

CSC290/420 ML Systems for Efficient AI Compound Structures / Memory

Sreepathi Pai

September 21, 2026

URCS

Outline

Memory

Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

Outline

Memory

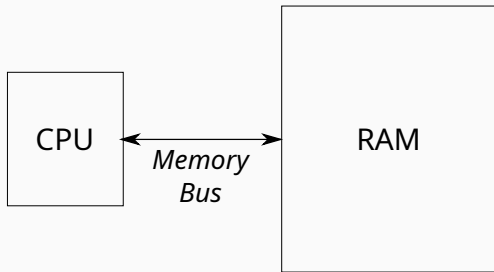
Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

A Simple Memory Organization



- The CPU executes instructions
- Fetches instructions and data from memory
- The memory, or "RAM" (for random-access memory), is connected to the CPU using a memory *bus*

Memory buses and data transfers

- Buses vary in size from processor to processor
 - Most CPUs have 64-bit buses
- Many GPUs with GDDR memory have 256-bit to 384-bit buses
 - “Graphics Double Data Rate” RAM
- GPUs with HBMs have 4096-bit buses
 - High-bandwidth Memory
- A memory controller interfaces the CPU and memory
 - Responsible for the operating the bus
 - Used to be separate component, now part of the processor

Memory Abstraction

- Memory is like a very large array
- Each location in memory has an address at the granularity of a byte
 - Byte-addressable
- Load/Store instructions usually read/write up to register-size data
- But, memory moves data to the CPU in bus-size chunks
 - Requires the use of *memory-level parallelism* (MLP) for performance
 - I.e., multiple, independent load/store instructions to locations in the same chunk

Memory Layout

- Most data discussed so far fits in a register
 - How is this “primitive” data stored in memory?
- Aggregations of primitive data
 - Structures and Unions
 - Arrays

Primitive Data Layout: Little and Big-endian

- Storing a 32-bit value `0xDEADCAFE` in memory
- Big endian: Most significant byte at lower addresses
- Little endian: Least significant byte at lower addresses

address	x	$x + 1$	$x + 2$	$x + 3$
big-endian	0xDE	0xAD	0xCA	0xFE
little-endian	0xFE	0xCA	0xAD	0xDE

- Different machines use different conventions
 - Intel/AMD usually little endian
 - SPARC/PowerPC usually big endian
 - ARM can switch between the two
- Big endian is sometimes called network byte order
 - Similar problem: which byte of a word gets on the wire first?

Outline

Memory

Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

Record Types

```
struct point {  
    float x;  
    float y;  
};
```

```
struct point pt;  
struct point pts[10];  
struct point *polygon;
```

- struct holds multiple values
- Each *field* of struct has a name
- struct can be nested

Recursive structures

```
struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
};
```

- This is a *recursive* structure
 - Possibly from a binary tree
- The “self-references” must be pointers

struct interactions with pointers

```
struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
};  
  
struct node *head;
```

Which of these accesses head's right node?

- `*head.right`
- `(*head).right`

The -> operator

```
struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
};  
  
struct node *head;
```

Since field access (.) has higher precedence than dereference, you must use (*head).right.

- Alternative: head->right, which has same precedence as .

Structure Layout

```
#include <stdio.h>

struct node {
    int value;
    struct node *left;
    struct node *right;
};

int main(void) {
    printf("%d\n", sizeof(struct node));
}
```

Output (on a LP64, i.e. long and pointers are 64-bits, system)?

