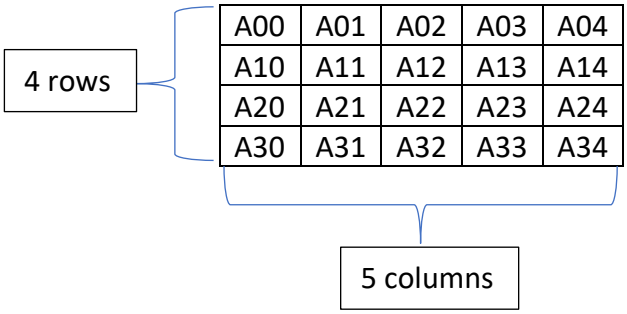
Hardware Systems (COSC 2200)

Part 2: 12 points

SIMD Image Lab

To do this lab, we need to get two perquisites done. 1st is understanding that a 2D array is stored as a 1D array internally. Also, 2D array can be stored as a 1D array. 2nd is to know the names of single precision floating point intrinsic functions because we will be using these now. So, far we only covered double precision floating point numbers. Single precision needs less storage (32 bits vs 64 bits). In terms of bytes, 4 bytes for float data type vs 8 bytes for double data type.

2D Matrix basics: A matrix with 4 rows and 5 columns is written as A[4][5].
An element can be accessed using A[i][j] where the first index i denotes row number and second index j denotes column number. The elements are indexed as shown below:



Memory layout: A matrix A[4][5] has 20 elements which is stored in memory row by row (row-major form) as follows. A[i][j] is equivalent to A[i*n + j] where n is the number of columns.
So, in the above matrix A[2][3] will have the index = (2*5 + 3) = 13 in memory.
The table below shows 2D array A[4][5] and 1D indexing for the same data as 1D array:

2D: i , j	0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4	3,0	3,1	3,2	3,3	3,4
1D: i*n + j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	18	19	20

Row-major storage of 2D array elements. The first matrix figure shows row index, column index pairs. The second figure shows the actual array elements. The third figure shows 1 dimensional array organization of the array elements from the second figure.

e.g., A and B have 3 rows and 4 columns.

00	01	02	03
10	11	12	13
20	21	22	23

1	2	3	4
2	1	4	3
1	1	1	1

Row-by-row storage
1, 2, 3, 4, 2, 1, 4, 3, 1, 1, 1, 1

Sample code to get used to 1D array access

```
int ROWS = 3;
int COLS = 4;
int A[ROWS * COLS];

int sum = 0;
// find sum of all values in the array A
for( int i = 0; i < ROWS; i++)
{
   for( int j = 0; j < COLS; j++ )
   {
       sum = sum + A[ i * COLS + j];
   }
}
```

Single precision intrinsics

So far you have been using double-precision intrinsics. Now you will use single precision intrinsics as shown below:

Single precision	Double precision
Variable __m256 val Size 32 bits	Variable __m256**d** val Size 64 bits
_mm256_load_p**s**	_mm256_load_pd
_mm256_add_p**s**	_mm256_add_pd
_mm256_sub_p**s**	_mm256_sub_pd
_mm256_mul_p**s**	_mm256_mul_pd
_mm256_div_p**s**	_mm256_div_pd
_mm256_set_p**s**	_mm256_set_pd
_mm256_store_p**s**	_mm256_store_pd

Memory allocation
`float *arr =`
`(float *)_mm_malloc( N*sizeof(float), 32);`
	`double *arr =`
	`(double*)_mm_malloc( N * sizeof( double ), 32);`