MMU (Memory Transactions and Runtime Interfacing)

# Data Loader

Definition (source: ExecuTorch documentation):

An interface that enables the ExecuTorch runtime to read from a file or other data source without directly depending on operating system concepts like files or memory allocation.

Data Loader must fulfill following responsibilities:

1. Abstract data sources: Support reading from files, buffers, or other sources (e.g., network streams) without OS dependencies.
2. Handle .pte files or .pt files (to get metadata): Parse ExecuTorch's serialized model format or PyTorch-compatible inputs.
3. Provide an interface: Allow the runtime to fetch input tensors or model parameters without direct file or memory management.
4. Support preprocessing: Optionally transform raw input data (e.g., normalization, reshaping) to match model expectations. (This is usually done on the host.)
5. Batching and streaming: Dividing data into batches for efficient processing. And stream the data into chunks to avoid the need to load the entire data at once for processing.
6. Transfer to accelerator’s memory.

Data Loader’s abstract interface ensures flexibility and modularity. The interface must avoid OS-specific constructs such as fopen, mmap, etc. It must support reading arbitrary data chunks for .pte files /.pt files or input tensors and allow metadata access for model specific information. It must be extensible for different data sources such as files, memory buffers, streams, etc.

Dataloader requires metadata to ensure it provides correctly formatted input data to the accelerator.

The data loader’s role is to read input data (e.g., from a sensor, memory buffer, or network) and model weights, preprocess them (e.g., normalize, reshape, convert layout), and feed them to the accelerator in a format it expects. Metadata provides the information needed to perform these tasks dynamically, ensuring compatibility with the model and accelerator.

Metadata needed:

* **Input Tensor Shapes**: Defines the dimensions of input tensors (e.g., [1, 3, 224, 224] for an image). The data loader needs this to validate and reshape input data.
* **Tensor Dtypes**: Specifies data types (e.g., float32, int8) for inputs and weights, allowing the data loader to perform conversions if the accelerator requires a specific dtype (e.g., int8 for quantization).
* **Batch Size**: Indicates the number of samples per inference (e.g., first dimension of input tensor). This helps the data loader allocate buffers and process inputs correctly.
* **Layout Formats**: Determines tensor layout (e.g., NCHW vs. NHWC). PyTorch uses NCHW, but the accelerator may require NHWC, necessitating a conversion.
* **Preprocessing Steps**: Defines transformations (e.g., normalization with mean/std, resizing) to prepare inputs, ensuring they match the model’s training-time expectations.

Example code of Data loader:

Data loader interface:

namespace custom\_runtime {

class DataLoader {

public:

virtual ~DataLoader() = default;

virtual bool read(uint8\_t\* buffer, size\_t size, size\_t offset) = 0;

virtual size\_t size() const = 0;

virtual bool is\_valid() const = 0;

virtual bool seek(size\_t offset) = 0;

virtual bool get\_metadata(uint8\_t\* metadata, size\_t\* metadata\_size) = 0;

};

} // namespace custom\_runtime

Memory Data Loader:

class MemoryDataLoader : public DataLoader {

public:

MemoryDataLoader(const uint8\_t\* data, size\_t data\_size)

: data\_(data), size\_(data\_size), offset\_(0) {}

bool read(uint8\_t\* buffer, size\_t size, size\_t offset) override {

if (!buffer || offset + size > size\_) {

return false;

}

memcpy(buffer, data\_ + offset, size); // Accelerator-safe memcpy

offset\_ = offset + size;

return true;

}

size\_t size() const override { return size\_; }

bool is\_valid() const override { return data\_ != nullptr && size\_ > 0; }

bool seek(size\_t offset) override {

if (offset >= size\_) {

return false;

}

offset\_ = offset;

return true;

}

bool get\_metadata(uint8\_t\* metadata, size\_t\* metadata\_size) override {

// Optional: Return input-specific metadata (e.g., shape, dtype)

return false; // Not needed for simple buffer

}

private:

const uint8\_t\* data\_;

size\_t size\_;

size\_t offset\_;

};

The read function reads size bytes from the internal memory (data\_) starting at a specified offset, and copies that data into the provided buffer.  
 It first checks:

* If buffer is valid (not null), and if reading size bytes from offset stays within the bounds of data\_.  
   If any of these checks fail, it returns false.  
   If all checks pass, it performs a memcpy from data\_ + offset to buffer, then updates the internal offset\_ to offset + size, and returns true.

In most cases the data loader operates on the host i.e. CPU side and the data is copied to the accelerator’s memory (HBM).

