pynvm Documentation

Release 0.2

Christian S. Perone

Dec 29, 2017

Contents

This framework aims to bring newer NVM (*non-volatile memory*)/SCM (*storage-class memory*) technology functionalty to Python ecosystem. This project contains mainly the Python bindings for some of the libraries present in the the excellent NVM Library together with modifications to make it Pythonic and easy to use without modifying the Python interpreter itself.

Contents:

CHAPTER 1

What's new ?

1.1 Version v.0.2

Changes in this version:

- Changes to mirror the nvml API changes regarding the pmem_map_file().
- Tests updates.

CHAPTER 2

Introduction

This library provides the unofficial Python bindings for the NVM Library. The bindings were created using the Python CFFI package (*C Foreign Function Interface for Python*).

2.1 Overview and Rationale

Currently, there are no Python packages supporting *persistent memory*, where by *persistent memory* we mean memory that is accessed like volatile memory, using processor **load and store** instructions but retaining its contents across power loss just like traditional storages.

The goal of this project is to provide Python bindings for the libraries part of the NVM Library. The **pynvml** project aims to create bindings for the NVM Library without modifying the Python interpreter itself, thus making it compatible to a wide range of Python interpreters (including PyPy).

These bindings were created using the Python CFFI package (*C Foreign Function Interface for Python*).

Note: This **is not** an official port of the NVM Library.

2.2 How it works

In the image above, we can see different types of access to a NVDIMM device. There are the standard and well known types of access like the one using the standard file API (fopen/open, etc.), and the type of access that we're really interested which is the one on the right using Load/Store and bypassing the Kernel space code. This is the shortest kind of access an application can do to access the memory, and in our case, this is not only a traditional kind of volatile memory, it is a **persistent memory**, and this is why it is so important, because you don't need to serialize data to disk anymore, you just need to keep your data structures in memory, and now this data is also persistent. However with great powers comes great responsabilities, now it is duty of your application to provide things such as flushes and hardware drains (i.e. CLWB/PCOMMIT instructions), that is where this framework and Intel's NVM Library comes in.

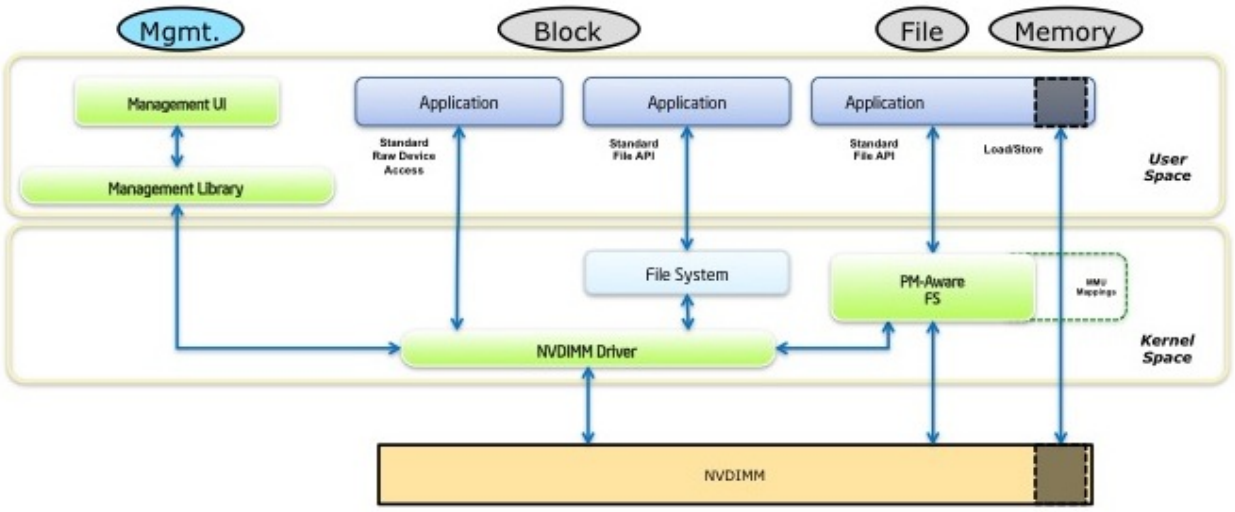

Fig. 2.1: *Image from: http://pmem.io*

See also:

Planning the Next Decade of NVM Programming.

Programming Models for Emerging Non-Volatile Memory Technologies.

Persistent Memory Byte-Addressable Non-Volatile Memory.

Persistent Memory: What's Done, Coming Soon, Expected Long-term.

