DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access

Pak Markthub*, Mehmet E. Belviranli†, Seyong Lee†, Jeffrey S. Vetter† and Satoshi Matsuoka‡*
*Department of Mathematical and Computing Sciences
Tokyo Institute of Technology, Tokyo, Japan
Email: markthub.p.aa@m.titech.ac.jp
†Future Technologies Group
Oak Ridge National Laboratory, Tennessee, USA
Email: {belviranlime, lees2, vetter}@ornl.gov
‡RIKEN Center for Computational Science, Kobe, Japan
Email: matsu@is.titech.ac.jp

***Abstract*—Heterogeneous computing with accelerators is growing in importance in high performance computing (HPC). Recently, application datasets have expanded beyond the memory capacity of these accelerators, and often beyond the capacity of their hosts. Meanwhile, nonvolatile memory (NVM) storage has emerged as a pervasive component in HPC systems because NVM provides massive amounts of memory capacity at affordable cost. Currently, for accelerator applications to use NVM, they must manually orchestrate data movement across multiple memories and this approach only performs well for applications with simple access behaviors. To address this issue, we developed *DRAGON*, a solution that enables all classes of GP-GPU applications to transparently compute on terabyte datasets residing in NVM. *DRAGON* leverages the page-faulting mechanism on the recent NVIDIA GPUs by extending capabilities of CUDA Unified Memory (UM). Our experimental results show that DRAGON transparently expands memory capacity and obtain additional speedups via automated I/O and data transfer overlapping.**

***Index Terms*—gpu, out-of-core, memory, driver, large data**

I. Introduction

Three important trends are emerging in high performance computing (HPC). First, heterogeneous computing with accelerators, such as GPUs and FPGAs, is growing in importance in HPC, machine learning, and other areas. Typically, these accelerators have their own high performance memory, which is discrete from and smaller than the host memory. Second, application datasets have grown much larger than the capacity of accelerator memory and even beyond that provided by host memory. The local data processed in scientific computations can easily exceed the host memory capacity [1], [2]; therefore, even with the use of NVIDIA's state-of-the-art Unified Memory, such applications would still need to rely on techniques like *manual buffer management* for *large memory* support (i.e., data larger than the system memory). Third, nonvolatile memory (NVM) storage, such as 3D-NAND flash, has emerged as a technology to provide massive amounts of memory capacity to a node with good power efficiency and rapidly increasing bandwidths at affordable costs [3]–[5]. With the recent introduction of 3D-Xpoint technology from Intel [6], NVM technologies are progressively competitive with expensive and density-limited DDR-based host memory.

These trends are evident in upcoming supercomputers, like ORNL's Summit and LLNL's Sierra, which will employ both NVM devices and several GPUs per node. Ideally, in these systems, the accelerators should be able to access massive NVM storage with both reasonable performance and minor impact on the programmability of their applications. The community has explored multiple software and hardware approaches for accelerator kernels to better exploit host memory, but not for NVM storage [7].

A traditional approach is to decompose the data into smaller chunks and employ two-way transfers between corresponding kernel executions [2]. While this approach fits well for streaming types of applications, it may require extensive application-specific programming effort and may not be applicable if the data access pattern is irregular or unsuitable for the computation to be partitioned into multiple kernel launches. Alternatively, hardware, OS-level, and application-based solutions have been proposed to process data larger than the GPU memory by utilizing the host-side DRAM as well (see § VI). However, they fall short of being practical for several reasons, such as requiring hardware modifications, having considerable overhead, or being limited to a specific class of applications.

More recently, the revised *Unified Memory (UM)* for the NVIDIA Pascal architecture [8], which we will refer to as *UM-P*, offers a native solution for accessing host memory from within GPU kernels. UM-P relies on native device-initiated page-faulting and driver-managed swapping mechanisms to present a unified virtual memory address range for accesses from both CPU and GPU. Although UM-P may provide faster performance than its predecessor [9], the amount of virtual memory that can be allocated by UM-P is still limited by the available physical host memory [10], [11].

I-A. Contributions

To address this challenge, we have developed *DRAGON*: a solution that enables all classes of General purpose GPU (GP-GPU) applications to *transparently* operate on very large datasets residing in NVM while also ensuring the integrity of data buffers, which is important for persistent data stored in NVM. *DRAGON* leverages the page-faulting mechanism on the recent NVIDIA GPUs and extends capabilities of CUDA UM-P to provide transparent data access to terabytes of NVM.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

More specifically, we make the following contributions in this paper:

- We design and implement a novel approach that transparently maps the memory space addressable by the GPU device code directly to NVM devices, with effectively no limits on capacity.
- We uniquely eliminate the need for manual buffer management for GPU kernels running on data larger than the GPU or host memory. We also present NVM-optimized access pattern types for read-only, write-only, and temporary data to decrease I/O overheads.
- We evaluate *DRAGON* on a set of scientific kernels on a NVIDIA P100 GPU and a 2.4 TB Micron 9100 NVMe card, demonstrate that the I/O overhead is hidden in most cases, and show that extra speedup is obtained against the original UM-P by utilizing Linux's page-caching mechanism for streamlined I/O operations.
- We also evaluate *DRAGON* with two popular deep learning (DL) workloads in Caffe [12] to demonstrate the feasibility of real-life scenarios where access to large datasets is a critical requirement.

II. Background and Motivation

Nonvolatile memory (NVM) technology is rapidly evolving. As their characteristics improve, NVMs are migrating up the memory hierarchy and becoming a viable alternative for system memory [3], where application data structures with sizes larger than the host's DRAM can reside. Early devices were straightforward replacements for magnetic hard disks, but upcoming options include NVMs [4], [6] that may be inserted directly into DIMM slots (i.e., NVDIMMs), providing considerably higher performance.

GP-GPU computing, on the other hand, has become the primary choice for a wide range of domains which vary from HPC to DL. Most GP-GPU programming models were originally developed around the assumption that the size of the data footprint of the problem is smaller than the GPU memory [11], [13]. GP-GPU problem sizes have grown to the point where we cannot simultaneously store all application data on the GPU or host memory. As GPUs get faster, programming models have evolved to embed mechanisms that support off-chip memory accesses. Meanwhile, application developers and researchers have developed algorithm-specific techniques [1], [2], [14]–[19] to enable the processing of *large data* on GPUs.

For HPC applications using GPUs, the large and high-throughput memory space provided by NVMs can provide an efficient solution to alleviate memory space restrictions and reduce data-staging overhead. Possible use cases include but are not limited to the following.

- Multi-GPU-based systems: Increased number of GPUs in the newest top-supercomputers (e.g., ORNL's Summit and LLNL's Sierra) requires a larger amount of data to be read from/into storage and also longer data-transfer times to/from local GPUs [20]. Direct and efficient access to local or shared (i.e., Lustre) NVM storage directly from GPUs will allow developers to effortlessly port their applications to fat-node-based systems.
- Workflow schemes: Multiple data-staging steps are needed where the output of one computational stage is serialized into the storage first, so that the next step can read the data [21]. In such cases (e.g., in situ visualization), using node-local NVMs as the primary memory location will minimize the data movement and also allow full access to the processed data on the later steps of the workflow.
- Deep learning: *Model-parallelism* [22] attempts to distribute network layers across multiple GPUs to fit models that require very large in-memory representation. However, even model-parallelism will fail when the input or output size of a single layer exceeds the GPU memory. For example, a CT-scan image's size can easily reach terabytes [23] and the only existing solution is to scale the image down until it fits into the GPU or host memory [24]. GPU-accessible NVMs will provide much larger capacity at lower costs.

Utilizing NVMs as large-memory addressable devices in GPU applications presents three major challenges.

Direct and transparent addressing: To integrate NVMs without increasing the complexity of kernels, GPUs should be able to directly address the very large space provided by NVMs with no or insignificant performance penalties. Tseng et al. [25] and Zhang et al. [26] proposed utilizing a custom NVMe driver to directly transfer data between GPU memory and NVM using NVIDIA's GPUDirect RDMA [27]. While these studies provide efficient means for direct NVM access for GPUs, GPUDirect RDMA technically limits the addressable space to the GPU's global memory size [27].

Low-latency access: NVIDIA's Unified Memory for the Pascal architecture (UM-P), introduced with CUDA 8, opens up a new opportunity for mapping larger addressable ranges with less overhead. UM-P relies on a hardware-based page-faulting mechanism to transparently address out-of-core memory requests. However, the addressable range is limited by the amount of physically available host memory [10], [11]. To support large data, applications still need to manually manage data movement between NVMs and the host memory. Also, without dynamic techniques to overlap data staging and PCIe transfers with computation, the application throughput can be even lower than expected.

