CSC290/420 Machine Learning Systems for Efficient AI
Memory and Storage in ML Programs

Sreepathi Pai

October 20, 2025

URCS

Outline

Outline

Memory Management Basics

Memory Management in ML Programs

Optimizations for ML Programs

Recommended Reading

Memory Management in Programs

- Static (compile-time) management
  - Done by the compiler for fixed-size data structures
- Dynamic memory management
  - `malloc`, `free`, in languages like C
  - In garbage-collected languages, these are done under the hood by the runtime
- `malloc` returns a pointer to an area of memory on success
  - Usually from the "heap"
- `free` returns the allocation to the heap

Multi-device Memory Management

- Implicit in `malloc` is that it allocates memory on the CPU
- However, other devices, in particular GPUs have their own memory allocators
  - e.g., CUDA has `cudaMalloc` and `cudaFree`
  - can call `cudaMalloc` on both the CPU and GPU (but don't!)
- These allocators must be used even in GC languages
- GPUs have multiple memory "regions" (constant memory, local memory, shared memory)
  - (note: CUDA nomenclature)
  - Not all of them have allocators (compiler+runtime allocated)
  - a *memory mapping* problem
- Some accelerators have no dynamic memory allocation ability
  - E.g., Google's TPU, Pixel 6 TPU

Address Spaces

- On 64-bit machines, all addresses lie in a single address space
- The pointer from `malloc` isn't any different from `cudaMalloc`
- However, dereferencing a pointer on the wrong device can cause problems
  - segmentation fault
- Even CUDA's different memory regions use a unified pointer space
- May not be the case with accelerators
  - but C++ rules require unified address space

Characteristics of a Memory Allocation

- Size in bytes
- Scope
  - Local, Global
- Lifetime (or Lifespan, or Live Range)
  - from allocation to free
  - or from first user to last
- Type of accesses to data
  - Read/Write
  - Mostly read
- Source of data
  - from computation
  - from file

How Modern Dynamic Memory Allocators Work

- Allocate a global pool of memory from the system
- Divide the global pool in to per-thread pools
  - Leave some memory in the global pool
- Divide the per-thread pool into “buckets”
  - Each bucket satisfies one size of request (usually rounded up to power of 2)
  - Limited number of sizes supported
- Satisfy allocation requests from per-thread pools
  - Borrow from the global pool if running out
  - Or, for very large sizes, directly obtain memory from operating system
- Examples: hoard, jemalloc, tcmalloc

Large Allocations

- If the heap is full, the heap size can be increased using
  - `sbrk` (traditional)
  - `mmap` with anonymous pages
- Both ask the operating system for more memory
  - Involve a relatively slow system call
- Used by memory allocators for very large size allocations

Trying Out New Memory Allocators

- On Linux, the standard memory allocator is the one in `libc`
  - The C runtime library
- However, it is possible to override `malloc` and `free` at runtime
  - Use `LD_PRELOAD` environment variable on Linux
  - `LD_PRELOAD=mymalloc.so ./program`
- This only works if the C library is dynamically linked
  - Look for build options that allow changing the allocator

Garbage Collected Languages

- Garbage-collected languages also perform memory management
  - under the hood
- Garbage collectors can be tuned
  - out of scope for this class
- Remember, Python is a garbage-collected language

When is a Memory Allocation a Performance Issue?

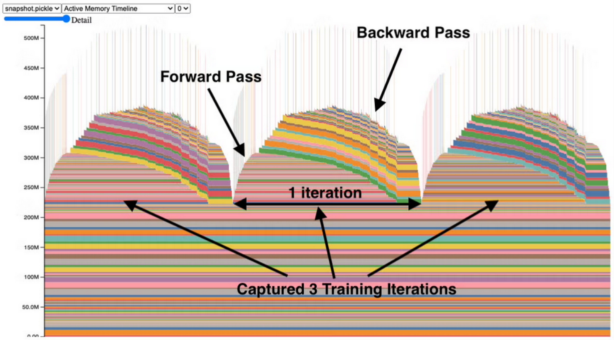
- Never
- Only under memory pressure
  - when the amount of free memory in the pool is low
  - or in the system or device
- At high rates of allocation/deallocation
  - book-keeping might consume all time
  - queueing might take up all time (e.g. `cudaMalloc` on the GPU)
- At high rates of system call invocation
- Due to poor synchronizing implementations
  - notoriously, on the GPU

Memory Allocator Debugging

- Production memory allocators can dump statistics about memory allocation
  - glibc: Allocator Debugging
  - Mostly useful for leaks and other statistics
- Valgrind also has tools for debugging memory
- Neither handles GPU memory very well

