Proceedings of 15th IEEE International Conference on Networking, Architecture, and Storage (NAS 2021), October 2021

GPU-Assisted Memory Expansion

Pisacha Srinuan*, Purushottam Sigdel*, Xu Yuan*, Lu Peng§, Paul Darby*, Christopher Aucoin*, and Nian-Feng Tzeng*

*University of Louisiana at Lafayette
{pisacha.srinuan1, purushottam.sigdel1, xu.yuan, paul.darby, christopher.aucoin1, tzeng}@louisiana.edu

§Louisiana State University, Baton Rouge
lpeng@lsu.edu

***Abstract*—Recent graphic processing units (GPUs) often come with large on-board physical memory to accelerate diverse parallel program executions on big datasets with regular access patterns, including machine learning (ML) and data mining (DM). Such a GPU may underutilize its physical memory during lengthy ML model training or DM, making it possible to lend otherwise unused GPU memory to applications executed concurrently on the host machine. This work explores an effective approach that lets memory-intensive applications run on the host machine CPU with its memory expanded dynamically onto available GPU on-board DRAM, called GPU-assisted memory expansion (GAME). Targeting computer systems equipped with the recent GPUs, our GAME approach permits speedy executions on CPU with large memory footprints by harvesting unused GPU on-board memory on-demand for swapping, far surpassing competitive GPU executions. Implemented in user space, our GAME prototype lets GPU memory house swapped-out memory pages transparently, without code modifications for high usability and portability. The evaluation of NAS-NPB benchmark applications demonstrates that GAME expedites monotasking (or multitasking) executions considerably by up to 2.1× (or 3.1×), when memory footprints exceed the CPU DRAM size and an equipped GPU has unused VDRAM available for swapping use.**

Keywords—GPUs** (graphic processing units), **NVMe** (non-volatile memory express), **PCI Express, Virtual Memory Page Swapping.

I. INTRODUCTION

The graphic processing unit (GPU) widely adopted recently by computer systems possesses a growing on-board memory capacity, up to 32 GB for the latest most powerful NVIDIA GPU V100 and 24 GB for TITAN RTX GPUs [4]. Meanwhile, AMD RDNA 2 (Radeon RX 6800 Family) has 16 GB memory. Parallel software libraries, such as Nvidia CUDA (compute unified device architecture) [1] and OpenCL [2], expose GPU programming to users for accelerating application executions ranging from simple web browsing to machine learning (ML) and data mining (DM). Although the GPU has sparked interest in parallelized tasks with regular data access patterns (e.g., ML and DM), it is not preferred for applications with large execution memory footprints and irregular data accesses. The GPU execution time rises markedly under irregular data accesses (in many scientific applications) and is observed to surge greatly in our experiments when memory footprints approach or exceed the on-board GPU physical memory size, as observed previously [5], [6].

We have evaluated off-load GPU execution performance for different parallel applications with a range of execution memory footprints, confirming significant performance degradation caused by the virtual memory subsystem when execution memory footprints approach or exceed the GPU on-board memory, which is also referred to as video DRAM (**VDRAM**) size, as detailed in Section 4 and also pointed out earlier [5]. Specifically, the execution times of NAS Parallel Benchmarks (NPB) [22] under the OpenMP version for CPU workloads and under the CUDA version [23] for GPU workloads, have been obtained by our experiments on the testbeds whose host computers (Dell servers) are equipped with TITAN RTX GPUs [4], as shown in Fig. 1. Their accompanying workload memory footprints are listed in Table I. From the table, it is evident that GPU workloads take far more memory than the CPU counterparts, as also found previously [8]. This is due chiefly to the heavy CUDA software stack available to ease GPU-accelerated application development. Such exceedingly high memory footprint overhead caused by the CUDA toolkit can make GPU executions inferior to their CPU counterparts when the workload memory footprint approaches the GPU physical memory size. It is observed from Fig. 1(a) that GPU execution times can dwarf their CPU counterparts as a result of much larger memory footprints and thus far more page faults, besides naïve GPU memory virtualization support. This makes it unsuitable to run applications with large data sizes on the GPU, unless hardware provisions are made (like the addition of NVLink 2.0 [10] for pooling VDRAM of two GPUs together, as attempted in [5]). Recent studies on swapping GPU pages to host machine memory smartly [6] and on tensor eviction/prefetching and recomputation [7] during deep ML training aim to let GPUs handle enlarged memory footprints, but they target solely at workloads with regular computation structures or predictable data access patterns known prior to execution, not at general parallel workloads without regular computation structures or predictable access patterns.

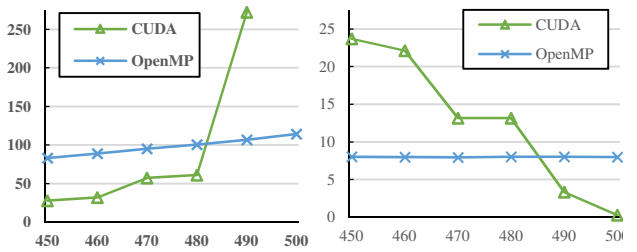
In contrast, CPU executions, with efficient virtual memory support, are seen in Fig. 1(a) to exhibit gradually degraded performance as the SP footprint grows (caused mainly by more virtual memory page swapping), free from abrupt, drastic performance declines. Other applications in the NPB benchmark suite [22] follow similar trends when comparing

This work was support in part by the NSF under Grants CNS-1948374 and OIA- 2019511.

their GPU executions versus compatible CPU executions. Hence, a server-grade computer equipped with the GPU accelerator(s) can favor CPU executions over their GPU counterparts for general parallel applications under (1) monotasking with large memory footprints and irregular data accesses, and (2) multitasking for resource consolidation [9] with the aggregate memory footprint of co-running tasks to exceed the GPU VDRAM size.