Going beyond system memory capacity limits: ActivePointers [28] is the only study that considered expanding GPU addressable memory range by mapping it to a file-system. ActivePointers is based on GPUfs [29] and relies on in-kernel software address translation (i.e., SW-based page faulting). It requires existing CUDA kernels to be modified to use custom pointers and APIs. Memory references are captured on the fly via operator overloading, and a page-fault handling kernel-code is executed for every access. This approach incurs significant overhead in addition to the extensive cost of modifying existing GPU kernels; therefore, it is relatively inefficient as it does not exploit the new hardware paging support in GPUs, as demonstrated in § IV.

To the best of our knowledge, there are no general-purpose solutions that can address all of the above challenges efficiently. As a solution, we introduce *DRAGON*, a framework that allows NVMs to be directly mapped to GPUs to take full advantage

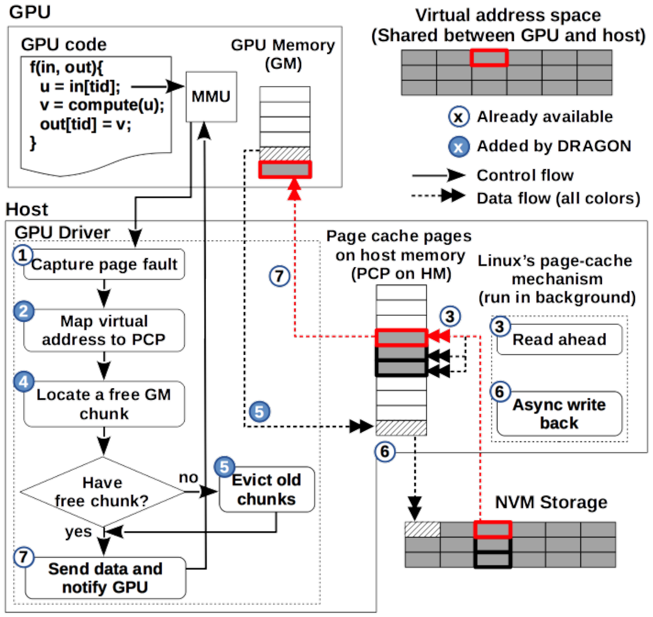


Fig. 1: *DRAGON* driver operation

of the large capacity and high bandwidth of NVM devices.

III. *DRAGON*

DRAGON, ***D****irect* ***R****esource* ***A****ccess for* ***G****PUs* ***O****ver* ***N****VM*, is a host-based framework that transparently extends the GPU addressable global memory space beyond the host memory using NVM-backed data pointers. *DRAGON* allows storing the binary memory dump of application data in a file on NVM and mapping it to the global memory space of the GPU. This enables GPU kernels to access the data via regular load/store instructions, similar to how `mmap()` operates in CPUs.

DRAGON introduces a novel driver design that incorporates NVM-specific optimizations into modern GPUs to exploit the unique benefits of the underlying heterogeneous memory hierarchy. For the proof-of-concept implementation, *DRAGON* is developed as an extension to Unified Memory for Pascal [8], and it is made accessible to users via a separate user-level host API. Our extension and modifications to the NVIDIA driver are limited to the MIT-licensed open-source `nvidia-uvm` submodule. *DRAGON* preserves all existing CUDA features without any interface change or performance penalty.

Changes required for CUDA applications to work with the *DRAGON* framework are minimal. Applications can take advantage of *DRAGON* by using the `dragon_map()` API function with a file path and other optimization parameters (explained later) to map previously dumped binary data into a unified memory space. This function internally finds an unmapped virtual address space, registers this space along with the file path to the internal tracking mechanism of our driver extension, and returns the virtual address to the application. Once mapped, the memory range is accessible by both GPUs and CPUs, and *DRAGON* transparently allows direct access down to NVM devices during GPU kernel execution.

III-A. Driver Operation

In *DRAGON*, there are three locations where data can reside in: GPU memory (GM), host memory (HM), and nonvolatile memory (NVM). The content of the file that is mapped via `dragon_map()` is visible to both CPUs and GPUs under the same unified virtual address space. NVIDIA driver for Pascal and Volta architectures captures page faults from both GPUs and CPUs, and *DRAGON* relies on this hardware-based page-faulting mechanism to handle the accesses from GPU kernels to the mapped files. The memory consistency between GM and HM is handled by the original UM-P module, whereas data swapping and consistency between HM and NVM are handled by our driver extension (as explained in detail in § III-B).

For file operations, *DRAGON* utilizes Linux's page-cache mechanism and read-ahead operations [30] to efficiently handle NVM I/O. Read-ahead is automatically triggered after every NVM file read operation, and as soon as the kernel thread finishes serving a GPU page fault, consecutive blocks from the NVM file are retrieved in the background as the GPU execution continues. Read-ahead allows *DRAGON* to implicitly overlap data-staging operations with CPU-to-GPU transfers and GPU computation, hence exploiting better performance than the default case where all operations are serially performed.

In this section, we focus on the extended driver operation proposed as part of the *DRAGON* framework. Figure 1 shows the high-level operation of *DRAGON* driver extension and explains a scenario where a page fault originated from the GPU is served by the driver. The following steps describe how *DRAGON* handles load and store operations to provide access to the corresponding data in NVM.

- **Map and Load:** ① When the driver receives the GPU page-fault signal, ❷ *DRAGON* maps the virtual address that comes along with the signal to the corresponding page-cache page (PCP). If the PCP is not physically found in host memory (HM) or is not up to date, ③ *DRAGON* invokes page-cache to fill in the data from the storage, similarly to how `mmap()` works internally [30], [31]. Then, *DRAGON* temporarily pins this PCP in HM and rejects further modification to the page by setting the `lock` bit in the Linux page structure.
- **Locate and Evict:** ❹ Next, *DRAGON* tries to locate a free memory chunk on the GPU. If there is no such chunk, ❺ *DRAGON* will evict *not-recently-used* GPU chunks to free up the necessary space. The evicted GPU data is written into the corresponding PCPs on the HM but not immediately written back to NVM. ⑥ The page-cache mechanism, which is running in the background, is responsible for writing them to the storage (write-back), as needed.
- **Transfer and Notify:** ⑦ Finally, *DRAGON* initiates the transfer from the pinned PCP to the recently freed GM chunk via DMA. The original GPU driver is responsible for the internals of this data transfer. Under specific circumstances, *DRAGON* sends data on multiple PCPs to the GPU in the same transaction, depending on the size of the found GPU chunk. Details of this operation are discussed in § III-B.

There are two mechanisms that can trigger the propagation of dirty data from GM to NVM: eviction (❺) and `dragon_sync()`.

Eviction is the main mechanism and works implicitly as explained above. `dragon_sync()`, on the other hand, explicitly triggers the eviction for all used GPU chunks, or as specified by the virtual address and size, to HM. `dragon_sync()` relies on `vfs_fsync()` to flush dirty data from PCPs to NVM. Both eviction and `dragon_sync()` invalidate the corresponding GM copies to ensure data consistency (see § III-B). This allows applications to ensure that all changes on both GM and HM are committed to NVM.

For a load/store operation originating from the CPU, ① *DRAGON* captures the page-fault signal and ② attempts to resolve the corresponding virtual GM address. The original GPU driver keeps a lookup table that indicates where each memory page is located: in GM, in HM, or not allocated. If the page is in GM, ⑤ *DRAGON* evicts the GPU chunk to the corresponding PCPs and invalidates the GM copy. Otherwise, it just locates the corresponding PCP and ensures that the PCP is up to date by reading it from the NVM, if needed. Finally, the PCP is inserted back into the process page table. Since the GM data is evicted, GPU may issue a future page-fault signal if it needs to access the evicted data again.

III-B. Data Consistency and Access Granularity

DRAGON preserves the consistency model provided by NVIDIA's original UM-P implementation [11] as long as all accesses to the mapped data on NVM are performed via the memory space returned by `dragon_map()`. Internally, *DRAGON* relies on Linux's page caching mechanism to keep NVM and HM copies in sync. All PCPs are flushed back to NVM prior to application termination or when `dragon_unmap()` is called.

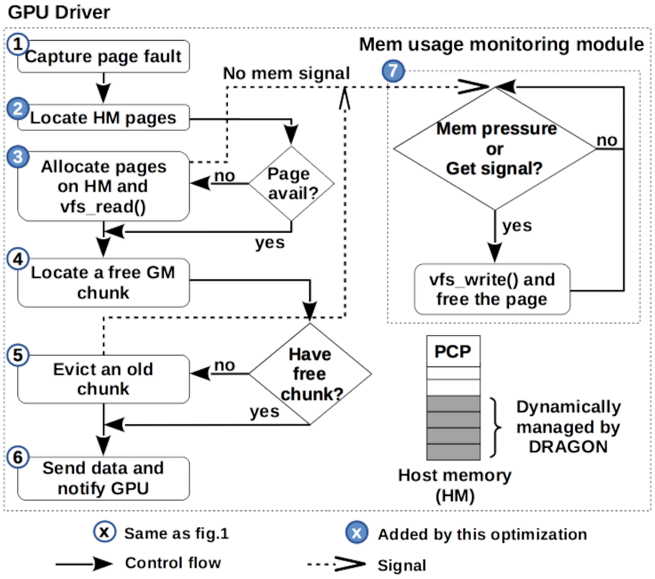
DRAGON also ensures eventual consistency [32] via `dragon_sync()`. After calling this function, all dirty data is flushed to the NVM-backed files, and all future accesses to both dirty and non-dirty data are freshly fetched from the files. This allows multiple GPU applications to use the same NVM file as source for `dragon_map()`, along with proper combination of inter-process synchronization (e.g., message tunnels) and communication primitives (e.g., `MPI_Barrier()`).