Memory Profiling

- Understanding GPU Memory 1: Visualizing All Allocations over Time



Outline

Memory Management Basics

Memory Management in ML Programs

Optimizations for ML Programs

Recommended Reading

Memory Usage

- Inputs, Outputs
- Weights (also an input, but usually read only and associated with an operator)
    - Plus biases
- Activations (also an output, but usually write only and associated with an operator)
- Temporaries (usually operator-specific)

Typical Sizes

- Usually referred to as parameters
  - Weights + Biases only
- AlexNet: 60M parameters (possibly fp32)
- GPT-3: 175B parameters (fp16)
- Deepseek-V3: 671B MoE parameters / 37B “activated parameters”

How do Memory Allocations Scale?

- Memory for inputs and outputs scale linearly with number of inputs and outputs
  - e.g., parallelism
- Weights occupy constant space (being read only)
  - but may need to be replicated across address spaces
- Activations scale linearly with parallel instances of an operator
- Temporaries also scale linearly with parallel instances of an operator
- Number of operators: size of neural network

Memory Allocation Strategy

- On demand: per-operator
  - just before operator begins execution
- Offline: per-graph
  - before execution of the graph
  - may be only option for accelerators
  - challenging for dynamic input sizes and changing parallelization

Orchestrating Memory Transfers Across Devices

- When an operator runs on a GPU, it must have space for its inputs and outputs
  - And all of its temporaries
- Then, its inputs must be transferred to it, and outputs transferred back
- In multi-device parallelism (e.g., data parallel), data shared across devices must be synchronized
  - data must be transferred across devices
  - later: Communication

Orchestrating Memory Transfers Across Machines

- When using a cluster of machines, data must be transferred across machines
- later: Communication

Micro-scale Memory Management: Placement (or Mapping)

- GPUs contain multiple memory spaces
    - constant memory: for read-only, broadcast data (e.g., convolution filters)
    - shared memory: on-chip data
- GPU compilers+runtimes will give a program a block of memory in these spaces
    - Must be populated by the CPU program (constant memory)
    - or by the GPU program/kernel (shared memory)
- Deciding *which* data resides *where* and *when* is a hard problem

Outline

Data Layout

- Certain implementations of operators expect data in a particular format
  - e.g., tensor cores like NHWC vs NCHW
- Frameworks will translate behind the scenes!
  - Extra memory and extra work

Reducing Data Size: Quantization

- Change parameters from FP32 to FP16
  - halves memory usage
- Change parameters from FP32 to INT8
  - 1/4th the memory usage!
- Weight quantization (static)
- Activation quantization (dynamic)
- Many schemes for quantization exist

Reducing Data Size: Sparsification

- Weight and bias tensors mostly contain zeroes
- Different sparsification techniques exist
- Use sparse tensor formats (COO, etc.)
- Semi-structured Sparsity
  - Zeroes follow a predictable pattern

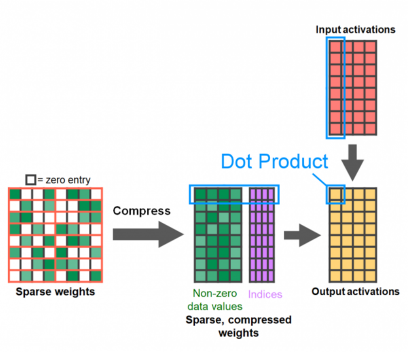


Image source: `https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt`

Reducing Data Size: Compression

- Lossless compression
  - Entropy-based schemes
- Lossy compression
  - e.g. quantization
- See, for example, QMoE: Sub-1-Bit Compression of Trillion-Parameter Models

Loading Weights Fast

- Loading weights from file is slow for large models
  - max tens of GB/s
- Example: PyTorch uses Pickle files for weights
  - also adds parsing overhead
- Best case would be equivalent to `mmap`-ing weights directly into memory

mmap-ing data

- On virtual memory systems, storage is part of memory address space
- Use the mmap system call to map a file into memory

```
// no error handling code!
FILE *f = fopen("/path/to/weights.file", "r")
fseek(f, 0, SEEK_END);
size_t sz = ftell(f)

unsigned char *contents = mmap(NULL, sz, PROT_READ,
                               MAP_SHARED, fileno(f), 0);
fclose(f);

// read contents as if it is an array
```

- What issues do you see with this scheme?

Outline

Memory Management Basics

Memory Management in ML Programs

Optimizations for ML Programs

Recommended Reading

Optional Recommended Reading

- MiMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning
  - Source: `https://github.com/google/minimalloc?tab=readme-ov-file`
- TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators
- jemalloc Post-Mortem
  - a look at the life of a memory allocator