2.3 Installation and Requirements

To install **pynvm**, you'll need to meet some requirements:

- NVM Library (install instructions at Github)

After installing the requirements, you'll just need to install the **pynvm** from the Python PyPI repositories using **pip**:

```
pip install pynvm
```

pip will automatically install all dependencies for the Python package and then you should be able to use the package.

CHAPTER 3

Getting Started

In this section you'll find tutorials on how to use each library supported by the **pynvm** framework.

3.1 Using pmem (*low level persistent memory*)

The pmem module exposes a "pythonic" interface to the *nvm.pmem* API, which provides low level persistent memory support.

See also:

For more information regarding the **libpmem**, please refer to libpmem manual or to the documentation on the API itself at *nvm.pmem*.

3.1.1 Opening files, writting and reading

You can see an example below on how to use the pmem API:

```
import os
from nvm import pmem
from fallocate import posix_fallocate

# (optional) check the pmem library version
pmem.check_version(1, 0)

# Open file to write and fallocate space
fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# mmap it using pmem
reg = pmem.map(fhandle, 4096)

# Write on it and seek to position zero
reg.write("lol" * 10)
```

```
reg.write("aaaa")
reg.seek(0)

# Read what was written
print reg.read(10)
print reg.read(10)

# Persist the data into the persistent memory
# (flush and hardware drain)
pmem.persist(reg)
```

3.1.2 Context managers for flush and drain and numpy buffers

You can also use context managers present in the API like the *FlushContext* or the *DrainContext*:

```
import os
import numpy as np
from nvm import pmem
from fallocate import posix_fallocate

fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# Will persist (pmem_persist) and unmap
# automatically
with pmem.map(fhandle, 4096) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

    # This will create a numpy array located at
    # persistent memory (very cool indeed) where you
    # can reshape as you like
    n = np.frombuffer(reg.buffer, dtype=np.int32)
    print n.shape

# Flush context will only flush processor caches, useful
# in cases where you want to flush several discontiguous ranges
# and then run hardware drain only once
m = pmem.map(fhandle, 4096)
with pmem.FlushContext(m) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

# Will only execute the hardware drain (aka PCOMMIT)
m = pmem.map(fhandle, 4096)
with pmem.DrainContext(m) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

fhandle.close()
```

3.2 Using pmemlog (*pmem-resident log file*)

The pmemlog module exposes a "pythonic" interface to the *nvm.pmemlog* API, which provides pmem-resident log (*append-only*) file memory support.

See also:

For more information regarding the **libpmemlog**, please refer to libpmemlog manual or to the documentation on the API itself at *nvm.pmemlog*.

3.2.1 Creating log pool and appending into it

You can see an example below on how to use the *nvm.pmemlog* API:

```
from nvm import pmemlog

# Create the logging and print the size (default is 2MB when not
# specified)
log = pmemlog.create("mylogging.pmemlog")
print log.nbyte()

# Append to the log
log.append("persistent logging!")

# Walk over the log (you can also specify chunk sizes)
def take_walk(data):
    print "Data:", data
    return 1

log.walk(take_walk)
# This will show: "Data: persistent logging!"

# Close the log pool
log.close()
```

3.3 Using pmemblk (*arrays of pmem-resident blocks*)

The pmemblk module exposes a "pythonic" interface to the *nvm.pmemblk* API, which provides arrays of pmem-resident blocks support.

See also:

For more information regarding the **libpmemblk**, please refer to libpmemblk manual or to the documentation on the API itself at *nvm.pmemblk*.

3.3.1 Creating block pool and writing into the blocks

You can see an example below on how to use the *nvm.pmemblk* API:

```
from nvm import pmemblk

# This will create a block pool with block size of 256 and
# 1GB pool
blockpool = pmemblk.create("happy_blocks.pmemblk", 256, 1<<30)
```

```
# Print the number of blocks available
print blockpool.nblock()

# Write into the 20th block
blockpool.write("persistent block!", 20)

# Read the block 20 back
data = blockpool.read(20)
blockpool.close()

# Reopen the blockpool and print 20th block
blockpool = pmemblk.open("happy_blocks.pmemblk")
print blockpool.read(20)

blockpool.close()
```

CHAPTER 4

Examples

Warning: Under Construction.

CHAPTER 5

API Documentation

All modules listed below are under the "nvm" module.

5.1 pmem – low level persistent memory support

See also:

NVML libpmem documentation.

class nvm.pmem.**DrainContext**(*memory_buffer*, *unmap=True*)
A context manager that will automatically drain the specified memory buffer.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**FILE_CREATE = 1**
Create the named file if it does not exist.

nvm.pmem.**FILE_EXCL = 2**
Ensure that this call creates the file.

nvm.pmem.**FILE_SPARSE = 4**
When creating a file, create a sparse (holey) file instead of calling posix_fallocate(2)

nvm.pmem.**FILE_TMPFILE = 8**
Create a mapping for an unnamed temporary file.

class nvm.pmem.**FlushContext**(*memory_buffer*, *unmap=True*)
A context manager that will automatically flush the specified memory buffer.