The finest data granularity that *DRAGON* uses is one page – 4 KiB on most x64 Linux kernels. For GM, NVIDIA uses three different sizes of data chunks (or pages), 4 KiB, 64 KiB, and 2 MiB, in the driver. *DRAGON* properly manages one-to-many mapping (one GPU chunk to multiple PCPs) using pointers and arrays inside the driver, and the storage overhead for this mapping is four bytes for every 2MiB.

III-C. Optimization for Input/Output Data

The operation of *DRAGON* explained so far takes only the default `dragon_map()` behavior into consideration, where an NVM-based data structure is mapped as `R_W`, forcing both read and write operations to travel all the way between GM and NVM. However, we can apply further optimizations on the data movement for some specific data access patterns.

Read-only: The input data is usually read-only, where the application uses the data but does not modify the files. If a specific file is mapped as read-only, *DRAGON* marks the evicted GPU chunks immediately as *free* and does not actually transfer the data back from GM to HM. This optimization saves redundant GPU-to-host transfers, which would otherwise be inevitable in the default UM-P based implementation.



GPU Driver
Mem usage monitoring module
1 Capture page fault
No mem signal
7
2 Locate HM pages
Mem pressure or Get signal?
no
3 Allocate pages on HM and vfs_read()
no
Page avail?
yes
yes
4 Locate a free GM chunk
vfs_write() and free the page
PCP
5 Evict an old chunk
no
Have free chunk?
yes
Dynamically managed by DRAGON
6 Send data and notify GPU
Host memory (HM)
X Same as fig.1
X Added by this optimization
Control flow
Signal

Fig. 2: *DRAGON* operation for handling intermediate data.

Write-only: For some applications, output data are write-only, and the contents of the mapped files need not be read into HM or GM on the first access. In such cases, *DRAGON* only needs to locate a free GPU chunk on a GPU-originated page fault without triggering the read-ahead mechanisms for NVM file read. Once the data is written back to the NVM and corresponding pages are swapped out, consecutive accesses to write-only data are treated as read&write. This is necessary to maintain consistency because *DRAGON* operates at single-page granularity, whereas the actual GPU and CPU instructions can modify the data in smaller granularities.

III-D. Intermediate Data Handling

Intermediate data is the memory space allocated and used by applications to pass temporary information between different phases of the computation. Even though such data are not required to directly hold input or output values of the computation, they might occupy a considerable amount of space that may be significantly larger than HM. For example, in deep neural networks (DNN), the data passed between the internal layers of the network are considered as *intermediate* since such data are needed only during the lifetime of a program, unlike the input or output data.

Applications may utilize *DRAGON* to handle large intermediate data by creating a temporary file and *mapping* it to the unified memory space using `dragon_map()`. While the basic *DRAGON* operation explained earlier in this section will accurately handle the intermediate data, applications that use an excessive amount of intermediate data may not show optimal performance with *DRAGON*. The page-cache write-back mechanism, which is controlled by the Linux kernel, periodically flushes dirty PCPs back to storage, and this behavior may unnecessarily increase I/O traffic in the existence of intermediate data. Similarly, redundantly reading such data back from the NVM may reduce total bandwidth.

```
dError_t dragon_map(const char *filename, size_t size,
     off_t offset, unsigned short flags, void **addr);
dError_t dragon_sync(void *addr, size_t size);
dError_t dragon_unmap(void *addr);
```

Listing 1: *DRAGON* user-level APIs

To address this performance problem, *DRAGON* utilizes an additional location on HM to store intermediate data, so that load and store requests from GPUs can quickly be served from this cache without invoking page-cache's write-back mechanism at all. Figure 2 explains how *DRAGON* achieves this behavior. The optimization relies on the memory usage monitoring module, MUMM, (❼), which monitors the sum of the number of free memory pages and PCPs. When this sum falls under a specific threshold, the module writes back a predetermined amount of pages to NVM using `vfs_write()` and then frees those pages. In addition to MUMM, we add another submodule to all memory allocations occurring inside the driver (❸,⑤). This submodule sends a *no-memory signal* to the MUMM when the memory allocation fails. The signal forces MUMM to free some pages before attempting the memory allocation again. Doing so prevents the driver from failing due to running out of free memory. *DRAGON* uses `vfs_read()` to read the pages requested by GPUs or CPUs back from NVM (❸). The reads are not performed if *DRAGON* finds the requested pages in HM (②). To prevent freeing pages that are being copied to the GPU, each module sets and waits for the `lock` bit of the page that it is working on. *DRAGON* also ignores `dragon_sync()` for intermediate data, since they are *volatile* (not to be stored) and internally used within the mapped regions (i.e., not shared with other processes).

The threshold for MUMM to start *evict*ing some pages to NVM is set as the sum of the total PCP and free memory sizes. *DRAGON* guarantees that the sum of those memories always reaches or exceeds the specified threshold. Thus, the amount of HM for keeping intermediate data can grow or shrink based on the amount of HM occupied by PCPs. When there is no space left for keeping intermediate data (e.g., when the user space uses up all memory), *DRAGON* changes the handling back to the original operation (fig. 1), which require no extra space. How to adjust this threshold is application and system specific. Setting this value too high (biased towards having more free memory and PCPs) reduces the space for keeping intermediate data in HM. This results in more data movement between HM and storage, which lowers *DRAGON*'s performance. On the other hand, setting the threshold too low reduces the number of PCPs, which lowers the efficiency of the page-cache mechanism. We further discuss the effect of this threshold in § IV-E.

III-E. Integrating DRAGON

DRAGON, from a user's perspective, works like Linux's `mmap()` and allows the NVM to be accessible by both the host and the GPU via the same virtual address space. *DRAGON* host API provides `dragon_map()`, as shown in listing 1. This function takes the path string to the file, its size, file start offset to be mapped, and a set of flags as its input and assigns the starting address of the corresponding unified virtual memory to the last parameter. `dragon_map()` replaces manual buffer allocations (e.g., `malloc()` and `cudaMalloc()`) and user-managed data movement (e.g., `fread()`/`fwrite()` and `cudaMemcpy()`). For legacy applications, developers can replace these operations with `dragon_map()` for the data buffers they want to manage via DRAGON. On the other hand, *DRAGON* does not have a device API; therefore, *no modification to GPU kernels is required*. Further steps on how to integrate *DRAGON* into legacy GPU applications are given in § IV-B.

`dragon_map()` accepts an additional parameter, named *flags*, to indicate the data access type. `D_READ` and `D_WRITE` correspond to the read-only and write-only optimizations, respectively (§ III-C). Combining them (the default value) tells *DRAGON* that the mapped data are for both reading and writing and thus do not apply those optimizations. The presence of `D_VOLATILE` tells *DRAGON* to apply the intermediate data optimization (§ III-D). The other two API functions, `dragon_sync()` and `dragon_unmap()`, allow applications to manually flush GM and HM contents and release all memory space occupied by the driver, respectively.

DRAGON also allows programmers to implement more advanced prefetching behavior by using `cudaMemAdvise()` and `cudaMemPrefetchAsync()`, which are provided by the original UM-P [10]. The `advice` parameter will hint the UM-P driver (and hence DRAGON) about the data access behavior. When `cudaMemPrefetchAsync()` is called, DRAGON will replicate the intended effects of `advice` for the addresses residing on NVM. This behavior can be utilized to minimize NVM read overhead for scattered data, where Linux's read-ahead will not help due to nonconsecutive access patterns.

IV. Evaluation

To experiment with our proposed driver extension and analyze its performance, we applied *DRAGON* to several applications and compared them with other alternative execution schemes. In addition, in § V, we provide a further detailed analysis of *DRAGON* with the Caffe DL framework. We tested *DRAGON* on a system containing dual 12-core Intel Xeon E5 processors, 64 GiB of DDR4 memory, an NVIDIA P100 GPU with 12 GiB of HBM connected via PCIe gen.3 x16, and a 2.4 TB Micron 9100 HHHL U.2 PCIe NVMe card connected via PCIe gen.3 x4. Our experiments were run with CentOS 7 Linux 3.10.0-693.5.2.el7.x86_64 kernel and CUDA 9.0 toolkit with NVIDIA driver version 384.81.

