Cluster Computing
https://doi.org/10.1007/s10586-022-03805-x

Improving Oversubscribed GPU Memory Performance in the PyTorch Framework

Jake Choi[1] · Heon Young Yeom[1] · Yoonhee Kim[2]

Received: 28 April 2022 / Revised: 2 October 2022 / Accepted: 25 October 2022

Abstract
Popular deep learning frameworks like PyTorch utilize GPUs heavily for training, and suffer from out-of-memory (OOM) problems if memory is not managed properly. CUDA Unified Memory (UM) allows the oversubscription of tensor objects in the GPU, but suffers from heavy performance penalties. In this paper, we build upon our UM implementation and create and utilize a minimal overhead CUPTI dynamic profiler to trace unified memory page fault and memory transfer statistics in PyTorch applications. We also implement CUDA memory prefetch and advise API which can be called directly from the PyTorch application based on the dynamically profiled statistics to improve oversubscription performance in various PyTorch models including Resnet and BERT.

Keywords CUDA · Unified memory · PyTorch · prefetch · Advise · CUPTI

1 Introduction

Deep learning (DL) training is widely performed in graphics processing units (GPU) because of greater performance and efficiency over using central processing units (CPU) [1]. Even though each individual GPU core may not be as powerful as a CPU core, GPUs compensate by having a greater quantity of cores allowing for more parallelism. In order to efficiently utilize GPUs for computation, entire DL models and data need to be copied into the GPU memory before training begins. With increasingly larger models and sample mini-batches, this can take up a significant amount of memory [2]. However, even state-of-the-art GPUs have limited memory (e.g. 12GB for NVIDIA's Titan XP and 16GB for NVIDIA's V100 GPU) compared to host memory. Therefore, if no consideration is given to the memory usage of a DL training process in any particular framework (e.g. TensorFlow [12], PyTorch [23]), then out-of-memory (OOM) faults could occur and the entire process would fail.

In this paper, in order to rectify the OOM problem, we present a case study to investigate the effects of implementing CUDA Unified Memory (UM) [40] on the PyTorch framework. Unified memory allows the virtualization of GPU and CPU host memory to become a *single address space*, and performs background data migration from GPU to CPU host memory and vice versa when GPU memory is insufficient to store data. This allows for automatic out-of-core computation on widely-used DL frameworks with no modification to user code. As far as we know, few research in literature have investigated the specific effects of implementing CUDA UM on PyTorch. The rest of this paper is organized as follows. Section 2 provides related work, background knowledge about CUDA UM, PyTorch, and insights about our design. Section 3 discusses the experimental setup and the implementation of UM on PyTorch. Section 4 evaluates the performance of PyTorch with UM on the machines and the current limitations of the implementation. Section 5 concludes this work with future insight.

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Jake Choi
kidcoder@snu.ac.kr

Heon Young Yeom
yeom@snu.ac.kr

[1] Department of Computer Engineering, Seoul National University, Seoul, South Korea

[2] Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

Published online: 11 November 2022

2 Related Work and Background

2.1 Related Work

Several existing methods in literature are used to overcome OOM limitations by reducing memory consumption. Some methods use lower-precision floating points [3, 4] or compression [5, 6] in the parameters of the models. However, such methods influence the accuracy of the model and require lots of manual parameter tuning. Other methods involve deletion of intermediate activation tensors after the forward pass, and recomputation [7] when needed during the backwards pass, but this affects the performance of the model because of the trade-off of memory space to additional compute cycles and does not work well for large models where intermediate activation tensors cannot be easily recomputed. Additionally, for both of these methods mentioned, manual intervention is needed by the programmer.

Swapping out model tensors from host and GPU memory is another technique used in recent years to reduce the memory footprint of models. vDNN [8] is a run time memory management solution, prototyped as a layer above cuDNN [41] that reduces average GPU memory usage by releasing intermediate feature maps from GPU memory if no reuse is required, or offloads them to CPU memory if further reuse exists but is not immediately required. vDNN++ [9] extends upon the previous work by performing asynchronous transfer of feature maps, additional heuristics to address memory fragmentation, and usage of compression to reduce the pinned main memory footprint. SuperNeurons [11] is a dynamic GPU memory scheduler for training deep non-linear neural networks. It uses memory techniques to dynamically analyze and offload tensors of each convolution layer of a DNN. These solutions only swap out specific activation tensors which are determined through manual heuristics, and are limited to a specific subset of the entire data residing in GPU memory. Furthermore, their implementations are all built as separate prototype frameworks used to compare against widely used production-level frameworks like Torch [13], TensorFlow, Caffe [14], or MXNet [15].

Dataflow graphs generated from DNN computation structures also provide knowledge to overlap computation with communication, allowing for minimization of performance overhead. SwapAdvisor [2] uses Genetic Algorithm [16] and a static dataflow graph with no control-flow primitives to find the optimal tensor swapping strategy. It also takes into consideration the GPU operator scheduling, and memory allocation. However it requires a static dataflow graph, which is not used by frameworks like PyTorch or TensorFlow eager mode and only works with a single GPU. ZeRO-Offload [18] is a GPU-CPU hybrid DL training library based on PyTorch that allows heterogeneous GPU and CPU training and swapping of data across memory spaces to train huge models with over 13 billion parameters on a single GPU. It uses a static dataflow graph to partition the model between the CPU and GPU devices, and requires modification of application level user code to use the library. While all of the above outlined methods optimize performance in terms of communication to computation cost, all rely on manual awareness of GPU memory usage on the part of the programmer and do not really focus on the actual framework being used.