Parameters **memory_buffer** – the MemoryBuffer object.

class nvm.pmem.**MemoryBuffer**(*buffer_*, *is_pmem*, *mapped_len*)
A file-like I/O (similar to cStringIO) for persistent mmap'd regions.

read(*size=0*)
Read data from the buffer.

Parameters **size** – size to read, zero equals to entire buffer size.

Returns data read.

seek(*pos*)
Moves the cursor position in the buffer.

Parameters **pos** – the new cursor position

write(*data*)
Write data into the buffer.

Parameters **data** – data to write into the buffer.

nvm.pmem.**check_version**(*major_required*, *minor_required*)
Checks the libpmem version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

nvm.pmem.**drain**(*memory_buffer*)
Wait for any PM stores to drain from HW buffers.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**flush**(*memory_buffer*)
Flush processor cache for the given memory region.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**has_hw_drain**()
This function returns true if the machine supports the hardware drain function for persistent memory, such as that provided by the PCOMMIT instruction on Intel processors.

Returns return True if it has hardware drain, False otherwise.

nvm.pmem.**is_pmem**(*memory_buffer*)
Return true if entire range is persistent memory.

Returns True if the entire range is persistent memory, False otherwise.

nvm.pmem.**map_file**(*file_name*, *file_size*, *flags*, *mode*)
Given a path, this function creates a new read/write mapping for the named file. It will map the file using mmap, but it also takes extra steps to make large page mappings more likely.

If creation flags are not supplied, then this function creates a mapping for an existing file. In such case, *file_size* should be zero. The entire file is mapped to memory; its length is used as the length of the mapping.

See also:

NVML libpmem documentation.

Parameters

- **file_name** – The file name to use.
- **file_size** – the size to allocate
- **flags** – The flags argument can be 0 or bitwise OR of one or more of the following file creation flags: *FILE_CREATE*, *FILE_EXCL*, *FILE_TMPFILE*, *FILE_SPARSE*.

Returns The mapping, an exception will rise in case of error.

nvm.pmem.**msync**(*memory_buffer*)
Flush to persistence via *msync()*.

Parameters **memory_buffer** – the MemoryBuffer object.

Returns the msync() return result, in case of msync() error, an exception will rise.

nvm.pmem.**persist**(*memory_buffer*)
Make any cached changes to a range of pmem persistent.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**unmap**(*memory_buffer*)
Unmap the specified region.

Parameters **memory_buffer** – the MemoryBuffer object.

5.2 pmemlog – pmem-resident log file

See also:

NVML libpmemlog documentation.

class nvm.pmemlog.**LogPool**(*log_pool*)
This class represents the Log Pool opened or created using *create()* or *open()*.

append(*buf*)
This method appends from buffer to the current write offset in the log memory pool plp. Calling this function is analogous to appending to a file. The append is atomic and cannot be torn by a program failure or system crash.

On success, zero is returned. On error, -1 is returned and errno is set.

close()
This method closes the memory pool. The log memory pool itself lives on in the file that contains it and may be re-opened at a later time using *open()*.

nbyte()
This method returns the amount of usable space in the log pool. This method may be used to determine how much usable space is available after libpmemlog has added its metadata to the memory pool.

Note: You can also use *len()* to get the usable space.

Returns amount of usable space in the log pool.

rewind()
This method resets the current write point for the log to zero. After this call, the next append adds to the beginning of the log.

tell()
This method returns the current write point for the log, expressed as a byte offset into the usable log space in the memory pool. This offset starts off as zero on a newly-created log, and is incremented by each successful append operation. This function can be used to determine how much data is currently in the log.

Returns the current write point for the log, expressed as a byte offset.

walk (*func*, *chunk_size=0*)
This function walks through the log pool, from beginning to end, calling the callback function for each chunksize block of data found. The chunksize argument is useful for logs with fixed-length records and may be specified as 0 to cause a single call to the callback with the entire log contents.

Parameters

- **chunk_size** – chunk size or 0 for total length (default to 0).
- **func** – the callback function, should return 1 if it should continue walking through the log, or 0 to terminate the walk.

nvm.pmemlog.**check** (*filename*)
This method performs a consistency check of the file indicated and returns *True* if the memory pool is found to be consistent. Any inconsistencies found will cause this function to return False, in which case the use of the file with libpmemlog will result in undefined behavior.