IV-A. Compared Execution Schemes

We compared *DRAGON*-driver-based execution with four different CUDA-based schemes and reported three of them in our evaluation.

Default: This is the standard CUDA implementation where the applications are originally designed to run with data sizes that fit in the GM. A common implementation for such applications is to copy all input data from the files to the HM using `fread()`, then transfer the data to the GM using `cudaMemcpy()`, and copy the output data back using `cudaMemcpy()` and `fwrite()` to the files.

Hostreg: This approach uses `cudaHostRegister()` with `mmap()` as an alternative method to enable GPUs to access

TABLE I: Evaluated applications

Application	Category	Vol:NonVol
backprop	Unstructured Grid	1:1
binomialOptions	Linear Algebra	0:1
BlackScholes	Linear Algebra	0:1
hotspot	Structured Grid	0:1
lavaMD	N-Body	0:1
pathfinder	Dynamic Programming	0:1
srad_v2	Structured Grid	5:1
vectorAdd	Dense Linear Algebra	0:1

the mapped file directly. While this technique uses a different internal mechanism from UM-P, the total size is still limited by the HM, as discussed in Appendix A.

UM-P: As our baseline, we used unmodified CUDA UM-P via `cudaMallocaManaged()` combined with POSIX I/O operations (e.g., `fread()`/`fwrite()`) to stage data to/from NVMs. Although this technique cannot handle data larger than the host memory (HM), it is one of the popular choices among programmers to make their applications process larger data without reimplementing the GPU kernels [9].

ActivePointers: We have integrated ActivePointers [28] into two of our benchmarks: BlackScholes and vectorAdd. The integration involved changing each kernel to first map the files via `gvmmap()` and then replacing the corresponding pointers with `ActivePtr`. Due to extensive porting effort and poor performance, we limit the evaluation of this approach to two benchmarks only and report the results separately.

IV-B. Applications

We selected eight applications from the Rodinia benchmark suite [33] and the CUDA SDK as shown in table I. We followed the steps below to integrate *DRAGON* into these applications:

1) Identify the data staging code used to read the contents of in-memory data buffers (variables) from an input file. Then, serialize (i.e., memory dump) such variables into separate files created on NVM. Once dumped, data-staging file I/O calls are no longer needed and they are removed from the code.
2) Identify device data allocations (`cudaMalloc()`) and replace them with `dragon_map()`.
    a) *Input/Output data:* Set the filename parameter for `dragon_map()` to point to the NVM file that was used to serialize the original variables, and set `D_READ` and `D_WRITE` flags properly.
    b) *Intermediate data:* Create a temporary file for each GPU-allocated data region, which was identified as intermediate data buffers in the previous step. Set the `D_VOLATILE` flag.
3) Replace the type of all indexing variables (i.e., `ints`) with `longs` in order to support addressing for very large memory spaces. Also, remove all instances of `cudaMemcpy()`, since they are no longer needed.

The first step above is for data preparation, and it is necessary because `dragon_map()` expects the file to be a *byte-to-offset* memory dump of the variable in question, similar to `mmap()`. Because some of the applications in our experiment create random input data on the fly, we also serialized such data into files and supplied them as inputs. For a fair comparison, we made all the techniques use *memory-dumped* input/output files. *It is essential to note that DRAGON did not require any changes to the existing GPU kernels or the application logic.*

Table I also includes a column for volatile vs. nonvolatile (`Vol:NonVol`) buffer ratios. These numbers correspond to the least common multiples for the sizes of the variables declared as intermediate (via `D_VOLATILE` flag) and input/output data (i.e., no `D_VOLATILE` flag) – see § III-C and III-D for more details. If an application has no intermediate data, it is marked with a zero. We refer to these ratios later in the analysis.

IV-C. Overall Performance

We ran each application listed in table I using *Default* (original implementation), *Hostreg*, *UM-P* (baseline), and *DRAGON* (this work) techniques described in § IV-A. We varied the total memory footprint (i.e., size of intermediate + input/output data) of each application between 4 GiB and 256 GiB in doubling increments. As the GM and the HM were 12 GiB and 64 GiB, respectively, our experiment covered all three memory execution cases: *incore-GPU* (4 - 8 GiB), *incore-host* (16 - 64 GiB), and *out-of-core* (128 - 256 GiB). The *Default* technique was able to cover only the *incore-GPU* case, whereas *Hostreg*- and *UM-P*-based executions successfully completed with data sizes falling in the *incore-host* range. Only *DRAGON* was able to operate on data in all three ranges, including *out-of-core*. For each run, we reset the page caches prior to application launch and measured the total execution time. We normalized these times to the total execution time measured while running with the baseline UM-P technique.

Figure 3 shows the results of this experiment. Each execution technique is represented with numbered bars, #1 to #4 (*Default*, *Hostreg*, *UM-P*, and *DRAGON*, respectively). The x-axis corresponds to the total memory footprint in log scale. The left-hand y-axis corresponds to the execution time normalized with respect to the UM-P technique. The right-hand y-axis, which is also in log scale, corresponds to the total execution time (i.e., wall-clock) of the *DRAGON* technique (Bar #4) and is represented by a solid line. For the *out-of-core case*, since only *DRAGON* can handle this range, we projected values of bar #3 (*UM-P* technique) using linear extrapolation so that we were able to present a comparison. Each bar is shaded to show the breakdown across GPU-CPU data transfers (device-to-host and host-to-device combined), kernel execution time, NVM accesses (read/write) and map/free operations. For bar #2 (*Hostreg*), the breakdown is limited to `mmap()` and `free()` operations since we did not have access to internals to measure data transfers and file accesses. For bar #3 (*UM-P*), kernel execution time also includes CPU-GPU transfers – we used the original NVIDIA GPU driver and thus could not measure CPU-GPU transfer time separately.

Overall, our experiment shows that *DRAGON* is able to *exclusively* provide direct access to *out-of-core* memory from GPUs. Additionally, the application execution time reduced by 31.74% on average and 56.27% at maximum (`pathfinder`), which corresponds to 2.3x speedup. *DRAGON* uniquely exploits the benefits of *read-ahead* behavior of the Linux page-cache