- 20
- 24
- 32

Structure Memory Layout

Tight packing:

value	left	right
[0123]	[01234567]	[01234567]

Packing with “holes” (also called padding)

value	left	right
[0123]	[0123]	[01234567]

- Structure layout in memory is implementation defined
- Usually:
 - Struct size is a multiple of largest field
 - Each individual field is naturally aligned

Unions

```
union intvar {  
    char c;  
    short s;  
    int i;  
    long l;  
};
```

- Like struct, union contains fields
- However, all fields in a union *overlap* in memory
 - At most one contains valid data

Union Memory Layout

```
c|  
s-|  
i---|  
l-----|  
[01234567]
```

- This union occupies 8 bytes of memory
 - The size of its largest field
- But all fields overlap
 - Writing to one changes the others as well
- Writing to one field, and reading that data through another field is allowed
 - But it is implementation-defined

Outline

Memory

Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

Memory Allocation

- From a program's perspective, memory is obtained using an *allocator*
 - The memory is then reserved for the program and unavailable for other programs
 - Must be de-allocated once it is no longer needed
 - Memory is allocated from the "heap"
- Manual memory management (C, C++, CUDA)
 - Programmer must explicitly deallocate memory
- Garbage collection (Java, Python, etc.)
 - Deallocation is automatic

C Runtime Allocator

The C standard library supports the following *heap* allocators, all require including `stdlib.h`:

- `malloc`: allocate memory of a certain size
- `calloc`: allocate memory and zero it
- `realloc`: change size of memory (may move data)
- `free`: free memory allocated by the above functions
- All allocators return a pointer to the newly allocated region of memory or `NULL` if they fail.

C memory allocation

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);

int *p;

// allocate 1000 ints, note size is size in bytes
p = (int *) malloc(1000 * sizeof(int));

// --- OR --- //

// or allocate 1000 ints initialized to zero
p = (int *) calloc(1000, sizeof(int));

// free allocated memory
free(p);
```

Allocating an array on the heap

```
int *alloc_and_init_array(int N) {  
    int *x;  
  
    x = (int *) malloc(N * sizeof(int))  
  
    // initialize it  
    for(int i = 0; i < N; i++) {  
        x[i] = i*i;  
    }  
  