2.2 CUDA Unified Memory

Unified Memory allows for the oversubscription of memory in GPU applications. Kernels running in the GPU can access data allocated with `cudaMallocManaged` even though the data is allocated on the host side. The order of operations that happens when such memory allocated on the CPU is accessed by the GPU is the following: 1. Allocate new pages on the GPU 2. Unmap old pages on the CPU 3. Copy from the CPU to the GPU 4. Map new pages on the GPU 5. Free old CPU pages. When Pascal and Volta GPUs access a non-resident page, the GPU generates a fault message and locks the translation lookaside buffer (TLB) for the corresponding streaming multiprocessor, which stalls any future translations until all page faults are resolved [21]. Duplicate fault messages for the same page can occur forming a *page fault group*. Driver fault handling to process and remove duplicate page faults, update CPU and GPU mapping and transfer data takes a lot of overhead. Despite the added benefit of memory over-subscription and elimination of explicit programmer effort, UM has been criticized as being slow due to excessive page fault handling [20]. Unified Memory tested on a set of different benchmarks like CUDA SDK's Diffusion3D Benchmark, Parboil Benchmark Suite and Matrix Multiplication ported on UM showed an average performance loss of 10% [22].

2.3 Choosing PyTorch as the Framework

PyTorch is a relatively new Python library that is popular in the research community, and is growing fast. As of the time of this writing, its main competitor is Tensorflow, and both frameworks are the most popular frameworks being used for deep learning. PyTorch performs immediate execution of dynamic tensor computations with automatic differentiation and GPU acceleration. Other frameworks

like Caffe, Tensorflow, or Theano [24] construct a static dataflow graph that represents the complete computation and is then repeatedly applied to batches of data. The static dataflow approach may be used to increase performance and scalability, but comes at the cost of ease of use, ease of debugging, and flexibility on the types of computation that can be represented. PyTorch uses dynamic eager execution, and still retains performance comparable to the fastest deep learning libraries. It is a lot easier to make flexible custom models that would be harder to express in other frameworks, and many inputs can be flexibly changed during runtime. Currently, Tensorflow has released eager execution mode on Tensorflow 2.0, making it more like PyTorch. However, programmers would have to change a lot of the existing Tensorflow 1.x code in order to shift to the newer versions, which leads to backward compatibility issues. Tensorflow has more production level, industry use cases where implementation speed is more of a factor than flexibility. Generally speaking, PyTorch is more research-oriented, Python-friendly, intuitive, and easy to learn than the original Tensorflow. The amount of research citing the PyTorch framework has grown a lot in the past few years and is still currently growing at a fast pace. These are the reasons why we chose PyTorch as the framework to implement CUDA Unified Memory.

Currently in research, we have not seen CUDA UM implemented in PyTorch. The main reason for this is that even though UM provides increased productivity for the programmer and ease of use by solving the OOM problem, it comes at the cost of heavy performance overheads. Therefore other explicit methods of memory management that we already outlined in Sect. 2.2 are used instead. For graph neural network (GNN) use cases, a unified tensor implementations for PyTorch exist [26]. Such work adds certain new functions to the PyTorch framework to allow programmers to declare tensor objects as a "unified" object. However, this object is not truly using CUDA UM in the form of a unified virtual memory space because of performance reasons. Graph objects have poor temporal and spatial locality, and therefore the "unified" object in this case is using zero-copy memory where the object is pinned in host memory and directly mapped to the GPU device. This is because certain graph tensors are not accessed frequently even though they make take up a lot of memory space. Additionally, the programmer has to be aware of such objects and declare them manually in the code as unified objects using a different syntax in order to take advantage of this feature. This approach is not what we are aiming for in this paper. Our goal is to allow the programmer to be completely unaware of such memory management details, and still not encounter OOM problems while not sacrificing too much performance by utilizing CUDA UM in the PyTorch framework itself.

2.4 Design Differences with Other Frameworks

OC-DNN [25] is an out-of-core DNN framework that takes advantage of the CUDA UM features in Pascal and newer architecture GPUs. It uses UM communication primitives and optimizations like prefetching and advising to avoid GPU page faults in the Caffe framework. It provides comparative performance for regular DNNs which fit into GPU memory, and provides a 1.9x speedup compared to the pre-existing out-of-core methods that do not utilize UM for the Caffe framework. The implementation also removes over 3,000 lines of redundant Caffe code that must be changed as CUDA UM is implemented instead of explicit memory copies. However this work implements UM on the Caffe framework based on C++, which has largely been inactive for the past couple of years. Caffe is also limited in the types of deep learning models that it can effectively deploy. In fact, a successor, Caffe2 [38] was built on top of the original Caffe in order to increase support for more non-vision use case models, distributed computation, and mobile deployment. It also merged with the PyTorch framework in 2018. Now, the original Caffe has largely been deprecated after the merge. PyTorch now holds the entire Caffe2 code base and has the additional benefits of being more flexible with prototype models and is more research-oriented. Therefore it makes sense that we focus on the latest DL framework.

In the design of OC-DNN, file access (obtaining the training samples from disk) is optimized by replacing all the file-to-host (F2H) buffer transfers and host-to-GPU (H2D) transfers with a single unified file-to-managed (F2M) transfer. OC-DNN converts D2D copies that occur during the layer stage using the *Layer* class to UM accesses. OC-DNN improves upon these D2D data dependencies between kernels by using prefetching and advising to evict source buffers in the forward pass to host memory and prefetch data back in the backward pass to reduce GPU faults. Such optimizations can also be applied to PyTorch to work for intermediate data, but would be more complex to implement in the backend CUDA kernel management. Finally, Caffe uses `forward_cpu()`, `backward_cpu()`, `forward_gpu()`, `backward_gpu()` member functions to deal with CPU and GPU training separately. OC-DNN unifies the functions into one type and removes redundant code. Because PyTorch uses Python, the syntax is more general (e.g. `.to(''cuda'')`), and the framework is more complex and automated in dealing with heterogeneous architectures. Therefore the modifications required are less straightforward.

