In summary, good things to do,

- Repeated references to the local variables are good (temporal locality). Cache them in the registers!
- Stride-1-reference patterns are also good because all caches store data sequentially as contiguous blocks.
Now consider this program

```c
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for(int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}
```

- C stores arrays in a row-major order.
- Again, stride-1-reference pattern.
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}

<table>
<thead>
<tr>
<th>a[i][j]</th>
<th>j=0</th>
<th>j=1</th>
<th>j=2</th>
<th>j=3</th>
<th>j=4</th>
<th>j=5</th>
<th>j=6</th>
<th>j=7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>1 [m]</td>
<td>2 [h]</td>
<td>3 [h]</td>
<td>4 [h]</td>
<td>5 [m]</td>
<td>6 [h]</td>
<td>7 [h]</td>
<td>8 [h]</td>
</tr>
</tbody>
</table>
int sumarrayrows(int arr[4][8]) {
    int sum = 0;

    for (int i = 0; i < 4; i++)
        for(int j = 0; j < 8; j++)
            sum += arr[i][j];

    return sum;
}

Miss Ratio = 8/32 = 0.25
Now what if we reference the array in column-major order?

```c
int sumarraycols(int arr[4][8]) {
    int sum = 0;

    for(int j = 0; j < 8; j++)
        for (int i = 0; i < 4; i++)
            sum += arr[i][j];

    return sum;
}
```

If the cache is large enough (to hold the entire array) we may get away with this.

But it’s highly unlikely.

So...
Access a[0][0]. Cache miss!
- So we load block containing a[0][0] along with a[0][1], a[0][2], a[0][3] onto the cache.
- In the second iteration of the innermost loop, we access a[1][0]. Cache miss again!
- Now load block containing a[1][0] along with a[1][1], a[1][2], a[1][3] onto the cache.
- In the third iteration, we access a[2][0]...

Try working on this example for the following params,
\( m = 8, S = 4, B = 16 \) and \( E = 1 \).

How many misses do you get?
cache = (cache_set_t*) malloc
(size of (__) * S);

A pointer to
a set
A pointer to
an array of
sets.
set[0].

cache[0] = (cache_line_t*) malloc
(size of (c_line_t) * E)

cache[0][1]. valid_bit
Linked List

```
cache_line_t * newline = malloc ( sizeof ( cache_line_t ) );
head
```
Memory (Physical)

Memory → organized as an array of M contiguous byte-sized cells.

Each byte → has a unique address.

CPU → when it wants to access something from the main memory, it uses its physical address.

Get me the value at address 2.

Diagram:

- CPU
- Array of memory cells
- Address labels: 0, 1, 2, 3, 4, 5
- M-1
Problems

0 Program A

Need to store a variable X at 0x805c.

Let them run together.

What will be at 0x805c?

Solution?

Not let them run together. X

Program B

Need to store a function Y at 0x805c.
"not_a_virus.exe"

Access the address space allocated to Proj B.

Has a bad pointer → modifies a value in A.
4. Some parts of the memory may be allocated to the OS, hardware devices.

Solution?

Virtual Addressing

CPU accesses main memory by generating a virtual address that gets converted to a physical address.

Virtual address $\rightarrow$ physical address

Address translator