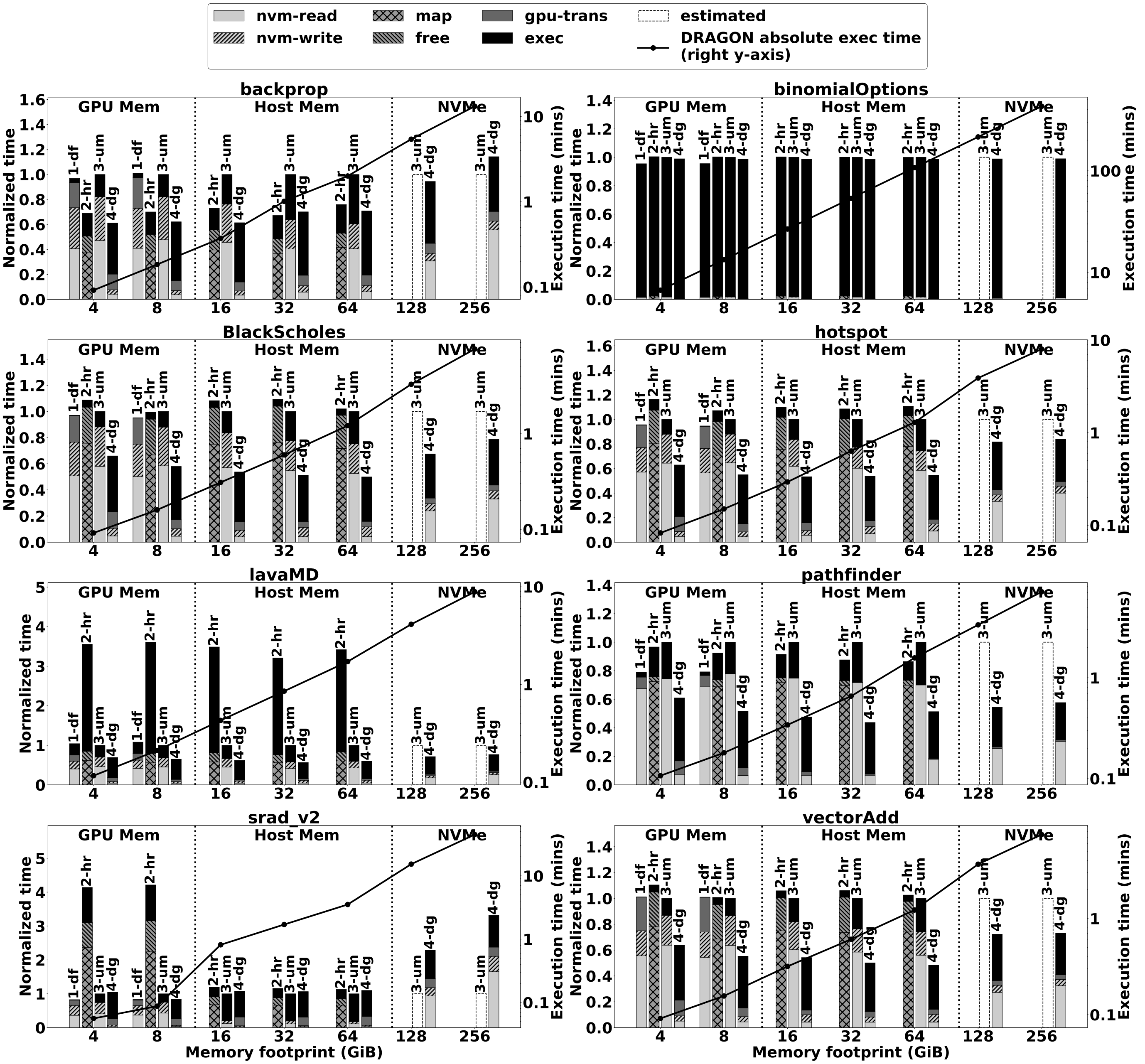


Fig. 3: Application performance comparison: the graphs show the normalized execution time of each technique with respect to the UM-P implementation. Bar #1-df, #2-hr, #3-um, and #4-dg represent Default, Hostreg, UM-P, and *DRAGON* techniques, respectively. GPU memory is 12 GiB, while host memory is 64 GiB. Right y-axis (log scale) is for the solid line and shows the total execution time of *DRAGON* in minutes.

mechanism in the context of NVMs and GPUs so that data-staging is *stream*-lined with the ongoing GPU execution. *Read-ahead* pro-actively starts reading blocks from NVM to the HM as soon as the kernel thread goes idle after finishing the driver operation (§ III). For this reason, the *hit* rate on the GM and the HM (i.e., PCP) is usually high for applications that have *consecutive* data access patterns. In addition to the streaming effect, *DRAGON* was able to further hide the NVM latencies by transparently overlapping CPU-GPU transfers with NVM read/write operations. For non-data-intensive applications, like `binomialOptions`, *DRAGON* performed as well as the other techniques, due to its low overhead operation. Unlike *DRAGON*, other techniques did not have the streaming advantage. This is because all input data were loaded once from the files at the beginning and all output data were dumped out to the files at the end of the applications.

For the *out-of-core* case, where there is no GPU alternative to *DRAGON*, the performance of *DRAGON* was better for most applications, when compared with the linearly extrapolated UM-P values. *This is one of the most significant observations of our*

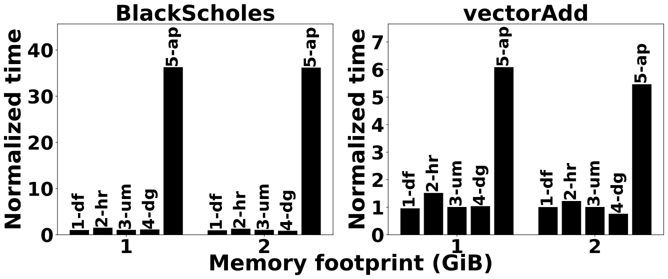


Fig. 4: Total execution time comparison of ActivePointers (#5-ap), DRAGON (#4-dg), and other methods

experiments, and it demonstrates the feasibility of DRAGON as the only efficient out-of-core GPU computing solution with user-oblivious addressing. In two specific cases, `backprop` and `srad_v2`, *DRAGON* was slower than the extrapolated UM-P values. These applications commonly employ a significant amount of *out-of-core* intermediate data buffers, as shown by the ratios reported in table I. Because intermediate data in the UM-P technique are always kept on the HM, the linearly extrapolated values also inherit this property. On the other hand, the *DRAGON* technique *swaps out* those intermediate data to the NVM due to data being larger then HM.

The performance comparison with ActivePointers (AP) is shown separately in fig. 4 for BlackScholes and vectorAdd. The executions with AP terminated only with 1 GiB and 2 GiB memory footprints, although we followed the suggested guidelines to adapt our benchmarks. The results showed up to 35x slowdown when compared with the baseline UM-P execution. In AP, unlike regular memory accesses, warp scheduler cannot switch to another warp on a memory reference. The page fault handling code needs to be run to perform SW TLB operations, and the execution is blocked until the proper data is brought to the GPU. The overhead of this approach is significant.

IV-D. Importance of Read-ahead

To further demonstrate the performance benefits of *DRAGON*, we selected two applications (`hotspot` and `vectorAdd`) and showed the importance of the read-ahead mechanism by running with the *streaming* (i.e., Read-ahead) optimization enabled and disabled. For this experiment, we disabled read-ahead for the mapped areas using the same technique as `posix_fadvise()` with `POSIX_FADV_RANDOM` flag [34] and showed the results in Figure 5. Bars #1, #2, and #3 correspond to the execution time when using the UM-P (baseline) technique, *DRAGON* (default), and *DRAGON* with read-ahead disabled, respectively. Similar to fig. 3, we normalized the results with respect to bar #1 and showed the linearly extrapolated value of the baseline for *out-of-core*.

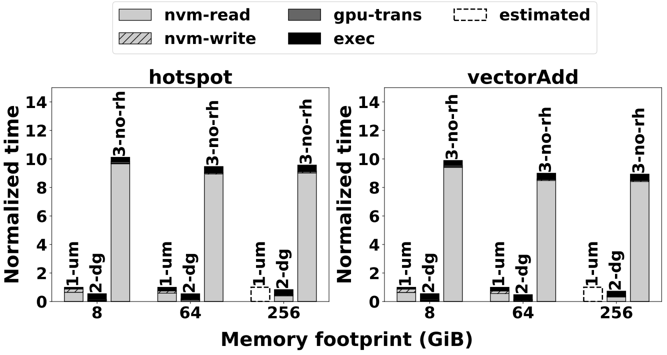


Fig. 5: The importance of read-ahead: the graphs show the normalized execution time with respect to the UM-P implementation. Bars #1, #2, and #3 represent UM-P, *DRAGON*, and *DRAGON* with read-ahead disabled, respectively.

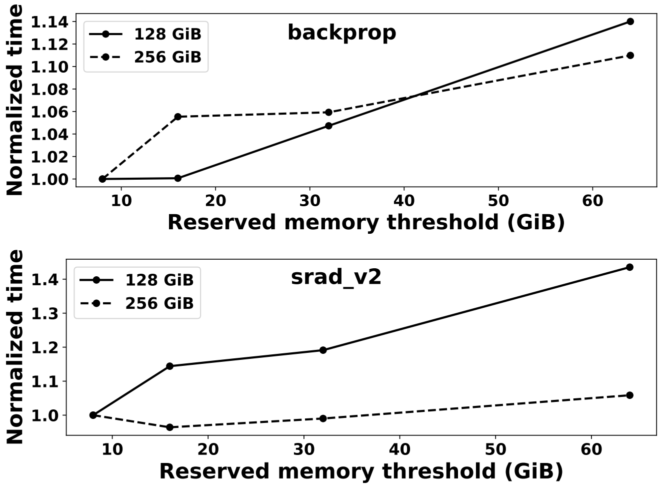


Fig. 6: The effect of intermediate data optimization and threshold for (a) backprop and (b) srad_v2.

The results show how *DRAGON effectively integrates host-based page-caching benefits into NVMs and GPUs*. When read-ahead is disabled, the applications experienced significant slowdown in comparison to the baseline. This was because all data accesses had to go to the NVM on every single page-fault. The latency for initiating a read command also contributes to the slowdown in the non-optimized case. `fread()` that we used in the UM-P technique was faster because we read large data at once (in GBs as opposed to 4 KiB at a time), which amortized the latency overhead.

IV-E. Effect of the Intermediate Data Optimization

As discussed in § III-D, *DRAGON* treats intermediate data specially. This optimization is controlled by a threshold parameter to adjust the size of the HM-based cache dedicated to intermediate data before swapping them out to NVM. We evaluated this optimization on `backprop` and `srad_v2`, since these are the only two applications that employ large intermediate data. In this experiment, we varied the *reserved memory threshold*, the amount of HM reserved for Linux kernel and other processes (§ III-D). We measured the total execution time of these two applications when running *out-of-core* on thresholds varying between 4 and 64 GiB. The 64 GiB threshold means that *DRAGON* does not keep intermediate data on the HM and falls back to use direct PCP access (§ III-A).