Choi et al. [31] performs similar work to this paper where they implement Unified Memory in the PyTorch framework. They the implemented memory advise and prefetching in the back end with a simple Python interface

to memory advise the intermediate results, input data tensor, and model parameters of a PyTorch model. Therefore they classify all PyTorch data into those three types, and then perform automatic prefetching and memory advise for those models based on when the PyTorch tensors call the `.to()` function on a single activated GPU. Our approach is more fine-grained, and we allow complete control over every single GPU allocated object, and provide the API for the programmer to use memory prefetch and advise at any place in the PyTorch application code. In addition to this, we support memory optimization techniques for multiple GPUs, and perform dynamic profiling to measure the amount of GPU page faults and data transfers from host to GPU generated by each individual GPU allocated tensor.

Tensorflow Huge Model Support (HMS) is a library designed to speed up huge model training on unified memory. It analyzes the Tensorflow computation graph, and implements group execution and prefetch by editing the graph automatically [34, 35]. This does not apply to PyTorch because PyTorch uses dynamic computation graphs. Also, our approach allows the programmer to have control over prefetching and memory advise, instead of automatically performing only prefetching and group execution on a computation graph. PyTorch-UVM-GPT2 [45] is another related work that evaluates the performance of PyTorch-UVM with large-scale language models like GPT-2 and GPT-3. PyTorch-UVM is an implementation of CUDA UM on PyTorch. However, this work does not use memory optimization techniques like prefetching and advise with tensor information to speed up the performance of large models on PyTorch-UVM.

This paper extends upon and adds functionality to the Unified Memory implementation in [36]

3 Implementation and Experimental Setup

PyTorch framework consists of Tensor libraries programmed in C++, Python and CUDA. These libraries store and operate on tensors, which are multidimensional rectangular arrays of numbers. Application programs must import the torch library to user the PyTorch functionality. Figure 1 shows a simplified view of the PyTorch architecture. We implemented CUDA UM in the PyTorch framework backend itself (details outlined in Sect. 3.1). Then we added functionality to the PyTorch API to be able to call CUDA optimization functions like `cudaMemPrefetchAsync` and `cudaMemAdvise` to the CUDA API using PyTorch CUDA/C++ Extensions (Sect. 3.2). Finally, we added functionality to profile tensor statistics and bring forth the information to the application level using CUPTI (Sect. 3.4). Our implementation also

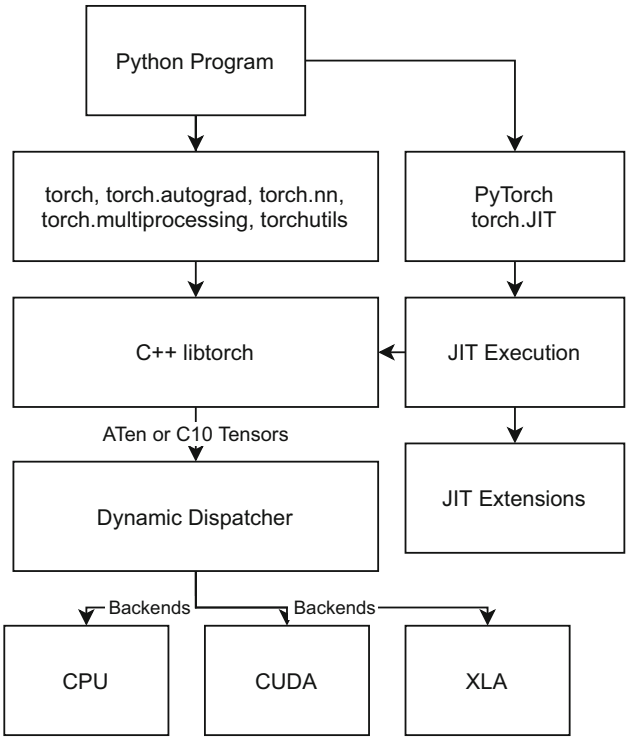


Fig. 1 Brief PyTorch Architecture

adds support for Multiple GPUs (Sect. 3.3). Figure 3 shows the modified PyTorch architecture. Our procedure (shown in Figure 2) of combining dynamic tensor profiling with memory optimization works generally as follows:

1. Insert CUPTI profiling markers at beginning and end of deep learning training loop for any workload.
2. Insert prefetching operations and memory advise at heuristically determined points in code where potential improvements can be made
3. Run training loop for 2 to 10 iterations in the profiling phase.
4. Once profiling data is collected, organize the data and sort based on largest amounts of GPU faults.
5. Dynamically activate the memory optimization on the specific tensors that have the greatest amount of GPU page faults or or eviction transfers based on profiled data.
6. Continue executing the training loop.