CPU executions on such a computer can avoid performance degradation even when their memory footprints exceed the host machine memory size by borrowing available GPU-side memory on demand, called GPU-assisted memory expansion (GAME). This work addresses GAME in support of memory-intensive CPU executions for better performance. The GAME approach can be promising in such a computer system because (1) an equipped GPU usually has unoccupied on-board memory, when applied for lengthy ML training because it is often provisioned to handle extremely deep and wide ML models, instead of regular ones, (2) communication bandwidth between CPU and GPU will increase significantly under the future PCIe 5.0 [11], and (3) multiple GPUs may exist in a system, each leaving a portion of its memory unoccupied when conducting its ML/DM executions; collectively, available GPU memory can be aggregated to accelerate CPU executions that take place concurrently. Our GPU testbeds, for example, are established for weather parameter prediction based on the LSTM ML model, whose model training memory footprint is found to be far smaller than 24 GB (of VDRAM in each equipped TITAN RTX GPU [4]). Two GPU cards exist for each of our testbeds in order to train many weather parameter models simultaneously [12], knowing that certain motherboards can support many GPU cards (even up to 20 each via PCIe over USB or Thunderbolt). Available GPU memory so exposed serves as the swap partition to house cold pages that are claimed by the operating system (OS) to free up host memory needed upon page faults. Page faults come naturally with memory virtualization that enables application executions with arbitrarily large memory footprints on the CPU [13]. In essence, GAME keeps swapped-out pages in GPU's fast VDRAM transparently instead of a slow storage device, realizing low-latency memory page swapping efficiently to boost execution performance without any additional hardware cost, since GPUs are equipped to expedite ML training and DM tasks (and *not to serve dedicatedly as backing store*).

GAME is implemented in user space to prototype testbeds for evaluation, by employing a Linux network block device framework, called *nbdkit*, to let GPU memory house fault-memory pages (as swap-out partitions) transparently, without any modification to application codes upon executed on CPU for high usability and portability. The implemented GAME testbed (under Ubuntu 20.04 Focal Fossa) has demonstrated that CPU executions with an aid of GAME under multitasking workloads (when aggregate memory footprints far exceed the VDRAM size) can be an order of magnitude faster (see Fig. 3) than their compatible GPU executions, confirming the potentials of substantial advantages of CPU executions on server-grade machines equipped with GPU accelerators. GAME achieves marked speedups up to 2.1 times (or 1.6 times) in comparison to its counterpart without GAME support



(a) Benchmark SP execution time (in minutes) versus problem size

(b) Operations per second (billions) versus problem size

Fig. 1. CPU and GPU execution performance comparison.

TABLE I. TOTAL WORKLOAD MEMORY FOOTPRINTS

Problem size	450	460	470	480	490	500
SP CUDA	29.3 GB	31.2 GB	32.8 GB	34.6 GB	36.7 GB	38.5 GB
SP OpenMP	15.4 GB	16.4 GB	17.5 GB	18.6 GB	19.8 GB	21.0 GB

for large monotasking workloads under ample (or limited) VDRAM available from the GPU for swapping use.

Although current GAME design has demonstrated its advantages in support of general parallel executions on computer systems equipped with GPU cards, further improvement is possible. Specifically, instead of estimating the maximal unused GPU memory before a GPU execution starts (according to [14]), one can monitor and predict available GPU memory amounts dynamically during execution (see Section III.D). In addition, one may schedule the best swapping use of available VDRAM over multiple GPUs in a computer system or even two systems connected by the NVLink 2.0 [10], taking GPU memory availability and PCIe communication load into consideration.

II. RELATED BACKGROUND

GAME harvests available GPU memory for use to accelerate CPU executions with big memory footprints. Its related background is outlined below.

A. *GPU Applications and Memory Consumption*

GPU application executions usually have different resource needs (e.g., streaming multiprocessor counts, VDRAM, communication bandwidth, etc.) at different execution stages, just like CPU executions as documented previously [13]. Effort has been made for profiling GPU resource needs [15] or for estimating GPU memory consumption [14] during GPU executions. In particular, ML applications have regular computation structures with orderly, known data access patterns, making it possible to estimate memory consumption reasonably precisely. For example, GPU memory consumption under deep ML of VGG-16 (with the min-batch size of 128) and TensorFlow is estimated to be 16.9 GB versus the actual measured 17.4 GB, whereas LSTM under TensorFlow is estimated to consume 4.3 GB versus the actual 4.1 GB [14]. When ML models are trained on a GPU, the unused memory amount over the often lengthy training duration is model-dependent and is known *apriori*, so that GAME can easily utilize it to expedite CPU executions which take place in the duration. We evaluate GAME on one of our testbeds under varying amounts of GPU memory available to hold CPU execution swapped-out pages on demand.