# Memory Manager

The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

Memory management mostly involves management of main memory. In a multiprogramming computer, the Operating System resides in a part of the main memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

## Why is the memory manager (memory management) required?

* To allocate and deallocate the memory.
* To keep track of used memory space by the processes.
* To minimize fragmentation issues
* To properly utilize the memory
* To ensure data integrity while execution of the processes.

## Logical Address Space and Physical Address Space:

**Logical Address Space:** Logical Address also known as Virtual Address. Logical Address is generated by the CPU. Logical Address Space is a set of addresses. Logical Address Space is defined as the size of the process meaning the logical address space is the range of addresses that a process can use during its execution.

**Physical Address Space:** Physical Address is also known as Real Address. Physical Address Space is the set of actual addresses in the physical memory. It is an address seen by the memory unit (i.e. the one loaded into the memory address register of the memory). The set of all physical addresses corresponding to these logical addresses is known as Physical address space. **A physical address is computed by MMU.** **The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit (MMU).** The physical address always remains constant.

**Static and Dynamic Loading:** Loading a process into a memory is done by the loader. There are two types of loading- Static loading and Dynamic loading.

1. Static Loading: Static Loading:

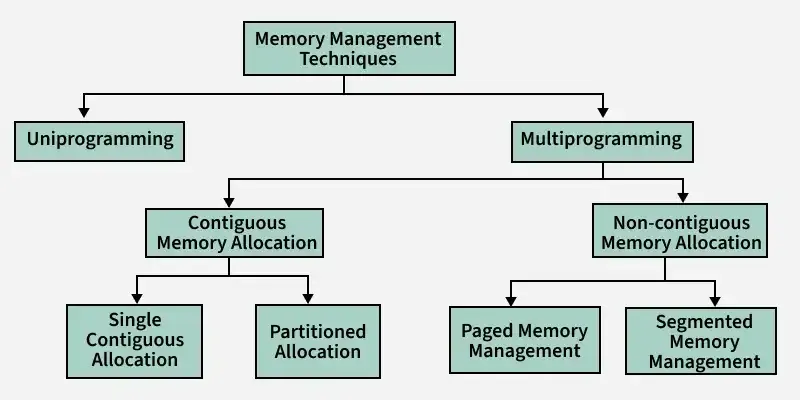
Static loading is a method where the entire process is loaded into the main memory by the loader.

1. Dynamic Loading:

Dynamic loading is a method where certain parts of a program (like functions or modules) are not loaded into memory initially. Instead, they are loaded into memory only when they are needed (i.e. during runtime).

## Memory Management Techniques:

Memory management techniques are methods used by an operating system to efficiently allocate, utilize, and manage memory resources for processes. These techniques ensure smooth execution of programs and optimal use of system memory



Memory Management Unit is crucial hardware component in modern computer systems, which is responsible for managing how processing unit interacts with the main memory.

Key roles and functions of MMU:

* Virtual to Physical Address Translation (Address Relocation)
* Memory Protection: The MMU enforces access permissions for different memory regions. It ensures that a program can only access the memory it is authorized to use.
* Virtual Memory Support: The MMU is fundamental to the implementation of Virtual Memory.

# Memory Allocator

A memory allocator is a component of a programming language or runtime system that is responsible for managing the allocation and deallocation of memory during program execution for tensors, model parameters, and intermediate computations. It handles memory requests for weights, activations, and gradients during model execution, optimizing for low latency and high throughput.

Memory Allocator sits between two entities, downward interface (towards accelerator) and upward interface (towards runtime components). The downward interface provides calls to the accelerator hardware specific APIs to request or release memory from specific regions like HBM, weight memory, input memory, etc. Whereas the upward interface (data loader, graph executer/ model runtime, kernel launcher/ op executer, etc.) interacts with the memory allocator.

The primary purpose of a memory allocator is to provide a mechanism for allocating memory blocks of various sizes to fulfill the memory requirements of a program. When a program requests memory, the allocator locates a suitable free block of memory and returns a pointer to that block. The program can then use the allocated memory for storing data.

Memory allocators typically work with a lower-level memory management system, such as the operating system’s memory manager or a heap manager, to obtain larger blocks of memory from the system and divide them into smaller allocations requested by the program. The allocator keeps track of which parts of the allocated memory are in use and which parts are free or available for reuse.

Memory allocators can have different strategies for managing memory, such as first-fit, best-fit, or buddy allocation etc. These strategies determine how the allocator searches for free memory blocks and selects the most appropriate one for a given allocation request. The choice of allocator can have a significant impact on the program’s performance, especially in memory-intensive applications.