3.1 CUDA Unified Memory Implementation

We implemented CUDA UM in the core c10 CUDA library inside the PyTorch framework. C10 is similar to ATen (tensor mathematical operator library), but contains more recent core PyTorch code. We specifically modified the CUDA caching allocator in the CUDA backend of the c10 library. The caching allocator is used to speed up

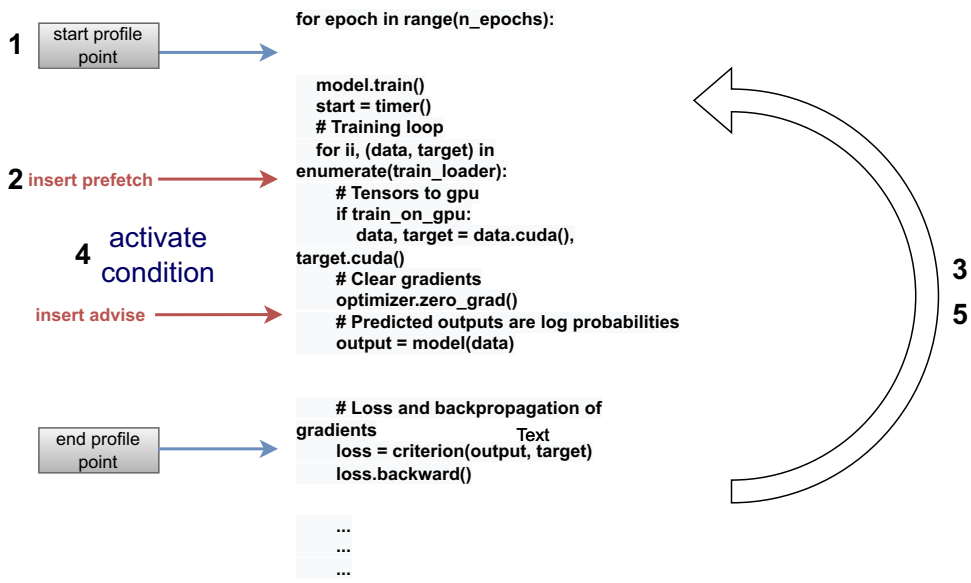


Fig. 2 Implementation Procedure

memory allocations and also allows fast memory deallocations without device synchronizations. We specifically replaced `cudaMalloc` calls in the block allocator with `cudaMallocManaged`. This ensures that all GPU device allocations will be made using unified memory.

Then we ran the test program shown in Listing 1 using Python to test if there would be an OOM error. The program creates a 2D tensor filled with 16 GB of random 4-byte floating point numbers in the CPU host side, and then sends a copy to the GPU side. Then it calculates the square of each number on both the CPU tensor and GPU tensor and tests for correctness. In our results, the two tensors had equal results to each other. Listing 1 results in an OOM error in the unmodified case if the GPU does not have enough memory to accommodate 16 GB of floating point data.

Simply modifying the GPU device-side memory allocation mechanism to perform UM allocations is an incomplete implementation. The PyTorch framework is designed to be flexible by utilizing a dispatcher mechanism to allow several backends (CPU, GPU, XLA, or other devices) and data types to be used for identical operators, through function tables, and dispatcher keys used to distinguish between different overloaded versions of the same function. This allows the flexibility of Python application code to work the same across CPU and GPU, with minimal programmer effort. In order to be a more complete implementation of UM, tensors allocated on the CPU backend also need to use `cudaMallocManaged` from the CUDA library and explicit memory copies from device to host or vice versa need to be eliminated. Currently, when PyTorch functions like `.to(''cuda'')` are used, copies of the same tensor data are maintained on both host and GPU, leading to redundant host memory usage on x86 systems when GPU memory is oversubscribed. We will deal with these issues in future work.

```
import torch
x = torch.rand(100000,40000)

y = x.to("cuda")

y.pow_(2)
x.pow_(2)

print(torch.equal(x,y.to("cpu")))
```

Listing 1 Example code to oversubscribe GPU memory on PyTorch

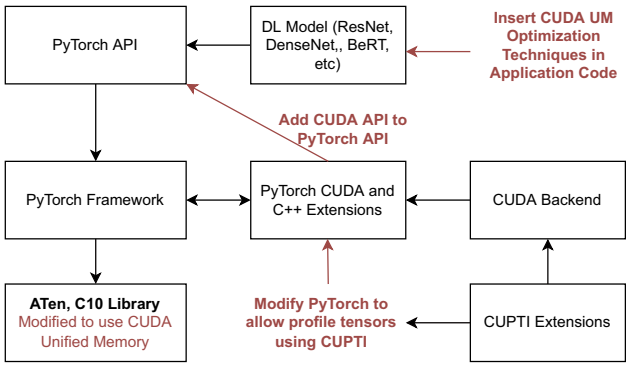


Fig. 3 Modified PyTorch Architecture

When profiling the results from Listing 1 using *nvprof* [42] after making the UM modifications to PyTorch code, we notice that OOM errors disappear even though GPU memory usage is maximized by checking `nvidia-smi`. Figure 4 shows a simplified diagram of what NVIDIA Visual Profiler [43] outputs when Listing 1 is executed. In

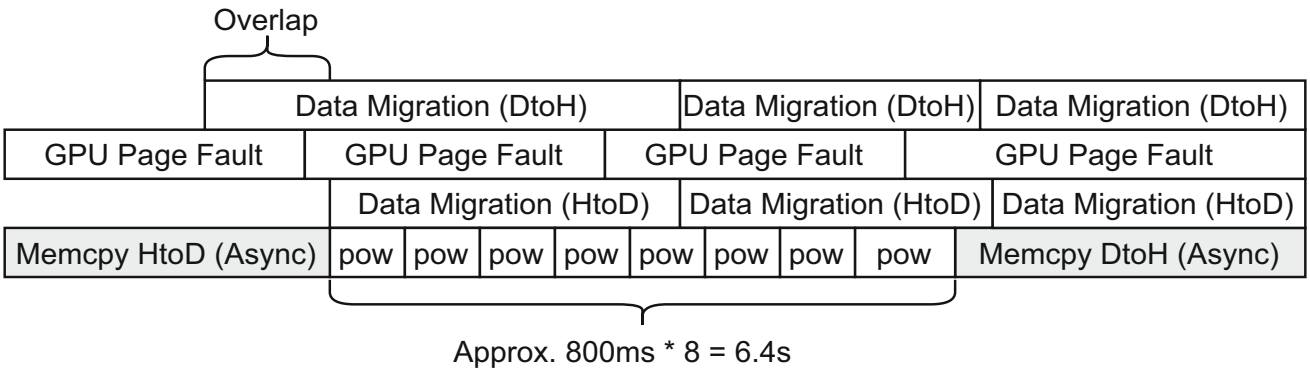


Fig. 4 Profiling UM page faults with *nvvp* (16 GB data oversubscribed)