Figure 6 shows the results of this experiment. The y-axis represents the execution time normalized with respect to the corresponding 4 GiB threshold, the smallest value in this experiment. The results show that without this optimization (64 GiB threshold), applications would suffer up to 43.5% more overhead, due to more data movement to/from the NVM device. The *sweet spot* depends on the application and the system characteristics. We will leave thorough investigation regarding the interference to future work. In this paper, other than this experiment we report all results when using the 4 GiB threshold.

V. Case Study: Deep Learning with Caffe

In recent years, machine learning, and notably deep learning (DL), has gained significant attention in HPC. Most DL frameworks rely on GPUs to accelerate the computation. However, the problem sizes and complexity of the underlying networks on which those frameworks can operate are limited by the GPU memory. In this case study, we show how *DRAGON* addresses the out-of-core processing problem for large DL inputs while still keeping the GPU performance faster than multicore execution with inputs larger than the total available system memory.

V-A. Caffe Framework

Caffe [12] is a popular neural network framework for training and classifying data from various domains. It supports a large variety of common operations (e.g., conv-2D, ReLU, and pooling) to be executed on GPUs and/or CPUs. In Caffe, each operation is represented by a *layer*. Solvers for a specific classification problem are implemented by forming DAGs of *layer*s, which are called *net*s.

Caffe encapsulates internal data and its corresponding parameters in multidimensional objects called *blob*s, which act as inputs and outputs to the *layer*s of a given *net*. *Blob* objects provide interfaces for users to access CPU or GPU copies of the data being stored. Corresponding data allocation, transfer, and bookkeeping operations required by the access requests are transparently handled by the `SyncedMemory` class.

Adapting Caffe to use *DRAGON* requires minimal changes to data-handling interfaces, mainly in the `SyncedMemory` and `DataLayer` classes. We replaced `malloc`, `cudaMalloc`, `cudaHostAlloc`, `cudaMemcpy` calls, and IO operations with `dragon_map()`.

V-B. Tested Neural Networks and Datasets

We tested the *DRAGON*-integrated version of Caffe using two different neural networks: ResNet[1] [35] for images and Facebook's Convolution3D (C3D) [36] for videos.

ResNet [35] is an award-winning network for deep residual learning for image recognition with a very high detection and localization rate. ResNet provides a set of Caffe nets with varying number of layers for increased accuracy. However, as the layer counts increase, the amount of intermediate data required for neural network cycles increases significantly; therefore, memory requirements quickly exceed GPU and host memory. We trained ResNet with the ILSVRC12 [17] dataset.

C3D is built on top of Caffe to support 3D convolutions for visual classification of video inputs [36]. Memory footprint limitation constitutes a serious obstacle in 3D convolutions since it is not practically possible to break the input video into smaller chunks for classification. For this experiment, we used the UCF101 video dataset [37] as the training data.

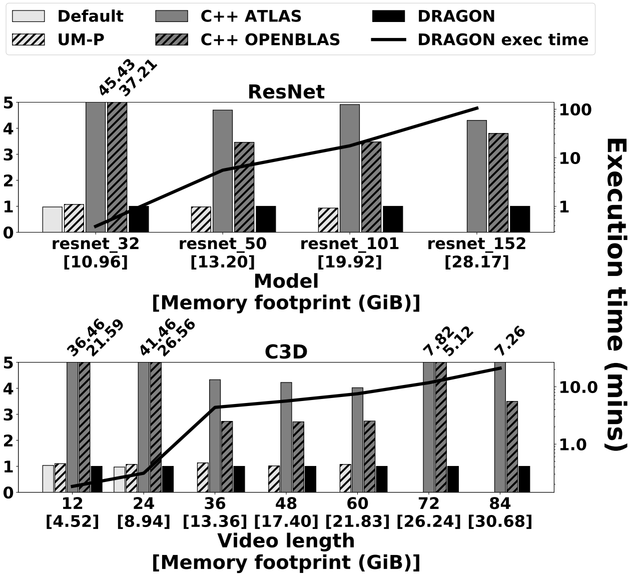


Fig. 7: Comparison of the execution times of various Caffe versions with ResNet and C3D models: all bars are normalized w.r.t. the *DRAGON*-integrated version. Right y-axis (log scale) is for the solid line and shows the total execution time of *DRAGON* in minutes.

V-C. Experiments

In our experiment with Caffe, we used the same platform as specified in Section IV. We performed two separate sets of training, one for each dataset. For ResNet, we used four different networks: ResNet-32 for *incore-GPU*, ResNet-50 and ResNet-101 for *incore-host*, and ResNet-152 for *out-of-core*. We fixed the batch size to 40. For C3D, we modified the network script to increase the video dimension of one batch to $240 \times 240 \times length$ and reduce the batch size to one. We varied the $length$ (i.e., number of frames) variable to change the total GPU memory footprint. For both experiments, we limited the total amount of available host memory to 24 GiB, so that the maximum memory footprints obtained by the largest input in the datasets fall into the *out-of-core* range. We set the number of iterations to 30 so that all experiments (explained later) finished in reasonable time.

We compared the following techniques of execution.

1) *Default* [incore-GPU]: This method is the original CUDA implementation that comes with Caffe, and the memory footprint is limited by the GPU memory size.
2) *CUDA UM-P* [incore-host]: To integrate UM-P, we modified `SyncedMemory` and `DataLayer` and used `cudaMallocManaged` for memory allocation.
3) *C++ ATLAS* [out-of-core]: Because there is no alternative GPU implementation of Caffe for out-of-core processing, we compare *DRAGON* against multi-core execution. For the CPU cores to process data larger than the system memory, we used `mmap()` to directly use the NVM device. ATLAS is the default BLAS library and is limited to four threads.

[1] Network models from https://github.com/yihui-he/resnet-imagenet-caffe

4) *C++ OPENBLAS* [out-of-core]: To utilize all the cores in CPU execution, we replaced the `BLAS := ATLAS` parameter with OPENBLAS, which relies on OpenMP to launch as many threads as needed for the computation.
5) *DRAGON* [out-of-core]: This is the *DRAGON*-integrated version of Caffe, as explained earlier in this section.

Figure 7 shows the normalized execution times on the left y-axis with respect to the *DRAGON*-integrated version. The values on the right y-axis (log scale) correspond to the absolute execution time of the *DRAGON*-integrated version, which is represented with a solid line. The x-axis represents the maximum application memory consumption. For the *incore-GPU* case (up to 12 GiB), *DRAGON* showed the same performance as the Default and UM-P versions. For the *incore-host* case (12 - 24 GiB), *DRAGON* performed significantly faster than the multi-core versions and stayed within 7% overhead range when compared to the UM-P version. The overhead was caused by excessive GPU evictions due to repetitive iteration of intermediate data accesses across network layers. Additionally, because most of the data were intermediate data, *DRAGON* could not exploit the benefits of read-only and write-only related optimizations (§ III-C).

For the *out-of-core* case (rightmost bar for ResNet and two rightmost bars for C3D), both Default and UM-P versions failed to execute. On the other hand, ATLAS-based multi-core execution utilized only four threads, whereas OpenMP-based OpenBLAS C++ execution launched more than 50 threads on our 48 HW-threaded system. The results show that *DRAGON* successfully performs up to 3.8x faster than the OpenBLAS-based execution, which employs a higher number of threads. Overall, multi-core techniques become significantly slower as the footprints increase.

In summary, our experiments demonstrate that GPUs can be used to accelerate *real-world problem*s where the device and host memories are not large enough to carry the computation.

VI. Additional Related Work

Out-of-core data processing has been an important problem in scalable scientific computing on GPUs. Many studies [19], [38], [39] have proposed application-specific algorithms to break down the computation and data into smaller chunks that fit into GPU memory. While these approaches are optimized for the applications they target, they rely on manual data staging, orchestration, and transfers that are developed exclusively for the algorithm in question and fail to provide any generic solutions.

A few studies developed more generic software approaches to enable efficient data management between CPUs and GPUs for out-of-core data processing. Several compiler-based techniques [1], [2], [14] analyze the data flow, inject code to automatically partition parallel loops and tasks into smaller regions, and then map them into GPU(s) and CPU(s). Other software-based approaches [15], [16] provide user-level APIs along with runtimes to dynamically manage the data movement and consistency across different regions of large data. These approaches either require complex compiler analysis that is limited to certain code structures or significant programming effort to integrate their APIs.

There have also been several OS- and hardware-based studies to address the problem in a lower-level way, with minimal software involvement. Papers in [40], [41] proposed driver-based memory management solutions accompanied with architectural modifications. Most notably, a few studies [42], [43] built hardware-based page-faulting mechanisms similar to UM-P. Overall, these studies commonly suggested impractical hardware modifications, and most of them became obsolete when NVIDIA introduced unique UM capabilities of the Pascal architecture [44]. More importantly, none of the studies above considered data beyond host memory.