Types of Allocator: Stack Allocator, Heap Allocator, Buddy Allocator, Pool Allocator, Slab Allocator, Region Based Allocator, Object Pool Allocator, TC Malloc

# Hierarchical Allocator

A Hierarchical Memory Allocator is a memory management strategy where memory is organized and managed in a tree-like hierarchy — meaning allocations are grouped in parent-child relationships.

Memory is allocated in groups where each group (parent) can have subgroups (child). By deleting the parent group all its children are deleted.

# Memory

## HBM (High Bandwidth Memory):

* **Role**: Acts as a large, fast, external DRAM used for bulk data storage and transfer.
* **Purpose**: Stores complete models, large feature maps, datasets, or intermediate tensors.

Acts as a global memory reservoir for weights, input data, or output activations when they exceed the capacity of on-chip memories (Input, Output, or Weight Memory). HBM is optimized for high-throughput data transfers, critical for feeding the compute unit during large-scale matrix operations or during data-intensive phases of neural network processing.

* **Access Pattern**: Slower than on-chip memory but much larger. Frequently accessed data is copied to on-chip memories.
* **Use Case**: If the model doesn’t fit entirely on-chip, weights or activations are streamed in/out via HBM.
* **Usage Example**: Stores the full set of model weights for a deep neural network or large input feature maps before they are partitioned and moved to on-chip memories for processing.

## Input Memory:

* **Role**: Dedicated on-chip memory for storing input data (e.g., input feature maps, tensors, or batches of data) that the compute unit will process in the current computation cycle.
* **Purpose**: Provides fast, low-latency access to input data to keep the compute unit fed with data for operations like convolutions or matrix multiplications. It minimizes the need to fetch data repeatedly from slower HBM or external storage.
* **Access Pattern**: Read-mostly; accessed heavily at the beginning of layers (e.g., convolution).
* **Use Case**: A new input batch is loaded into this memory, which is then read by the compute unit.
* **Usage Example:** Holds a batch of image data or a subset of a feature map during a convolutional neural network (CNN) layer computation.

## Output Memory:

* **Role**: Temporarily stores results of a layer or operation.
* **Purpose**: Temporarily holds computation outputs before they are either reused (e.g., as inputs for the next layer in a neural network) or transferred to HBM for longer-term storage or further processing. It ensures efficient data handling by keeping outputs close to the compute unit.
* **Access Pattern**: Write-mostly during a layer's execution, read later by the next stage.
* **Use Case**: After conv or matmul, results go here. Output may later be written to HBM or go to next layer.

## Weight Memory:

* Role:On-chip memory dedicated to storing model parameters (weights and biases) used by the compute unit during neural network operations.
* Purpose: Enables fast access to weights, which are frequently reused during computations like convolutions or fully connected layers. By keeping weights on-chip, it reduces latency compared to fetching them from HBM
* Access Pattern: Read-only during inference/training execution.
* Use Case: Holds the filter weights for a convolutional layer or the weight matrix for a transformer model during inference or training.

## Scratchpad Memory:

# Memory Transaction

A **memory transaction** is a low-level operation involving reading from or writing to memory. It represents a single unit of interaction with the memory system. A transaction is often atomic meaning if any failure occurs, the entire transaction is rollbacked i.e. undone.

When your AI accelerator offloads a matrix multiplication, each tensor might trigger multiple memory transactions to:

* Fetch input tensors.
* Write output results.
* Access weights or intermediate buffers.

# Runtime Interfacing

Runtime interfacing refers to the communication between the program (or compiled binary) and the runtime system.

In custom runtimes or hardware (e.g., your own AI accelerator), **runtime interfacing** involves:

* Designing APIs to pass tensors/data.
* Defining how and when to move data to/from device memory.
* Handling synchronization and error reporting.

# .pte file

.pte file is a program file used by ExecuTorch runtime that is serialized as modified flatbuffer file with optional data segments appended.

It consists of a header- standard flatbuffer header and one optional header- executorch extended header.

So the purpose of the program file is to use the program data and segment data by properly reading this file right. And it consists of the program data and segment data apart from the two headers mentioned to access this data. Program data describes the logic and structure of the model (computational graph, metadata (shapes, data type, etc.) and segment layout info (sizes, offset of weight tensors)). Segment data is the actual binary data used by the model i.e. weight tensors, constants, buffers, etc. that is used to build or define the model.

# Executorch codebase: https://github.com/pytorch/executorch/tree/main/runtime/core