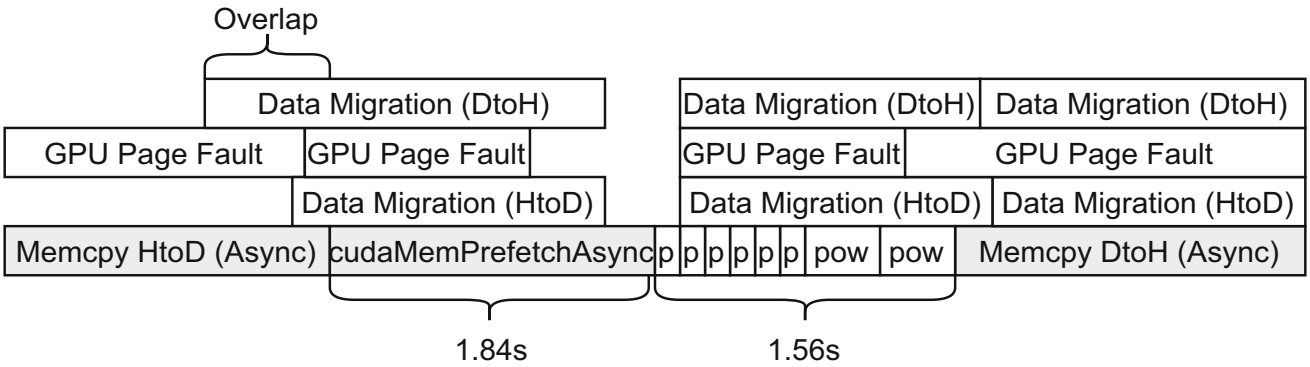


Fig. 5 Profiling UM after prefetch optimization (16 GB data oversubscribed)

the actual profiled results, 18 additional PyTorch CUDA kernels are also executed, but have very low significance that they are omitted from Figure 4. The `cudaMemcpyAsync` operation from host-to-device overlaps with the CUDA driver page faults and data migration caused by UM. This is because data explicitly copied to the device exceeds available device memory. In this situation, the CUDA driver forces the data that is least recently copied to be sent back to host memory due to oversubscription. After memory copy is complete, the main mathematical operation kernel (PoW) is executed. Page faults and data migration are shown in bars above the kernel stream. Each bar represents groups of many page faults or data migrations occurring during that period.

When tensor sizes are larger than a certain amount (in this case about 2 GB), PyTorch automatically divides the vectorized kernel into smaller chunks by invoking a loop to call the CUDA kernel repeatedly to process different parts of the tensor. In this case the PoW kernel is invoked a total of 8 times, and each invocation takes approximately 800 ms in duration on average. During this period, large amounts of background GPU page faults occur causing more delay in kernel execution time. When the tensor was able to completely fit into GPU device memory by reducing the data size from 16 GB to 1.6 GB, a single PoW kernel executed and had a duration of about 7 ms. This means that if GPU device memory is insufficient, there is an additional overhead of roughly 6330 ms due to GPU page faults.

When PyTorch creates CUDA kernels that access individual elements in a tensor, it does not consider the physical location of the accessed data. If this information is known advance, UM optimization techniques can be applied like prefetching using `cudaMemcpyAsync` or `cudaMemAdvise`, in order to provide preferences for data placement in certain address ranges. In the Listing 1 example, the PoW kernels access the tensor data in a sequential manner from the beginning. However data has already been migrated to host memory due to oversubscription. Therefore we prefetched a portion of the data to the GPU device before launching the kernels (shown in Figure 5), and were able to save about 3 seconds in total execution time. The time spent in kernels is reduced from 6.4s to 1.56s and additional overhead from `cudaMemPrefetchAsync` to fetch data from the host side is 1.86s. Page faults do not occur when the first few PoW kernels are executed because data already resides in device memory after the prefetch. Early kernels took as little as 9

Table 1 List of API Functions

Function Name	Parameters	Description
getList	N/A	Retrieve the list of tensor information
getSortedData	N/A	Retrieve data sorted on order of GPU page faults
prefetchA	Address,bytes, device	Identical to `cudaMemPrefetchAsync` in Python
adviseA	Address,bytes, device,type	Identical to `cudaMemAdvise` in Python
profile_start	N/A	Start CUPTI Profiling of Unified Memory Access Data
profile_stop	N/A	Stop CUPTI Profiling of Unified Memory Access Data and Perform Count[1]
profile_register	N/A	Initialize and Register CUPTI Profiling

[1] Perform count is a separate function that organizes the CUPTI data, outlined in Sect. 3.4

ms to execute. Kernels that accessed the latter portion of the data took longer because they had to evict existing device GPU data and replace it with newer data from the host. Overlapping prefetching with kernels and removing explicit copies which wrongly send immediately needed data to host memory can further optimize such scenarios.

3.2 Prefetch and Advise API Functionality

Memory optimization techniques like `cudaMemPrefetchAsync` and `cudaMemAdvise` are techniques used to improve the performance of CUDA code when memory is oversubscribed. However, it would be cumbersome to implement the techniques in the backend CUDA portion of PyTorch because such memory optimization techniques can be arbitrarily placed at any point in code between CUDA kernels resulting in different performance results, and each PyTorch application would have different circumstances as to where such code points are. Therefore, instead of hard coding the optimizations into the backend, we created an API that allows the PyTorch user to call such memory optimization techniques whenever they want to perform such optimizations in the user application Python code.