VII. Conclusion

In this study, we propose the *DRAGON* framework to address the out-of-core data access problem for GPU applications that operates on data larger than the host memory. *DRAGON* allows NVM storage to be directly mapped to GPUs as their *primary addressable space* to enable *large memory* support for existing GPU-accelerated applications. *DRAGON* uniquely alters the conventional GP-GPU programming paradigm for large-data workloads by transparently extending the addressable global memory space to the limits of NVM storage. Our evaluation of various scientific benchmarks and real-life deep learning workloads shows that *DRAGON efficiently* enables *out-of-core GPU computing* for a wide range of application classes. Moreover, *DRAGON* improves incore-GPU and incore-host executions up to 2.3x compared with using UM-P + POSIX IO.

In a broader sense, *DRAGON* demonstrates that large-scale application developers can design and partition their algorithms *"free"* from the limitation of smaller GPU or host memories, without any additional performance cost or programming effort.

Acknowledgments

This research was partially supported by JST CREST Grant Numbers JPMJCR1303 (EBD CREST) and JPMJCR1687 (DEEP CREST), and performed under the auspices of Real-World Big-Data Computation Open Innovation Laboratory (RWBC-OIL), Japan. This research was also supported in part by Oak Ridge National Laboratory ASTRO Program sponsored by the US Department of Energy and administered by the Oak Ridge Institute for Science and Education, USA.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, “Scalable framework for mapping streaming applications onto multi-gpu systems,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’12)*, 2012, pp. 1–10.

[2] A. Sabne, P. Sakdhnagool, and R. Eigenmann, “Scaling large-data computations on multi-gpu accelerators,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS’13)*, 2013, pp. 443–454.

[3] J. S. Vetter and S. Mittal, “Opportunities for nonvolatile memory systems in extreme-scale high performance computing,” *Computing in Science and Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[4] W. Cheong and C. Y. et.al, “A flash memory controller for 15us ultra-low-latency SSD using high-speed 3D NAND flash with 3us read time,” in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 338–340.

[5] H. M. et. al, “A 512gb 3b/cell 3D flash memory on a 96-word-line-layer technology,” in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 336–338.

[6] Intel, “Intel optane technology,” https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, 2017, accessed: 2017-08-01.

[7] S. Mittal and J. S. Vetter, “A survey of CPU-GPU heterogeneous computing techniques,” *ACM Computing Surveys*, vol. 47, 2015.

[8] *Whitepaper NVIDIA Tesla P100*, Wp-08019-001_v01.1 ed., NVIDIA.

[9] N. Sakharnykh, “Beyond gpu memory limits with unified memory on pascal,” NVIDIA Corporation, Tech. Rep., 2018, accessed: 2018-01-22. [Online]. Available: https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/

[10] *CUDA Runtime API*, v9.1.85 ed., NVIDIA, July 2017.

[11] *CUDA C Programming Guide*, Pg-02829-001_v9.1 ed., NVIDIA, December 2017.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[13] *AMD APP SDK OpenCL User Guide*, rev1.0 ed., AMD, August 2015.

[14] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, “Dynamically managed data for cpu-gpu architectures,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO ’12)*, 2012, pp. 165–174.

[15] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, “An asymmetric distributed shared memory model for heterogeneous parallel systems,” in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS’10)*, 2010, pp. 347–358.

[16] N. Al-Saber and M. Kulkarni, “Semcache++: Semantics-aware caching for efficient multi-gpu offloading,” in *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS’15*, 2015, pp. 79–88.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] H. Seo, J. Kim, and M.-S. Kim, “Gstream: A graph streaming processing method for large-scale graphs on gpus,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’15)*, 2015, pp. 253–254.

[19] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, and S. Matsuoka, “Large-scale distributed sorting for gpu-based heterogeneous supercomputers,” in *2014 IEEE International Conference on Big Data (Big Data)*.

[20] S. Sreepathi, J. Kumar, R. T. Mills, F. M. Hoffman, V. Sripathi, and W. W. Hargrove, “Parallel multivariate spatio-temporal clustering of large ecological datasets on hybrid supercomputers,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[21] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, “The future of scientific workflows,” *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[22] J. Dean and G. C. et.al, “Large scale distributed deep networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1223–1231.

[23] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. Collier, J. Bauer, F. Xia, and T. e. a. Brettin, “Candle/supervisor: A workflow framework for machine learning applied to cancer research,” 2017.

[24] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49.

[25] H. W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, “Morpheus: Creating application objects efficiently for heterogeneous computing,” *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA’16)*, pp. 53–65, 2016.

[26] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, “Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures,” *Parallel Architectures and Compilation Techniques - Conference Proceedings (PACT’16)*, pp. 13–24, 2016.

[27] *Developing a Linux Kernel Module using GPUDirect RDMA*, Tb-06712-001 _v9.1 ed., NVIDIA, December 2017.

[28] S. Shahar, S. Bergman, and M. Silberstein, “Activepointers: a case for software address translation on gpus,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[29] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, “Gpufs: integrating a file system with gpus,” in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 485–498.

[30] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. O’Reilly Media, Inc., 2005.

[31] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O’Reilly Media, Inc.", 2005.

[32] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[33] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, “A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads,” in *IEEE International Symposium on Workload Characterization (IISWC 2010)*, 2010, pp. 1–11.

[34] *POSIX_FADVISE(2) - Linux manual page*, 2013rd ed., Linux, April 2013.

[35] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *arXiv preprint arXiv:1512.03385*, 2015.

[36] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, “Learning spatiotemporal features with 3d convolutional networks,” in *Proceedings of the IEEE international conference on computer vision*, 2015.

[37] K. Soomro, A. R. Zamir, and M. Shah, “UCF101: A dataset of 101 human actions classes from videos in the wild,” *CoRR*, vol. abs/1212.0402, 2012.

[38] K. Kabir, A. Haidar, S. Tomov, A. Bouteiller, and J. Dongarra, “A framework for out of memory svd algorithms,” in *International Supercomputing Conference (ISC’17)*. Springer, 2017, pp. 158–178.

[39] T. Endo, “Realizing out-of-core stencil computations using multi-tier memory hierarchy on gpgpu clusters,” in *2016 IEEE International Conference on Cluster Computing (CLUSTER’16)*.

[40] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang, “Gdm: Device memory management for gpgpu computing,” in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’14)*, 2014, pp. 533–545.

[41] J. Cabezas, I. Gelado, J. E. Stone, N. Navarro, D. B. Kirk, and W.-m. Hwu, “Runtime and architecture support for efficient data exchange in multi-accelerator applications,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 5, pp. 1405–1418, 2015.

[42] Y. Kim, J. Lee, J. E. Jo, and J. Kim, “Gpudmm: A high-performance and memory-oblivious gpu architecture using dynamic memory management,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA’14)*, 2014, pp. 546–557.

[43] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, “Umh: A hardware-based unified memory hierarchy for systems with multiple discrete gpus,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, 2016.

[44] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, “Towards high performance paged memory for gpus,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA’16)*, 2016, pp. 345–357.

[45] J. Glisse, “Heterogeneous memory management (hmm),” https://lwn.net/Articles/679300/, 2016, accessed: 2018-01-01.

[46] J. G, “Heterogeneous memory management (hmm),” https://github.com/torvalds/linux/blob/master/Documentation/vm/hmm.txt, 2018, accessed: 2018-01-22.

[47] C. Newburn, “Unifying memory on accelerated platforms,” hhttps://www.redhat.com/cms/managed-files/HMM%20for%20SC%20171114_0.pdf, NVIDIA, 2018, accessed: 2018-03-01.

[48] J. Hubbard, “Using hmm to blur the lines between cpu and gpu programming,” http://on-demand.gputechconf.com/gtc/2017/presentation/s7764_john-hubbardgpus-using-hmm-blur-the-lines-between-cpu-and-gpu.pdf, NVIDIA, 2018, accessed: 2018-01-22.

[49] “Direct access for files,” https://www.kernel.org/doc/Documentation/filesystems/dax.txt, 2017, accessed: 2017-08-01.

Appendix

Alternative Implementation Options

This appendix discusses other possible approaches that may be alternatives to *DRAGON*.

GPUDirect over RDMA is a popular means to directly transfer data between a DMA-capable device and GPUs without requiring a copy in the host memory. The NVM device driver can obtain the target GPU page via the kernel-level `nvidia_p2p_get_pages()` method and initiate a direct transfer. For in-core processing, this approach is shown to be feasible [25], [26] and provides considerable speedup against host-memory-based approaches. However, for out-of-core processing via UM-P, the CUDA manual [27] states that using RDMA on the UM-P's managed memory can result in possible data loss or corruption due to incoherency across address ranges. For this reason, implementation of an RDMA version of *DRAGON* while ensuring the memory consistency is not possible.

cudaHostRegister with mmap is an alternative method to extend the addressable range of GPUs to the storage devices via standard `mmap()`. `cudaHostRegister` exposes the specified host memory region to the GPU, allowing GPU to directly load/store data on that region [10]. By combining it with `mmap()`, kernels can directly address the NVM file, in a manner similar to UM-P. However, internally, this approach relies on the CUDA mapped memory concept, and each page in the non-locked memory region needs to be first copied into a pinned buffer in host memory before GPU can directly access it [10]. Therefore, the total data size we can *map* is still limited by the host memory. This approach is compared against *DRAGON* in our evaluation.