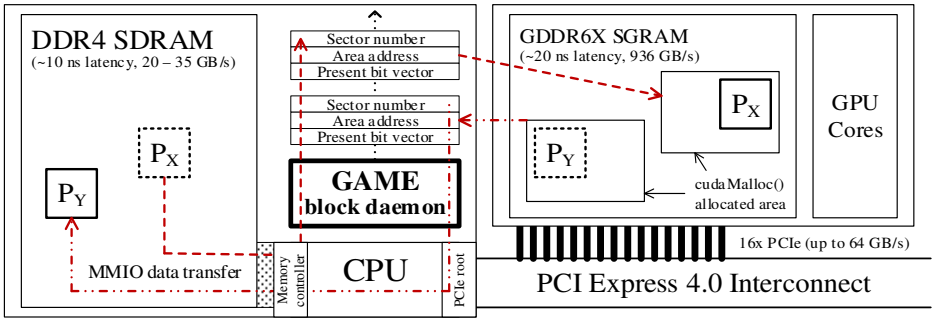


Fig. 2. GPU-assisted memory expansion high-level design diagram.

TABLE II. NPB SMALL (S) DETAILS AND FOOTPRINTS

Bench marks	Problem size	Footprint (GB)
BT	530*530*530	23.4
FT	512*1024*1024	20.0
LU	550*550*550	21.4
SP	530*530*530	24.5
MG	Class D	26.5

B. Network Block Devices

The network block device (NBD) is a network protocol that exposes software-emulated data blocks to the network. There are many NBD implementations available in various types and forms. Linux NBD is a basic NBD library included in the Linux kernel, and it provides simple functionalities employing the standard TCP/IP socket for communication. Mellanox nbdX and NVMe over communication fabrics aim to provide block devices through RDMA and Fiber channels [16]. Being flexible, ***nbdkit***, developed and maintained by Red Hat [17], is a high-performance software framework for block devices. It permits users to write plugins for managing unconventional sources. We exploit *nbdkit* to realize our GAME for intercepting every swapped-out and swap-in page request. Additionally, NBDs are software emulated solutions, and hence, they are not bonded to any hardware and available in user space. Although the user space methods have greater flexibility, they incur higher software overhead than their kernel-level implementation counterparts.

III. DESIGN AND IMPLEMENTATION

The virtual memory (VM) subsystem is widely adopted by OS for efficient memory utilization. It possesses many benefits for execution processes, such as contiguous private address space, process isolation, security, page frame sharing, demand paging, and copy-on-write. VM allows the running processes to allocate memory with the size far larger than that of the system physical memory. Nvidia CUDA exploited the VM technique to expand GPU memory onto host machine memory, arriving at UVM (unified virtual memory).

Our GAME utilizes unused GPU memory on-demand to quicken CPU execution under the virtual memory paradigm. It aims to lower the CPU execution time by dynamically borrowing available GPU VDRAM. GAME exposes GPU memory as a block device to the host OS during low GPU memory utilization upon a light GPU load (or an idle state).

Under Linux OS, GAME is mounted as a swap partition for memory expansion via a block device. During process execution, OS claims cold memory pages and evicts them to the swap device periodically, known as page frame reclaiming. More page frames are reclaimed under higher memory pressure. Instead of a slower block storage devices like a hard disk drive (HDD) or a solid-state drive (SSD), GAME keeps evicted pages in available GPU memory areas, which is much faster than resorting to other block storage devices. Nvidia CUDA library is employed to let GAME transfer data pages between GPU and CPU on demand transparently. Upon receiving evicted pages from OS, if GAME observes low GPU utilization with available memory to lend, it allocates a VDRAM area to store data. Note that the memory area size is configurable and must be a multiple of 4KB; thus, an allocated memory area can contain several pages. Whenever GAME predicts that the GPU is highly loaded with its memory mostly occupied, GAME writes the evicted pages to a temporary file (as one memory area) provisioned for the storage device (say, /var/tmp), achieving identical performance as the baseline SSD.

GAME keeps tracks of the GPU memory status using memory area descriptors, each of which represents one allocated memory area. The descriptors are managed in an LRU linked list, with the least recently used entry placed at its front for performance optimization. Maintained by CPU in host memory, a memory area descriptor consists of three fields: (1) sector number (the unit of block device referenced by OS), (2) GPU memory address (a pointer returned from CUDA memory allocation) for GAME to convert between two marked locations interchangeably, and (3) the present bit vector (PBV) to represent active pages in the memory area. The i^{th} bit of the PBV is raised to convey that the i^{th} page of this memory area is in use, and the bit is cleared when the corresponding page is freed by OS. A memory area descriptor is created when its associated memory area is first allocated for holding evicted pages. After that, the descriptor is populated and placed at the head of the linked list. On the other hand, whenever the system frees a backed swapfile, GAME clears its associated bit (set to zero) to denote that the page is no longer in use. If all bits in the PBV are cleared, GAME releases the allocated memory area back to GPU, realizing dynamic memory expansion.

Fig. 2 depicts the scenarios when the GAME block daemon receives a write-request (page frame reclamation) and read-request (page fault) from OS. Under high system memory pressure, a page (say, P_X) is marked as a cold page and sent to the block device subsystem from where the GAME daemon intercepts the evicted page P_X. As tracked by the dashed line in Fig. 2, upon receiving P_X, GAME checks the linked list for the corresponding memory area descriptor where P_X's sector number has resided. Suppose the descriptor exists in the list. In that case, GAME writes the received P_X to the determined offset position, sets the corresponding present bit to 1 in the PBV, and brings the descriptor to the front of the LRU list. If P_X is new and its corresponding descriptor does not exist in the LRU list, GAME allocates a new memory area in VDRAM

along with its descriptor, given that GPU is lightly loaded. Otherwise, instead of VDRAM, P_X is redirected to the system storage.