The user needs to know the **address**, **size**, and **GPU device ID** of memory allocated in the GPU in order to directly utilize the CUDA memory optimization functions, because they are the required parameters. However the user has no knowledge of such parameters unless the information can be retrieved from the PyTorch backend. We solve this by managing the tensor information when they are allocated in the CUDA caching allocator portion of the C10 library. Because the GPU tensors that are used by PyTorch are all allocated using `cudaMallocManaged`, we store the required parameters when they are allocated into a separate vector of arrays. The address stored is a virtual address that can be accessible in both host and GPU because of unified virtual addressing (UVA) by the UM driver already taking care of this. In order to add the API to the existing PyTorch framework, we utilize PyTorch CUDA and C++ extensions [44] to add the API functionality so that when we call the functions in the Python application, they can translate appropriately into the backend CUDA functions.

The API functions listed in Table 1 are defined and implemented to allow the user to perform memory optimization techniques in PyTorch applications. The user can obtain the entire list of tensors which is allocated in the PyTorch backend during the memory allocation of the GPU using `getList`. In addition to this, the sorted list, which contains the list of tensors in descending order of GPU page faults, based on the CUPTI profiling results (Sect. 3.4) can also be retrieved using `getSortedData` so that the user can perform fine-grained memory optimization based on the dynamically profiled data. The key functions of this paper `prefetchA` and `adviseA` are both equivalent to and used to perform the `cudaMemPrefetchAsync` and `cudaMemAdvise` functions within the user application for the GPU tensors, whose address information can be obtained by using the list retrieval functions. We can set the type of memory advise equivalent to the CUDA API with three major types, `ReadMostly`, `PreferredLocation`, and `AccessedBy`. We implemented these functions using the PyTorch C++ extensions framework, and prefetching uses the default CUDA stream 0. The last three functions in Table 1 are profiler functions, using the CUPTI tool [39], which is part of the NVIDIA CUDA toolkit. They are used to perform the functions outlined in Sect. 3.4

3.3 Multi-GPU Support

Unified memory allows for the support of a multi-GPU setup. In this case, managed memory allocations are visible to all the GPUs and can migrate to any processor on-demand. Our implementation is not limited to a single GPU, and we can run larger PyTorch models on multiple GPUs utilizing UM in the event of over-subscription. Our work mentions the usage of multiple GPUs with unified memory, which other literature fails to do so. Therefore by

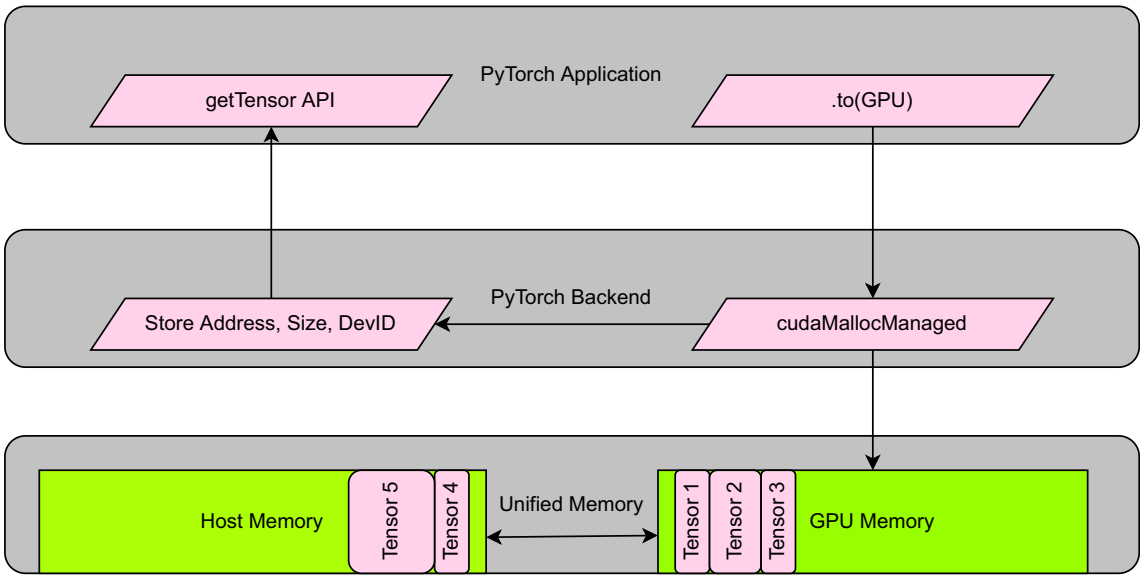


Fig. 6 Obtaining information about every single allocated tensor directly from backend

