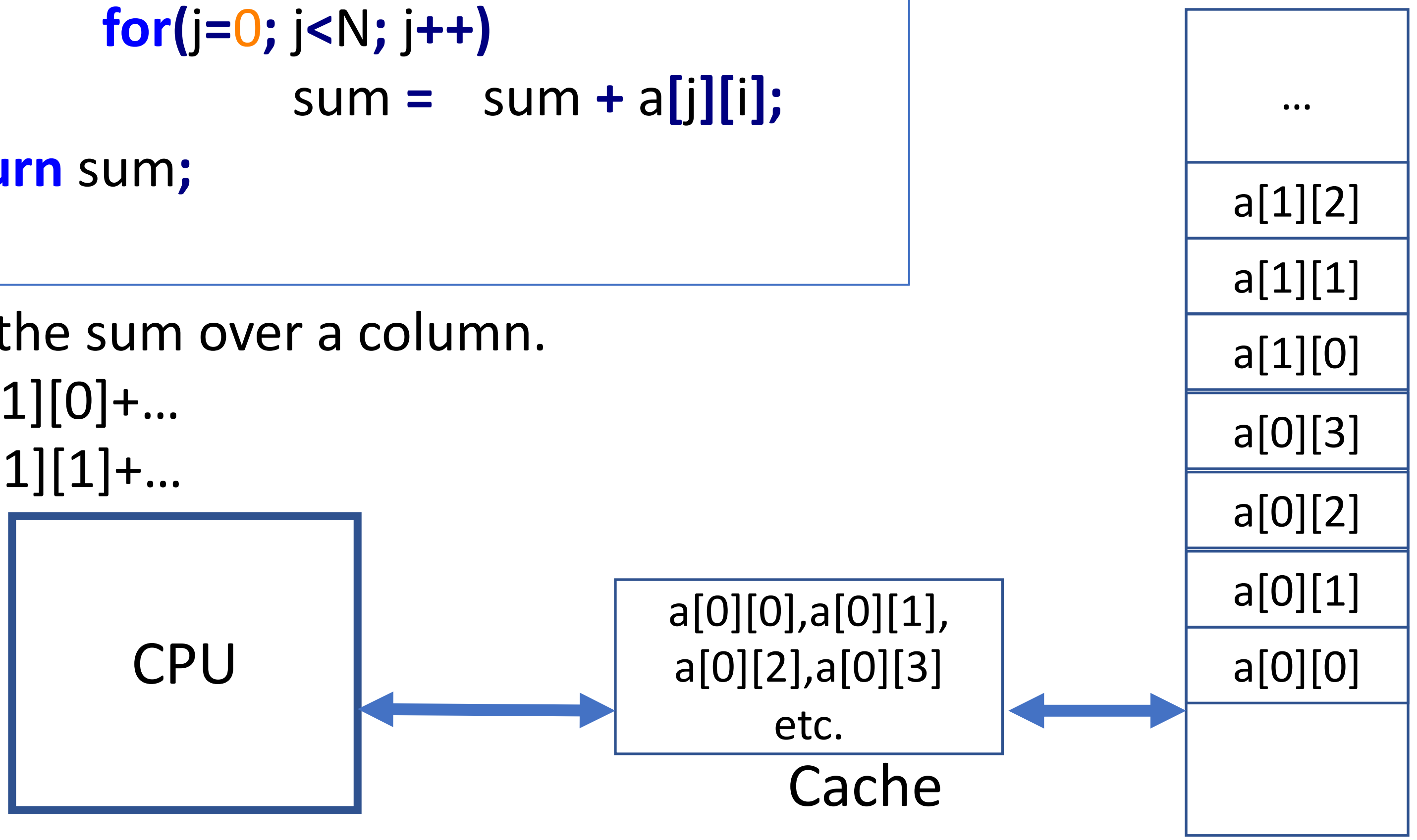
```
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

Array elements are initially in the Main Memory.

Computes the sum over a column.

a[0][0]+a[1][0]+…

+a[0][1]+a[1][1]+…




**Disadvantage: Cache miss happens always**

# Locality example: conclusions

```
int sum_2d_array1(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<N; i++)
                for(j=0; j<M; j++)
                        sum =   sum + a[i][j];
        return sum;
}
```

**High Cache hit rate!**
**Hence, much**
**faster execution ☺**

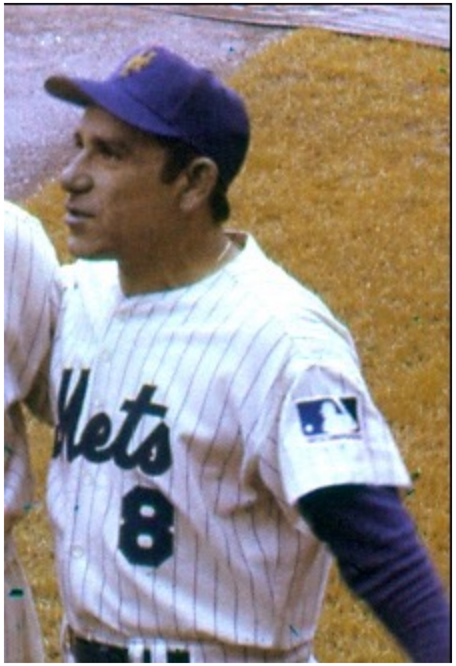
```
int sum_2d_array2(int a[N][M]){
        int i, j, sum=0;
        for(i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum =   sum + a[j][i];
        return sum;
}
```

Always Cache miss.
Order of magnitude
slower execution ☹

This is where the difference between a Java programmer and a C programmer becomes apparent.

# Theory vs practice

*"In theory there is no difference between theory and practice. But in practice there is."* - Yogi Berra




We also see 'theory vs practice' when we run algorithms.