Similarly, a read request for a page that is previously swapped-out (say, P_Y), causes a page fault. Intercepting the page fault request from a block device interface for P_Y, GAME looks up the memory area descriptor for P_Y, as denoted by the dotted-dash line in Fig. 2. In this case, GAME always finds the corresponding record in the LRU list that points to a target location either in VDRAM or in the temporary file (i.e., /var/tmp). After moving the P_Y to the designated buffer, the read request is fulfilled. Note that, at this point, P_Y still cannot be released, with its present bit still set to 1 since the swapfile can be requested by OS multiple times throughout the execution course. All present bits will be cleared only when the system explicitly releases them.

A. Kernel Mode and User Mode

To allow the system to utilize GPU memory, GAME targets VDRAM as a block device that can be realized in both kernel space and user space. The realization under kernel space versus user space has different advantages and limitations, as discussed next.

Natively supported by Linux, the block device driver can be realized as a kernel module. The block device driver serves as an entry point to a backing hardware device and is responsible for orchestrating I/O requests produced by applications [18]. Moreover, a block device driver is a glue layer that controls hardware and provides random accesses to the application's fix-sized block data. It is a low-level implementation that aims at high performance [19]; thus, the block driver is tied tightly to the system and is confined in kernel space. Despite having thin overhead and high performance, a kernel-level approach suffers from limited accesses to high-level libraries, such as Nvidia CUDA.

Although GAME aims at high-performance support for CPU execution to lean its implementation in kernel space, major problems and concerns emerge with kernel space implementation, since the Linux block driver has no access to Nvidia CUDA, which is essential to interface with GPU memory. Our first attempt at GAME implementation is to involve three parts: (1) Linux block device driver as an entry point (interface) to GPU memory, (2) user-level daemon as a background service to orchestrate GPU memory via Nvidia CUDA, and (3) a kernel module to provide kernel-user space communication between two prior modules. Netlink Socket is chosen for its full-duplex inter-process communication (IPC), which can transfer sizeable data [20]. However, according to our benchmarking results, the kernel-user solution exhibits little to no performance gains over the user-level alternatives due to its complex design with considerable software overhead.

Our ultimate GAME prototype is implemented in user space alone for flexibility, portability, and full access to the GPU library (Nvidia CUDA). The block device driver (see Section 2.2) framework, called ***nbdkit***, is employed to realize the GAME prototype mainly due to three reasons: (1) for high performance, old generic NBDs are avoided, (2) among NBD variants, *nbdkit* does not require such specific features or infrastructures as RDMA or InfiniBand, and (3) *nbdkit* is available in C language with its proven excellent performance, and it fits soundly with Nvidia CUDA API. We package GAME as a *nbdkit* plugin and make it easy to install on any Linux system, arriving at a portable user-level block device service.

B. Nvidia CUDA Memory Management

Nvidia CUDA exposes GPU memory to users in three memory addressing modes: (1) native cudaMemcpy(), (2) unified virtual addressing (UVA), and (3) unified virtual memory (UVM). From the three available methods, native cudaMemcpy() is adopted since other two modes fail to meet GAME's requirements, as explained next.

UVA permits GPU cores to access CPU memory via direct memory access (DMA) with CPU memory pinned and registered in GPU's page table. Hence, UVA serves opposite purposes and is unsuitable for GAME.

UVM is designed to enable GPU memory virtualization and reduce programming complexity in sophisticated tasks. However, UVM abstracts the data location, rendering users to have little control over data movement. Plus, GPU virtual memory reveals low performance, management, and utilization. These restrictions render UVM less attractive for GAME implementation. On the other hand, all benchmark codes executed on the GPUs of our testbeds for comparison with compatible CPU executions, are enhanced with the CUDA runtime API of cudaMallocManaged().

Being simple, cudaMemcpy() is the most robust and reliable method as observed [5], [8], and it is also recommended as a performance-optimized choice by Nvidia's guidelines [21]. Our GAME implementation thus adopts cudaMemcpy() for high-performance data transfer between CPU memory and GPU memory.

C. Asynchronous Data Transferring

Nvidia CUDA defines memory transfer from the host to a device and from a device to the host as different independent tasks that can operate concurrently [21]. However, due to PCIe characteristics, only one outstanding transfer in each direction is achieved at a time [11]. Thus, it limits the concurrent CPU-GPU data transfer streams to two.

By default, all CUDA execution kernels are assigned to the default CUDA stream (stream #0); thus, they are synchronized and executed in the FIFO order. To exploit coarse-grain parallelism (on top of GPU's fine-grain nature), programmer must create multiple CUDA streams and launch CUDA kernels into different streams manually.

GAME manages two separate CUDA transfer streams over the PCIe interconnect (see Fig. 2) and utilizes cudaMemcpyAsync() for maximizing the bi-directional data transfer throughput of the interconnect.

D. Memory Expansion Control

GAME can be provisioned with *soft-limit* and *hard-limit* thresholds, indicating the GPU memory availability level for GAME to act accordingly. It stops allocating GPU memory if its usage reaches *soft-limit*. GAME calls for releasing its allocated memory areas back to GPU upon reaching *hard-limit*, by gradually transferring the contents staged in GPU VDRAM

to temporary files (in /var/tmp), making immediate room for GPU. On the other hand, after GPU memory usage drops below soft-limit, GAME gradually moves the page contents in temporary files back to GPU VDRAM for ensuring its high performance. The current GAME implementation and its results presented in the next section are without dynamic memory expansion control.

IV. Evaluation Methodology

Experimental evaluation has been performed on our testbed using benchmark codes to characterize GAME's performance. We conduct various experiments under both monotasking to multitasking scenarios with varying execution memory footprints. Based on the evaluation results, we have subsequent findings.