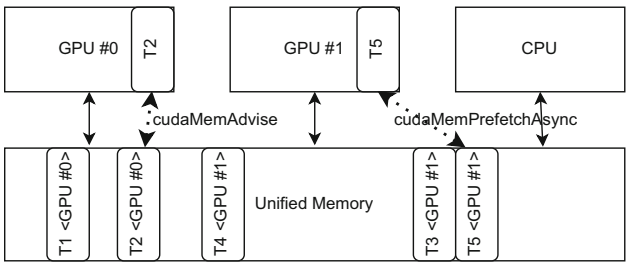


Fig. 7 Multiple GPU Support for Memory Optimization Techniques

combining multiple GPUs and unified memory, we do not have to worry about the allocation of memory to each individual device, because the UM driver will abstract the entire process. Therefore we can leverage the performance benefits of having additional GPUs along with their memory capacity, with a unified address space that includes the entire host memory. When each individual tensor is allocated in the backend PyTorch implementation using `cudaMallocManaged`, we also record the device ID that is performing the allocation by obtaining the current CUDA device using `cudaGetDevice` and store it in our list of tensors. This is possible because the PyTorch backend in data parallel mode divides tensors quite cleanly by default and allocates GPU memory corresponding to the specific GPU that will use that tensor when distributing tasks to separate processing units. We can simply obtain the ID of the current device that is responsible for its tensors. In addition to this, our memory optimization API outlined in Table 1 allows us to specify the **device ID** when we need to perform optimizations to specific GPU devices. Thus we can advise and prefetch tensors to their respective GPUs using the respective information from list as to which respective GPU the tensor was originally allocated in. Figure 7 shows how we can control which tensor is prefetched or advised to each individual GPU based on the device ID we obtain from the tensor list. Therefore in instances of memory oversubscription, the user can maintain control over which GPU device each tensor will be optimized to.

3.4 Dynamic Profiling of Unified Memory Access

During the course of training most models in PyTorch, varying numbers of GPU tensors are allocated. We can manage every single tensor that is allocated in each different workload by storing the tensor address, and size in a list. However, there would be no information as to what the allocated GPU object's purpose is. In addition to this, even if we knew the object's purpose, we would not exactly be able to measure how it is accessed during the course of model training. We believe that dynamic profiling using the CUDA Profiling Tools Interface (CUPTI) can achieve this purpose of measurement. CUPTI is a profiling and tracing tool provided as a dynamic library that targets CUDA applications and gives insight to the GPU and CPU behaviour. Tracing in this case refers to the collection of timestamps of activities like CUDA API calls, kernel launches, and memory copies. We can trace the Unified Memory activity of our CUDA applications by using the CUPTI Activity API, made accessible by us from the PyTorch application by the user when tracing wants to be performed.

Key (Address)	V1 (Type)	V2 (Value)
0-1000	GPU_PAGE_FAULT	0
	HTOD_TRANSFER	0
	DTOH_TRANSFER	0
1000-5000	GPU_PAGE_FAULT	0
	HTOD_TRANSFER	0
	DTOH_TRANSFER	0

- GPU PAGE FAULT @ 100
- DTOH (4MB) @ 300
- HTOD (16MB) @ 2000
- GPU PAGE FAULT @ 4000

Key (Address)	V1 (Type)	V2 (Value)
0-1000	GPU_PAGE_FAULT	1
	HTOD_TRANSFER	0
	DTOH_TRANSFER	4MB
1000-5000	GPU_PAGE_FAULT	1
	HTOD_TRANSFER	16MB
	DTOH_TRANSFER	0

Fig. 8 Adding all counter data into a managed hash table

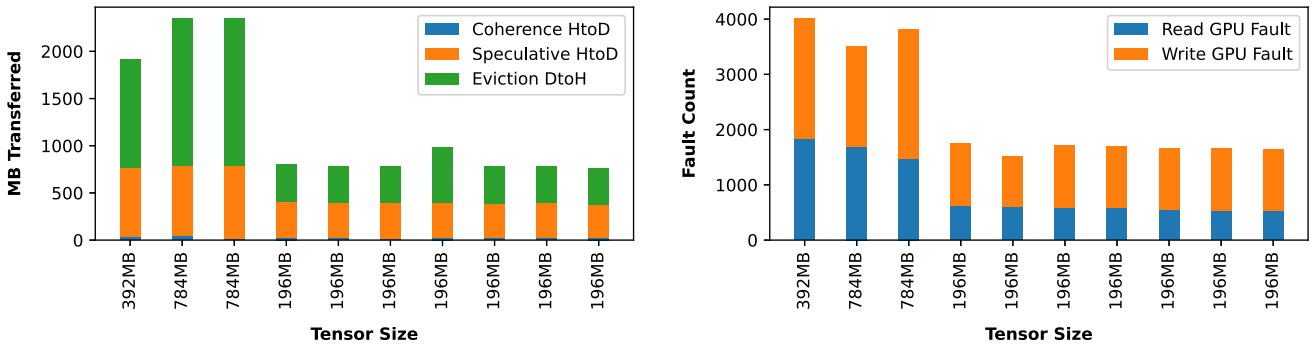


Fig. 9 CUPTI Profiling Output for Top 10 Largest Tensors in Resnet-18 Workload with Batch Size 256

In order to provide a fine-grained approach to know which GPU tensors are accessed the most, we profile the amount of GPU/CPU page faults, and the amount of host-to-GPU and GPU-to-host data transfers that take place when Unified Memory is used. Such metrics are provided by the CUPTI API by registering the `cuptiActivityConfigureUnifiedMemoryCounter` function. During registration, we provide the details about which specific Unified Memory counters we are going to profile. We use the `profile_register` function shown in Sect. 3.2 to initialize and register the CUPTI metrics and counter buffers. We can start profiling at and point of the PyTorch application code by inserting a call to `profile_start` and stop profiling by using `profile_stop` when we want to collect the metrics until that point. Once the CUPTI profiler finishes collecting all the metrics, we organize the data by creating a hash map where the **key** of each tensor is the **address** of the tensor. The metrics output data in the form of a Unified Memory counter every time an asynchronous activity happens, when data is oversubscribed in the GPU. If the address in the counter is within the range of the tensor address, we add the count of the corresponding metric value. Each time a GPU page fault occurs, we increment the number of page faults by one, and each time there is a data transfer, we add the total number of bytes that is transferred from device to host or vice versa, like shown in Figure 12. In addition to this, we also distinguish between the actual reason why there was a GPU page fault or cause of data migration.

