KUDU-3342: Add an implementation of the block cache on high bandwidth memory

Author: Sammy Nah (sammy.nah@intel.com)

Version 0.1 (24 Feb 2023) - First draft

Motivation

Intel Xeon CPU Max Series contain 64GB of high bandwidth (HBW) in-package memory with up to four times more memory bandwidth than DDR5[1]. The memkind library that is used for provisioning NVM block cache can also manage the allocation of HBW memory which makes introducing the new cache type relatively easier in terms of implementation[2]. The HBW block cache will provide better performance to workloads by enabling higher bandwidth than DRAM and reducing the main memory footprint required for Apache Kudu itself.

Background

Apache Kudu currently supports two implementations of block cache - DRAM and NVM. Benchmark results show that NVM block cache showed gain in throughput over DRAM-based configuration when the dataset is designed to exceed the capacity of DRAM, while fitting entirely inside the block cache on non-volatile memory. However, NVM block cache does not show performance improvement when the dataset fits entirely inside both cache types which is the typical case in real world scenarios[3].

---

[1] https://www.intel.com/content/www/us/en/high-performance-computing/hpc-high-bandwidth-memory-video.html

[2] http://memkind.github.io/memkind/man_pages/hbwmalloc.html

[3] https://blog.cloudera.com/maximizing-performance-of-apache-kudu-block-cache-with-intel-optane-dcpmm/

High-bandwidth memory based block cache will provide up to 4x higher bandwidth compared to block cache on DDR5 memory regardless of dataset size. A proof-of-concept patch of the high-bandwidth memory block cache shows that most of the required code exists in Apache Kudu that the implementation is low hanging fruit[4]. Majority of the design considerations would be focused on refactoring and minimizing redundant code.

Goals

- Introduce a new block cache type named HBW using HBW_POLICY_BIND_ALL memory allocation policy
- Share common code between NVM and HBW block cache implementations

Non-Goals

- Not considering NUMA optimization nor DRAM fallback policies provided by memkind hbwmalloc API

Design

HBW block cache

We will add a new block cache type to the --block_cache_type option named “HBW”. The tserver dlopen memkind and register three native functions.

- hbw_set_policy()
- hbw_check_available()
- hbw_malloc()
- hbw_free()
- hbw_malloc_usable_size()

Then we will set the memkind policy to HBW_POLICY_BIND_ALL and call hbw_check_available() to check if high-bandwidth memory is recognized by the OS. Exit process if ENODEV is returned. The functions hbw_malloc(), hbw_free(), and hbw_malloc_usable_size() will be called similar to the NVM block cache implementation.

Code Refactoring

Option1

Create an interface by relocating most of the code in nvm_cache.cc to memkind_cache.cc and consolidate the functionalities related with the memkind library. The InitMemkindOps() function

[4] https://gerrit.cloudera.org/#/c/19498/

can be shared between the NVM and HBW cache implementations by branching on which native function to load according to the --block_cache_type value. Likewise, the NvmLRUCache and ShardedLRUCache can be reused by calling the necessary memkind functions according to the configured block cache type as shown in the proof-of-concept patch.

Option2

Use Abstract Factory Pattern to create interface MemkindLRUCache based on NvmLURCache and have NvmLRUCache and HbwLRUCache inherit from it. The same refactoring needed on ShardedLRUCache and create concrete classes ShardedNvmLRUCache and ShardedHbwLRUCache from it.