Heterogeneous Memory Management (HMM) [45] is a recent effort by the Linux community to get rid of the decoupled memory management that has historically existed between host and device memories. HMM provides a helper layer for the device driver to shadow the page table of a CPU process so that both the device and the CPU can use the same memory space allocated via `malloc()`. Enabling HMM requires device-specific drivers, and it will allow CPU memory to be DMA'd without being pinned [46]. For NVIDIA GPUs, HMM support will handle page movement only between CPUs and GPUs (i.e., HM-to-GM) [47], similar to what UM-P does, and will not provide any help while accessing the storage (i.e., NVM-to-HM transfers) [48]. Hence, *benefits provided by DRAGON and HMM are orthogonal*. Once the driver support is released by NVIDIA, HMM will further help *DRAGON* to operate faster via improved HM-to-GM transfers. To use HMM with *DRAGON*, programmers would still need to replace `malloc()` calls with `dragon_map`. However, *DRAGON* will inherit internal driver-level benefits of HMM.

Linux direct access (DAX) [49] is another kernel feature added recently, and it enables skipping page caches entirely for memory-like block-access devices, such as NVDIMMs. When such modules become available, DRAGON will be able to utilize them through Linux file interfaces. Because the existence of page-caches is totally transparent to DRAGON driver, no changes will be required. The performance that is lost due to a missing read-ahead mechanism will be compensated by much lower access latencies provided by NVDIMMs.

Artifact Description

A. Abstract

Artifact described in this section includes the source code of *DRAGON* and applications used in over evaluation. For *DRAGON*, the source code is separated into two parts: 1) modified `nvidia-uvm` driver, and 2) *DRAGON* library to be used by applications. For applications, the artifact includes three versions: 1) the original version, 2) the UM version that we modified the original version to use Unified Memory (UM), and 3) the *DRAGON* version that we integrated *DRAGON* to the original version.

The scripts to compile the source code, generate or download inputs, execute binaries, validate results, and parse the outputs also included in the artifact and explained below in detail.

B. Description

B1. Check-list (artifact meta information):

- **Program:** Python2.7 or above
- **Compilation:**
  - NVIDIA `nvcc` version 9.0 or above
  - `gcc` version 4.8.5 or above
  - glibc-2.0 header
  - libatlas header
  - libopenblas header
  - cuDNN version 7.0 or above
  - opencv-3.4.0 or above
  - Linux kernel header of the running OS
- **Data set:**
  - To evaluate Caffe, `ILSVRC12` and `UCF101` datasets are needed. Please visit http://www.image-net.org/challenges/LSVRC/2012/ and http://crcv.ucf.edu/data/UCF101.php to obtain the data sets, respectively.
  - For the other applications, scripts for generating data are provided with the artifact.
- **Run-time environment:**
  - Linux OS (tested on CentOS 7)
  - Linux kernel version 3.10 or above but below 4.0.
  - NVIDIA GPU Driver v384.81
  - NVIDIA CUDA version 9.0 or above
  - glibc-2.0 library
  - libatlas
  - libopenblas
  - cuDNN version 7.0 or above
  - opencv-3.4.0 or above
- **Hardware:**
  - One NVIDIA Pascal P100 GPU or above (the GPU needs to support HW page-fault)
  - One NVMe storage formated with ext-4 file system
- **Output:** Verification results and detailed timings such as execution times and runtime overhead
- **Experiment workflow:** Linux bash or python scripts
- **Publicly available?:** Yes

B2. How software can be obtained: The source code of *DRAGON* and all experimented applications, including the baseline versions, can be obtained from https://github.com/pakmarkthub/dragon. Up-to-date documents and instructions can also be found in the repository.

B3. Hardware dependencies: We performed our experiments on an NVIDIA P100 GPU and a 2.4 TB Micron 9100 HHHL U.2 PCIe NVMe drive formatted with ext-4 file system. *DRAGON* is compatible with any NVIDIA GPUs that support hardware page-fault mechanism (Pascal P100 or above), and any NVMe drive that formatted with ext-4 file system. However, we recommend an NVMe drive that has capacity more than 1.5 TB as the largest dataset for an application can take up to almost 800 GB and some additional NVMe space is needed for holding intermediate data. We also recommend one to employ an additional storage that has at least 5 TB. This additional storage is for holding datasets of all experiments. Beside what have been stated, the host machine must also have main memory (DRAM) capacity more than that of GPU memory.

B4. Software dependencies: We implemented *DRAGON* as an extension of NVIDIA GPU driver. The artifact contains the driver patch for NVIDIA GPU driver version 384.81, which comes with CUDA 9.0 installation (obtainable from NVIDIA's website). Other required software packages/applications are as stated in appendix B1.

B5. Datasets: The artifact provides scripts for generating datasets for almost all of the applications. However, the datasets for evaluating Caffe need to be obtained separately. See appendix B1 for more details.

C. Installation

See appendix B2 on how to obtain the source code. Then go to the main folder, and run the `install.sh` script. Root privilege is needed to install *DRAGON*.

```
git clone https://github.com/pakmarkthub/dragon.git
cd <dragon-main-directory>
sudo ./install.sh
```

The steps above will compile *DRAGON* driver and library, and set up necessary environment variables. To compile the evaluated applications:

```
cd <dragon-main-directory>/benchmarks
make -j
```

D. Experiment workflow

All of the applications we discussed in § IV come with this artifact, including the baseline versions and dataset generator scripts. The applications can be found in `<dragon-main-directory>/benchmarks`. In that folder, we also provide a script for generating data (`data-generator.sh`) and a script for repeating all of the experiments (`run.sh`). Before running `data-generator.sh`, one needs to obtain `ILSVRC12` and `UCF101` datasets. The `data-generator.sh` script will generate and convert all necessary data on the specified folder. We recommend that the folder is on the additional storage, not on the NVMe device for doing experiments, since the total data size is about 4 TB. The `data-generator.sh` script can take several hours to a day depending on the CPUs.

After successfully generating datasets, one can run `run.sh`, which can also be found in `<dragon-main-directory>/benchmarks`, to repeat all of the experiments. The script needs root privilege to execute because it needs to insert and remove *DRAGON* driver multiple times to compare between the baseline and *DRAGON*-integrated versions. The script will also copy only necessary data of each experiment to the specified NVMe. The largest capacity needs for an experiment on NVMe should be less than 1.5 TB. The `run.sh` script can also take several hours to a day to complete all of the experiments.

```
cd <dragon-main-directory>/benchmarks
./data-generator.sh <path-to-folder-to-store-data> <path-to-
     ILSVRC12-dataset> <path-to-UCF101-dataset>
sudo ./run.sh <path-to-folder-on-nvme> <path-to-the-main-folder-
     that-store-generated-data>
```

E. Evaluation and expected result

After successfully executing `run.sh`, the result of each experiment will be recoded to files under `<dragon-main-directory>/benchmarks/<application-directory>/<results>`. We provide multiple scripts in `<dragon-main-directory>/benchmarks/analyzers` for converting raw result data into `csv` format and for plotting graphs we showed in § IV and V.

To convert raw result data into human-readable `csv` format, use the following command. The converted files will be stored in each application folder (`<dragon-main-directory>/benchmarks/<application-directory>/<results>`). This step needs to be run before generating a graph from the result.

```
cd <dragon-main-directory>/benchmarks/analyzers
python convert_result.py
```

Following python scripts can be draw to generate respective figures:

```
cd <dragon-main-directory>/benchmarks/analyzers
python ptc.py  # Figure 3
python plot_compare_readahead.py  # Figure 5
python plot_opt_result.py  # Figure 6
python plot_resnet_normalized.py  # Figure 7
python plot_c3d_normalized.py  # Figure 7
```

F. Experiment customization

We used different host DRAM capacity to run the experiments shown in § IV and V. Add `mem=24000m` line to the end of the line starting with `linux /boot/vmlinuz-...` in GRUB bootloader.

```
linux /boot/vmlinuz... mem=24000m
```

To run each experiment separately, go to each application folder (`<dragon-main-directory>/benchmarks/<application-directory>`) and run `run.sh` inside the folder.

To disable the read-ahead of *DRAGON*, include `DRAGON_READAHEAD_TYPE=disable` to the environment variables.

To change *DRAGON*'s reserved memory threshold (§ IV-E), include `DRAGON_NR_RESERVED_PAGES=<number-of-reserved-pages>` to the environment variables. One reserved page has 4 KiB size in default Linux configuration.