- Applications with massive memory footprints under irregular access patterns perform better on CPU than on GPU, due to naïve virtual memory management and far smaller CPU execution memory footprints to yield fewer page faults, among others.
- Data access patterns and working sets dictate the execution times of memory-intensive applications.
- For applications that are sensitive to the swap speed of different devices, a small speed difference can amplify performance gaps under those devices significantly.

A. Testbed Specification

Each of our established testbeds consists of a Dell Precision T7910 workstation equipped with: (1) Intel Xeon E5-2630 v3 8-Core 2.40 GHz CPU, (2) Samsung DDR4-2133 ECC 64 GB memory, (3) one Nvidia TITAN RTX PCIe 3.0 graphic card with 24GB GDDR6 on-board memory, and (4) one Kingston A400 SATA III 480GB solid-state drive.

Note that GAME can use at most 23.2 GB from the equipped Nvidia TITAN RTX card for swap space, since Nvidia CUDA reserves up to 860 MB GPU VDRAM for its management purposes, as stated in [21].

The workstation runs Ubuntu Server 20.04.2.0 LTS (Focal Fossa) with Linux 5.4.0 kernel and Nvidia CUDA 11.2. For system environment control, we utilize Linux's control groups (cgroups) to govern memory availability for benchmarks. Note that Linux cgroups is the mechanism employed in modern container management systems, such as Docker and Kubernetes, for low-level resource organizing.

B. Benchmark Suite and Execution Workloads

The NAS Parallel Benchmark suite 3.3.1 (NPB) from NASA Advanced Supercomputing Division [22] is used for performance evaluation. Among nine available benchmark applications in the NPB suite, five most memory-intensive benchmarks are chosen: BT (Block Tridiagonal), FT (Fast Fourier Transform), LU (Lower-Upper Symmetric Gauss-Seidel), SP (Scalar Pentadiagonal), and MG (MultiGrid). Others are CPU-intensive to have a light memory requirement, and hence, are excluded from the testing benchmarks. Except for MG, each of the chosen benchmarks' input parameters are re-configured to produce three different problem sizes: small (S), medium (M), and large (L), as listed in Table II, Table III, and Table IV, respectively. Note that MG's input parameters cannot be customized without heavy source code modifications, so we use the default MG class D parameters throughout all experiments.

TABLE III. NPB medium (M) details and footprints

Benchmarks	Problem size	Footprint (GB)
BT	600*600*600	33.9
FT	768*1024*1024	30.0
LU	650*650*650	35.2
SP	600*600*600	35.6
MG	Class D	26.5

TABLE IV. NPB large (L) details and footprints

Benchmarks	Problem size	Footprint (GB)
BT	680*680*680	49.3
FT	1024*1024*1024	40.0
LU	750*750*750	54.0
SP	680*680*680	51.7
MG	Class D	26.5

TABLE V. Multitasking workload details and footprints, with the common *scout workload* existing in each mix

Mix #	Benchmark 1	Benchmark 2	Total footprint (GB)
1	BT (S)	FT (M)	53.4 (+ 23.4)
2	BT (S)	LU (S)	44.8 (+ 23.4)
3	BT (S)	SP (S)	47.9 (+ 23.4)
4	BT (S)	MG	49.9 (+ 23.4)
5	FT (M)	LU (S)	51.4 (+ 23.4)
6	FT (M)	SP (S)	54.5 (+ 23.4)
7	FT (S)	MG	46.5 (+ 23.4)
8	LU (S)	SP (S)	45.9 (+ 23.4)
9	LU (S)	MG	47.9 (+ 23.4)
10	SP (S)	MG	51.0 (+ 23.4)

For multitasking evaluation, *one scout workload* is always included under every workload mix to determine its execution time of interest, provided that the scout workload is the smallest among all. Ten groups of benchmark mixes are constructed to run, each with the same scout workload (which is the small BT, whose footprint equals 23.4 GB; see Table II), as shown in Table V. All component benchmarks in a group run concurrently, with the *completion time of its included scout workload considered as the execution time of interest of the group*, since all component benchmarks are then co-running. The total execution memory footprint of each benchmark mix ranges from 68.2 GB to 77.9 GB.

C. CPU and GPU Execution Outcomes

Benchmark executions on CPU and on GPU of our established testbeds are compared first, utilizing NPB's OpenMP version [22] for CPU workloads and the enhanced NPB CUDA version for GPU workloads [23]. Since the original NPB CUDA codes do not support virtual memory, they are enhanced with the CUDA runtime API of cudaMallocManaged() to enable data allocation and accesses on both GPU and CPU physical memory transparently when executed on the GPU.

Fig. 1(a) shows the monotasking execution time comparison between CPU and GPU for Benchmark SP over a range of problem sizes. For small problem sizes when GPU memory can accommodate the execution working set entirely,

GPU enjoys vastly better performance than CPU. As the problem size rises to near the GPU VDRAM size (of 24GB), however, GPU performance drops quickly to become much inferior to CPU performance. Meanwhile, CPU exhibits gradually performance degradation when the input size increases, as depicted in Fig. 1(a). The comparative performance outcomes measured in the averaged execution operation rate (operations per second or OPS) for Benchmark SP are illustrated in Fig. 1(b). Naturally, a higher rate is better. From the figure, the CPU execution of SP is seen to be superior to its GPU counterpart for the problem size exceeding 485.