    return x;  
}
```

- Note we're returning the value of *x*
 - I.e. the address *in* *x*, not the address *of* *x*
- Since the address is on the heap, it is independent of the function call
- You can treat pointers to the heap as any other pointer

Common Bugs using Dynamic Memory

- Memory leaks
 - When you lose the pointer to an allocated region of memory
 - Can't be freed until program exits
 - “harmless”, program just consumes more and more memory as it runs
- Reading uninitialized memory
 - Memory from `malloc` should be initialized before reading it
- Out-of-bound accesses
 - Pointer points outside the allocated region
 - Undefined behaviour!
- Use-after-free
 - Attempt to access memory region after `free`
 - Essentially a dangling pointer pointing to the heap
 - Dangerous!
- Double-frees
 - Trying to call `free` on the same pointer twice
 - Undefined behaviour!

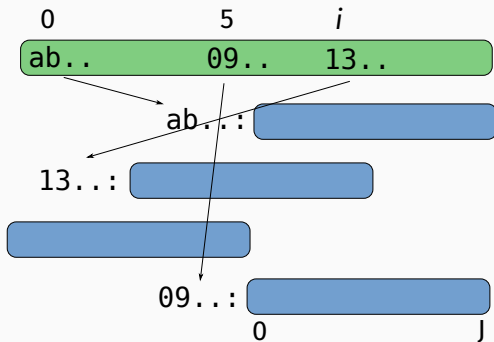
Arrays and Matrices

- Memory is 1-dimensional
 - addresses are linear ranging from 0 to some N
 - each address is a byte
- 1-D arrays are similar
 - indices range from 1 to some n , where n is the number of elements
 - i.e. arrays have a type and each index references an element
 - translation of index to address:
$$address = index * sizeof(element)$$
- Most languages also support multidimensional arrays
 - $A[i][j]$
- These must be mapped to 1-D memory
 - Multiple schemes exist

Implementing Multidimensional Arrays

- Scheme 1: All data is stored contiguously in a 1-D array
 - Requires translation of a multidimensional index (i, j, k) into a 1-D index
- Scheme 2: All dimensions are 1-D arrays of *pointers*, except the last which contains data
 - Requires multiple pointer dereferences $A[i][j][k]$: load $A[i]$ (a 1-D array), then look up j (another 1-D array), and look up k -th element as data

Pointer-based Scheme



- Example of a 2-D array implemented as an 1-D array of pointers to 1-D data arrays

Contiguous Storage

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \quad \begin{matrix} [0 & 1 & 2 & 3 & 4 & 5] & \text{row-major} \\ [0 & 3 & 1 & 4 & 2 & 5] & \text{column-major} \end{matrix}$$

0 5

- For access $A[\text{row}][\text{col}]$
- Row major: $\text{index} = \text{row} * \text{COLS} + \text{col}$
- Column major: $\text{index} = \text{col} * \text{ROWS} + \text{row}$
- The address then multiplies the index by the size of the element.
- Extension to more than 2 dimensions?

Outline

Memory

Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

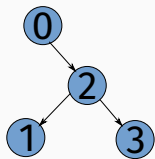
Dense and Sparse Matrices

- A sparse matrix is one in which the number of non-zeroes (NNZs) is significantly lower than the number of zeroes
- Occur commonly in large linear systems used in computational science
- Very common when storing graphs using *adjacency matrix notation*
- Most weight matrices in ML are *dense*, but increasing interest in sparse matrices

Sparse Matrices

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



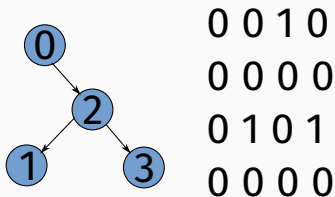
$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- Dense matrix
- Sparse matrix
- Graph
- Adjacency matrix representation of graph

Storing Sparse Matrices

- Various sparse formats
- COO - coordinate format
- CSR - compressed sparse row
- CSC - compressed sparse column

COO storage

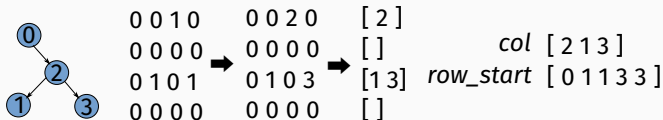


row [0 2 2]

col [2 1 3]

- Two arrays (*row* and *col*) track positions of non-zeroes
- An additional data array may be present to store the values in the cells
 - Here, all values are 1, so no data array needed

CSR storage



- First, the positions of the 1s are noted
 - can be seen as replacing 1s with their index
- Second, the zeroes are dropped
 - yields variable length column arrays for each row
- Third, these column arrays are concatenated to form a 1-d array `col`
- Fourth, a `row_start` array indicates the start and end of each row
 - start: `row_start[row]`
 - end (exclusive): `row_start[row+1]`
- A data array may also be present

- Indirect accesses
 - Need to load multiple values to get data
 - Similar to pointer-based schemes for multidimensional arrays
- Hard to vectorize
 - hence other formats: ELLPACK (block sparse)
- Changing a zero to a non-zero requires rewriting the structure!
 - mostly used for read-only data

Outline

Memory

Structures and Unions

Arrays, Matrices, and Tensors

Sparse Data Structures

Performance and Efficiency Considerations

Considerations

- Byte-addressable memory, with data is transferred in multibyte blocks
- Is space usage efficient?
- Is data access efficient?
- Can data access code be vectorized?

AoS vs SoA

```
struct pt {  
    float x;  
    float y;  
};  
  
struct pt pts[10]; // array of structures
```

OR

```
struct pts {  
    float *x;  
    float *y;  
};  
  
struct pts p; // structure of arrays  
  
p.x = malloc(...);  
p.y = malloc(...);
```

Row-major vs column-major

```
for(row=0; row < NROWS; row++)  
    for(col=0; col < NCOLS; col++)  
        out[row * NCOLS + col] = in[col * NROWS + row];
```

Sparse vs Dense

```
for(row = 0; row < NROWS; row++) {  
    for(j = row_start[row]; j < row_start[row+1]; j++) {  
        column = col[j];  
    }  
}
```