Returns True if memory pool is consistent, False otherwise.

nvm.pmemlog.**check_version** (*major_required*, *minor_required*)
Checks the libpmemlog version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

nvm.pmemlog.**create** (*filename*, *pool_size=2097152*, *mode=438*)
The *create()* function creates a log memory pool with the given total *pool_size*. Since the transactional nature of a log memory pool requires some space overhead in the memory pool, the resulting available log size is less than poolsize, and is made available to the caller via the *nbyte()* function.

Note: If the error prevents any of the pool set files from being created, this function will raise an exception.

Parameters

- **filename** – specifies the name of the memory pool file to be created.
- **pool_size** – the size of the pool (default to 2MB).
- **mode** – specifies the permissions to use when creating the file.

Returns the new log memory pool created.

Return type *LogPool*

nvm.pmemlog.**open** (*filename*)
This function opens an existing log memory pool, returning a memory pool.

Note: If an error prevents the pool from being opened, this function will rise an exception.

Parameters **filename** – Filename must be an existing file containing a log memory pool as created by the *create()* method. The application must have permission to open the file and memory map it with read/write permissions.

Returns the log memory pool.

Return type *LogPool*

5.3 pmemblk – arrays of pmem-resident blocks

See also:

NVML libpmemblk documentation.

class nvm.pmemblk.**BlockPool**(*block_pool*)

This class represents the Block Pool opened or created using *create()* or *open()*.

bsize()

This method returns the block size of the specified block memory pool. It's the value which was passed as block size to *create()*.

Returns the block size.

close()

This method closes the memory pool. The block memory pool itself lives on in the file that contains it and may be re-opened at a later time using *open()*.

nblock()

This method returns the usable space in the block memory pool, expressed as the number of blocks available.

Returns usable space in block memory pool in number of blocks.

read(*block_num*)

This method reads a block from memory pool at specified block number.

Note: Reading a block that has never been written will return an empty buffer.

Returns data at block.

set_error(*block_num*)

This method sets the error state for block number blockno in memory pool. A block in the error state returns errno EIO when read. Writing the block clears the error state and returns the block to normal use.

Returns On success, zero is returned. On error, an exception will be raised.

set_zero(*block_num*)

This method writes zeros to block number blockno in memory pool. Using this function is faster than actually writing a block of zeros since libpmemblk uses metadata to indicate the block should read back as zero.

Returns On success, zero is returned. On error, an exception will be raised.

write(*data*, *block_num*)

This method writes a block from data to block number blockno in the memory pool. The write is atomic with respect to other reads and writes. In addition, the write cannot be torn by program failure or system crash; on recovery the block is guaranteed to contain either the old data or the new data, never a mixture of both.

Returns On success, zero is returned. On error, an exception will be raised.

nvm.pmemblk.**check**(*filename*, *block_size=0*)

This function performs a consistency check of the file indicated by path and returns True if the memory pool is found to be consistent. Any inconsistencies found will cause it to return False, in which case the use of the file with libpmemblk will result in undefined behavior.

Note: When block size is non-zero, it will compare it to the block size of the pool and return False when they don't match.

Returns True if memory pool is consistent, False otherwise.

nvm.pmemblk.**check_version**(*major_required*, *minor_required*)

Checks the libpmemblk version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

nvm.pmemblk.**create**(*filename*, *block_size*, *pool_size=2097152*, *mode=438*)

This function function creates a block memory pool with the given total pool size divided up into as many elements of block size as will fit in the pool.

Note: Since the transactional nature of a block memory pool requires some space overhead in the memory pool, the resulting number of available blocks is less than poolsize / block size, and is made available to the caller via the *nblock()*.

If the error prevents any of the pool set files from being created, this function will raise an exception.

Parameters

- **filename** – specifies the name of the memory pool file to be created.
- **block_size** – the size of the blocks.
- **pool_size** – the size of the pool (default to 2MB).
- **mode** – specifies the permissions to use when creating the file.

Returns the new block memory pool created.

Return type *BlockPool*

nvm.pmemblk.**open**(*filename*, *block_size=0*)

This function opens an existing block memory pool, returning a memory pool.

Note: If an error prevents the pool from being opened, this function will rise an exception. If the block size provided is non-zero, it will verify the given block size matches the block size used when the pool was created. Otherwise, it will open the pool without verification of the block size.

Parameters **filename** – Filename must be an existing file containing a block memory pool as created by the *create()* method. The application must have permission to open the file and memory map it with read/write permissions.

Returns the block memory pool.

Return type *BlockPool*

CHAPTER 6

License

```
Copyright 2016 HP Development Company, L.P.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the <organization> nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Note: This library links with the NVML, you can see the NVML license here.

Note: This framework is in active development and it is still in beta release.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

p

Index

W