Various other execution scenarios on CPU are also compared with those on GPU, with their comparative outcomes shown in Fig. 3, where the Y-axis denotes normalized outcome values (with respect to the CPU execution times) and the X-axis gives multi-tasking workloads. The detailed mixes of those six workloads shown in the figure are listed in Table VI. The workload mixes involve benchmarks with myriad problem sizes and thus various aggregate execution memory footprint sizes. For example, Workload W2 involves 9 concurrent instances of BT with the small problem size of Class C. When executed on CPU, W2 exhibits a very small memory footprint, whereas its execution on GPU has an excessively large memory footprint. Similarly, Workload W5 involves 5 concurrent instances of BT with the small problem size of Class C, plus 2 concurrent instances of LU with the Class D problem size, to yield the aggregate memory footprint size of 21.2 GB when executed on CPU versus the much larger aggregate footprint size of 69.3 GB for GPU execution. In all, those six multi-tasking workload mixes exemplify three execution scenarios: (1) one single monolithic benchmark (i.e., W1), (2) multiple concurrent instances of one benchmark (i.e., W2, W3, and W4), and (3) multiple concurrent instances of different benchmarks co-existing (i.e., W5 and W6). All evaluation experiments are conducted on our testbeds with the host memory set to 64 GB.

From Fig. 3, we observe that the execution of an application with a large problem size (W1) can be inferior on GPU when comparing with on CPU, as also revealed in Fig. 1. For multi-tasking scenarios (W2 - W6), CPU executions are always much faster (by a factor of 3 or more) than its GPU counterparts, mainly because GPU executions (1) exhibit considerably bigger aggregate memory footprints, (2) are subject to far more page faults, each of which may take some 50 μs, because of relatively smaller GPU on-board memory, and (3) suffer from inadequate virtual memory support, as was stated earlier [9]. Note that Workload W3 crashes on GPU even its aggregate memory footprint is not the biggest among all, due to the fact that more than two concurrent LU instances always fail, whereas GAME can soundly handle them on CPU. Hence, its normalized GPU execution result is absent in the figure. In addition, Workload W4 fails to complete after five days on GPU whereas it takes less than 7 hours to finish on CPU, chiefly because each SP instance has the large problem size of Class D (23.2 GB), each of which barely fits in GPU's VDRAM, making its GPU execution extremely slow.

For each multi-tasking workload comprising different benchmarks (W5 and W6), we recorded its execution time when the instance(s) of its first component benchmark finished, given that different benchmarks would have different execution

TABLE VI. NPB WORKLOAD MIXES FOR COMPARING CPU AND GPU EXECUTIONS

Work load #	Benchmarks	Problem size	# of instances	Footprint (GB)	
				OMP	CUDA
W1	SP	490×490×490	1	19.8	36.7
W2	BT	Class C	9	6.1	75.6
W3	LU	Class D	5	44.5	67.0
W4	SP	Class D	2	24.1	46.4
W5	BT	Class C	5	21.2	69.3
	LU	Class D	2		
W6	BT	Class C	2	38.9	67.0
	LU	Class D	2		
	SP	Class D	1		

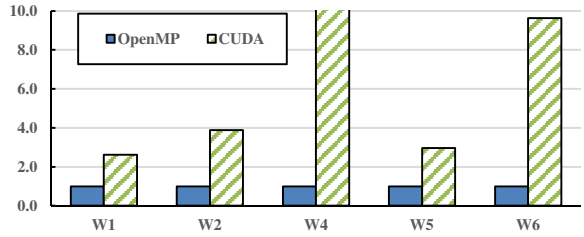


Fig. 3. Comparative results of CPU and GPU executions, with the CPU execution times normalized to 1.

times. The comparative results of W5 and W6 in Fig. 3 again signify that multi-tasking executions with large memory footprints tend to be far inferior on GPU and should be avoided, in favor of CPU executions. The result demonstrates that CPU executions under W5 and W6 are respectively 3 and 10 times faster than their GPU counterparts, since the GPU design originally intends for single task executions at a time [9] and the mix of different benchmarks with multiple concurrent instances amplifies the shortcomings of GPU executions.

It should be noted that a clear benefit with CPU executions on the testbeds is due to larger host machine memory (64 GB) than GPU VDRAM (24 GB) so that they incur negligible swap activities (see the aggregate memory footprint sizes listed under the column of OMP in Table VI), whereas compatible GPU executions are subject to heavy swap activities (due to far larger aggregate footprint sizes listed under the column of CUDA in Table VI). Swap activities tend to extend the total executive time vastly, since each of them may take some 50 μs.

The next two subsections present GAME performance results when unused GPU VDRAM available for swapping support equals 18 GB and 6 GB (out of 24 GB for the TITAN RTX GPU [4] in a testbed), respectively under monotasking and under multitasking CPU executions. If the GPU is training an LSTM ML model (adopted for our weather parameter prediction [12]), plentiful GPU VDRAM (~ 18 GB) is available for GAME use in the lengthy model training duration. On the other hand, training VGG-16 (with the mini-batch size of 128) consumes at most ~ 17.4 GB GPU VDRAM (see Section II.A), leaving about 6 GB for GAME use.

D. Monotasking CPU Executions

We have experimented the monotasking executions of five chosen NPB benchmarks on the testbed with its host memory sized at 32 GB (via the cgroup controller). The workload sizes of the five benchmarks are listed in Table IV, and they

are to experience considerable page swapping activities in the course of their CPU executions. Performance measures of interest include: (1) swapping activity and (2) execution time. Swapping activities are reflected by the swap-out and swap-in counts of the virtual memory subsystem (obtained by vmstat), with a larger count indicating more frequent data transfers between host memory and swap space. An application takes longer to run when its execution involves more swapping activities. While GPU is executing its ML workload, GAME borrows its unused VDRAM to expand host memory on demand. Depending on concurrent running GPU workloads, GAME can obtain different amounts of GPU VDRAM to support its swapping for accelerated CPU executions. When GPU VDRAM available for GAME support during job executions is short of what is required to accommodate swap-out pages, the swap partition of SSD will be involved.

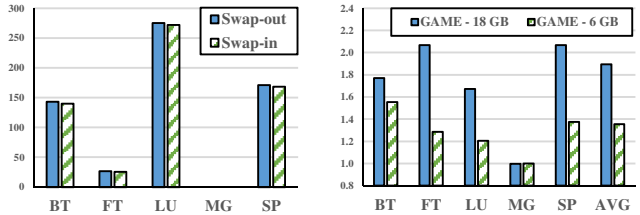
The performance results of memory-intensive benchmarks are depicted in Fig. 4. The swap activity counts generally depend on (1) the workload memory footprint size and (2) the data use profile during execution. For example, an application may allocate large memory (footprint) initially but does not require all data in its execution onset [13]. This scenario is evident for FT, which exhibits low swapping (see Fig. 4(a)) even with a sizeable memory footprint, as listed in Table IV.

The large memory footprint and execution profile of LU hike swap activities to exceed 250 million, whereas SP and BT have modest swap activities. On the other hand, FT and MG experience low swapping, especially MG with its 26.5 GB memory footprint, which is smaller than the host memory size of 32 GB. Hence, MG exhibits virtually no swapping during its execution, to yield no execution time reduction versus that of the baseline without GAME support, as shown in Fig. 4(b), where the execution time results are normalized with respect to those under the baseline. GAME is seen to achieve the execution speedup by 2.1 times (or 1.6 times) versus the baseline execution time when 18 GB (or 6 GB) of unused GPU VDRAM is available to expedite LU execution, where swap activities are highest among all five benchmarks. GAME enjoys the average execution speedup of 1.9 times across all benchmarks examined when plentiful GPU VDRAM (18 GB) is available for swapping use. Although the execution speedup drops if available GPU VDRAM becomes limited (at only 6 GB), GAME still has the impressive mean speedup of 1.4×.

Monotasking evaluation results confirm that CPU memory expansion into GPU memory on demand through GAME is effective under scientific workloads with big memory footprints and without regular data access patterns.

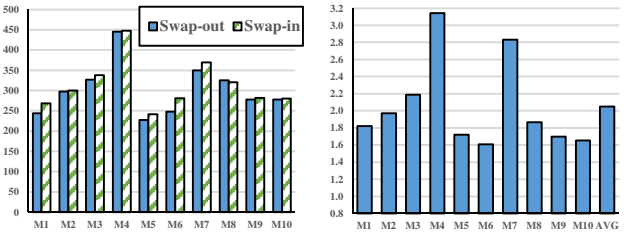
E. Multitasking CPU Executions

Execution results under multitasking workloads are depicted in Fig. 5, where workload details are provided in Table V. Ten workload mixes, each obtained by pairing two benchmarks plus one small benchmark as the scout (i.e., the small BT with its memory footprint equal to 23.4 GB, as listed in Table II) for a total of three workloads to run simultaneously, represent various execution profiles with complex memory access patterns for evaluating GAME



(a) swap activity (in millions) (b) normalized execution speedups

Fig. 4. GAME monotasking execution results under two GPU VDRAM sizes for swapping use.



(a) swap activity (in millions) (b) normalized execution speedups

Fig. 5. GAME multitasking execution results under 18 GB GPU VDRAM for swapping use.

effectiveness. The three component benchmarks of each workload mix execute concurrently on the host machine with 32 GB memory. The aggregate footprint of every mix (sized from 68.2 GB to 77.9 GB) surpasses the sum of system memory (sized 32 GB) and GAME support memory from the attached GPU (sized 18 GB), thereby involving the swap partition of SSD in its execution as well.

As expected, swap activities as seen in Figure 5(a) are mostly far higher than those under monotasking workloads, resulting from larger aggregate footprints. This is possibly due to two reasons: (1) each workload is individually smaller, with its memory footprint listed in Table II (for a small one) or Table III (for a medium one), and (2) the execution profile of each component benchmark tends to have a different memory access pattern and thus to peak its memory requirement at different points of time during the course of execution, likely to let multitasking benchmarks better share host memory.

Fig. 5(b) reveals that workload mixes which enjoy bigger execution speedups, tend to have bigger aggregate memory footprints and higher swap activities (see Fig. 5(a)). This is because an execution is accelerated more when it incurs more swap activities, which are staged at faster GPU VDRAM under GAME support versus at SSD without such support (as the baseline for speedup measurement). GAME results in large execution speedups under the high swap activities of Mix3, Mix4, and Mix7 (ranging from 320 million to 450 million), with Mix4 to have its largest speedup exceeding 3.1×. Overall, a mean speedup of 2.1× (to cut down the execution time by more than one half) is achieved across all ten workload mixes examined, signifying that multitasking executions can benefit profoundly from GAME even for huge aggregate memory footprints (up to some 78 GB; see Table V).

V. CONCLUSION

Off-loaded GPU executions are known to be advantageous for parallelized tasks without large memory footprints and with regular data access patterns, like ML and DM applications. This work has observed that applications with large memory footprints or irregular data accesses can favor CPU executions instead. An effective approach has been pursued to let memory-intensive applications run on the host machine CPU with its memory expanded on demand onto available GPU on-board DRAM, dubbed GAME (GPU-assisted memory expansion). Our GAME approach permits both monotasking and multitasking memory-intensive executions on CPU to harvest unused GPU VDRAM as swapping space for speedier execution. The GAME prototype, implemented in user space, employs a network block device driver, called *nbdkit*, to make GPU memory house fault-memory pages transparently, without any modification to application codes upon executed on CPU for high usability and portability. Our evaluation results of memory-intensive applications from the NAS-NPB benchmark reveal that CPU executions with an aid of GAME under big workloads can be more than one magnitude faster versus their compatible GPU executions, confirming the potentials of substantial advantages of CPU executions on server-grade machines equipped with GPU accelerators. In addition, GAME can achieve considerable execution speedups (by up to 3.1×) in comparison to its baseline counterpart without GPU VDRAM for swapping, if multitasking execution memory footprints exceed the host machine memory size. With larger CPU memory, which is less expensive and easily expandable, than GPU VDRAM, the host machine is preferred for application executions over off-loading them to GPUs, if the execution memory footprints approach or exceed the GPU VDRAM size.

REFERENCES

[1] J. R Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?," in ACM Queue, vol. 6, no. 2, March 2008, pp. 42 – 53. DOI: https://doi.org/10.1145/1365490.1365500

[2] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," in Computing in Science & Engineering, vol. 12, no. 3, May 2010, pp. 66 – 73. DOI: https://doi.org/10.1109/MCSE.2010.69

[3] NVIDIA Corporation, "NVIDIA Ampere Architecture Whitepaper," September 2020. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/rtx-30-series-ampere-architecture-whitepaper-download/

[4] NVIDIA Corporation, "Introduction to the NVIDIA TURING GPU Architecture," 2018. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[5] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects," in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, June 2020, pp. 1633 – 1649. DOI: https://doi.org/10.1145/3318464.3389705

[6] C. Huang, G. Jin, and J. Li, "SwapAdvisor: Push Deep Learning Beyond the GPU Memory Limit via Smart Swapping," in Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 2020. DOI: https://doi.org/10.1145/3373376.3378530

[7] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based GPU Memory Management for Deep Learning," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 2020. DOI: https://doi.org/10.1145/3373376.3378505

[8] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory," in Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19), June 2019, pp. 224 – 235. DOI: https://doi.org/10.1145/3307650.3322224

[9] B. Pratheek, N. Jawalkar, and A. Basu, "Improving GPU Multi-tenancy with Page Walk Stealing," in Proceedings of 27th IEEE Int'l Symposium on High-Performance Computer Architecture (HPCA '21), March 2021.

[10] NVIDIA Corporation, "NVLink and NVSwitch – The Building Blocks of Advanced Multi-GPU Communication," 2016. [Online] Available: https://www.nvidia.com/en-us/data-center/nvlink/

[11] PCI-SIG, "PCI Express Base Specification Revision 5.0," 2019. [Online] Available: https://pcisig.com/specifications

[12] Y. Zhang, X. Yuan, S. Kimball, E. Rappin, L. Chen, P. Darby, T. Johnsten, L. Peng, B. Pitre, D. Bourrie, and N.-F. Tzeng, "Precise Weather Parameter Predictions for Target Regions via Neural Networks," submitted to European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases for possible presentation.

[13] P. Srinuan, X. Yuan, and N. Tzeng, "Cooperative Memory Expansion via OS Kernel Support for Networked Computing Systems," in IEEE Transactions on Parallel and Distributed Systems (TPDS), vol. 31, no. 11, June 2020, pp. 2650 – 2667. DOI: https://doi.org/10.1109/TPDS.2020.2999507

[14] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating GPU Memory Consumption of Deep Learning Models," in Proceedings of 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, November 2020. DOI: https://doi.org/10.1145/3368089.3417050

[15] E. Can, R. Arora, and P. Chitale, "Profiling and Optimizing Deep Neural Networks with DLProf and PyProf," Nvidia Developer, September 2020. [Online] Available: https://developer.nvidia.com/blog/profiling-and-optimizing-deep-neural-networks-with-dlprof-and-pyprof/

[16] Z. Guz, H. Li, A. Shayesteh, V. Balakrishnan, "NVMe-over-fabrics Performance Characterization and the Path to Low-overhead Flash Disaggregation," In Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR '17), May 2017, pp. 1 – 9. DOI: https://doi.org/10.1145/3078468.3078483

[17] Richard Jones, "Better Loop Mounts with NBD Take Your Loop Mounts to the Next Level with nbdkit," in Proceedings of Free and Open source Software Developers' European Meeting (FOSDEM), February 2019.

[18] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems," in Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13), June 2013, pp. 1 – 1. DOI: https://doi.org/10.1145/2485732.2485740

[19] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Drivers," O'Reilly Media, Inc., February 2005.

[20] P. Neira‐Ayuso, R. M. Gasca, L. Lefevre, "Communicating between the Kernel and User‐space in Linux Using Netlink Sockets," Software: Practice and Experience, August 2010.

[21] NVIDIA Corporation, "CUDA Toolkit Documentation - v11.3," May 2021. [Online]. Available: https://docs.nvidia.com/cuda/index.html

[22] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," in NASA Ames Research Center, Moffett Field, CA, USA, December 1995.

[23] J. Dümmler and G. Rünger, "Execution Schemes for the NPB-MZ Benchmarks on Hybrid Architectures: A Comparative Study," in Parallel Computing: Accelerating Computational Science and Engineering (CSE), vol 25, pp. 733 – 742. DOI: 10.3233/978-1-61499-381-0-733.