In the case of a GPU or CPU page fault, the cause of the fault can be attributed to whether it is a read, write, or atomic memory instruction. Memory prefetch calls can also trigger a page fault. Our profiler distinguishes between read and write operations. In the case of a memory transfer operation (DtoH or HtoD), the cause of transfer in the case of DtoH is in order to make room for another block of memory on the GPU. In the case of HtoD, there are three cases: one is to guarantee the data coherence on the GPU, another is the UVM driver speculatively migrating data before being accessed by the GPU in order to increase performance, and the last is migration due to an explicit call like prefetch. Our profiler catches the speculative, and data coherence calls for the HtoD case, and we classify all DtoH operations as eviction calls.

4 Evaluation

4.1 Experimental Setup

The experimental setup is a single server machine equipped with 4 NVIDIA Titan XP GPUs, and a single home machine

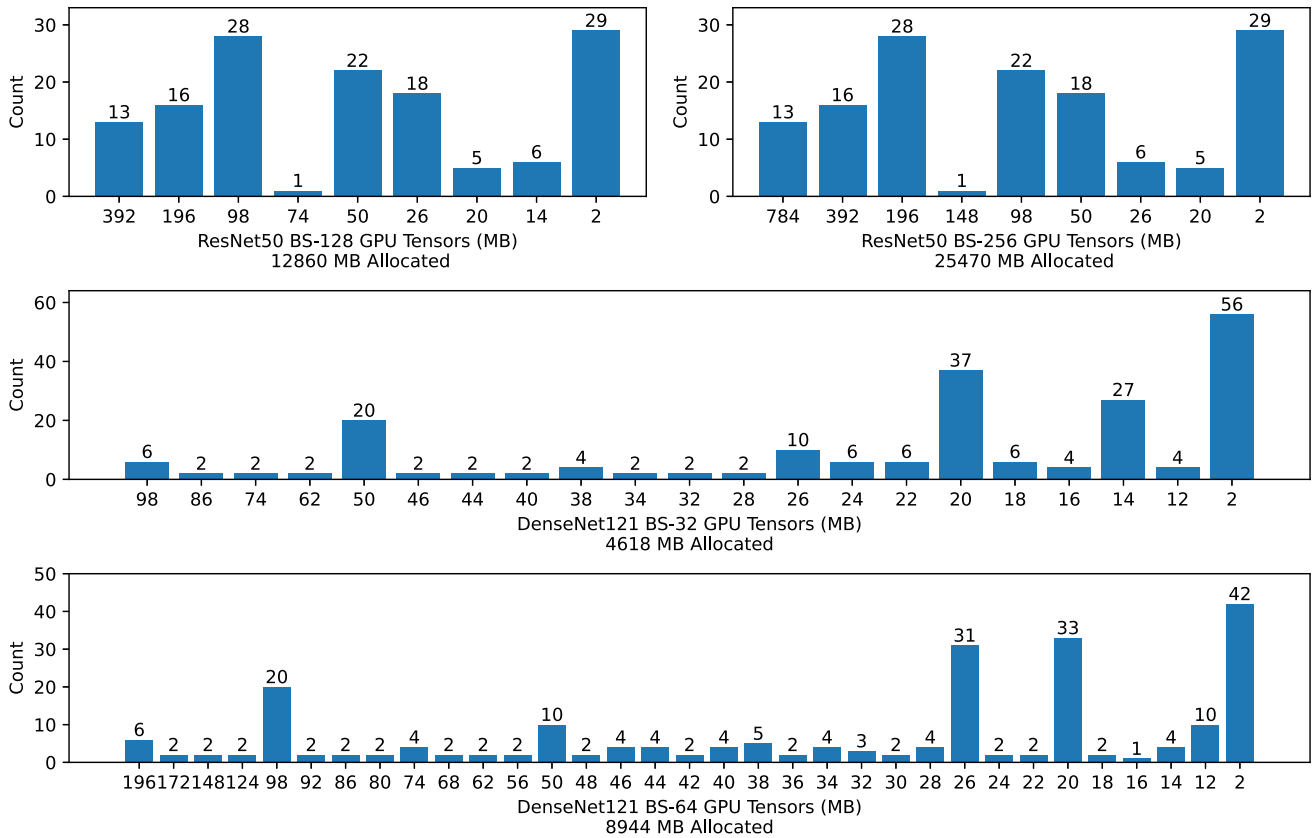


Fig. 10 PyTorch Vision Processing Workload Allocated GPU Tensor Count by Size

equipped with two GPUs, one GTX 1660 Ti and one GTX 1050 Ti. The Titan XP GPU has 12 GB, the 1660 Ti has 6 GB, and the 1050 Ti has 4 GB of memory available. The server has a total of 125 GB of host memory available. The home machine has a total of 48 GB available. We use PyTorch 1.9.0 in developer mode for implementation purposes, and also use Anaconda [37] to shift between our changes and the base case. Our workload that we run to test actual execution time in our evaluation is the PyTorch ImageNet Training example which has a lot of available DNNs for training vision like Alexnet [27], Resnet [28], and VGG [29]. The dataset used is the Tiny Imagenet 200 [30]. We also use the Transformers [32] models workload to test other deep learning model architectures in natural language processing (NLP) like Bidirectional Encoder Representations from Transformers (BERT).

4.2 Results

For the Tiny Imagenet 200 workload, we profiled the activities between the first iteration and second iteration of training by placing a `profile_start` and `profile_stop` call inside the training loop. Profiling results shown in Figure 9 showed us that there are 3 types of transfers occurring through the UVM driver, in the case of ResNet-18. The figure only shows us the results for the ten largest tensors, but there are actually 64 tensors for a total allocated memory of 8166 MB. In the case of DtoH transfers, all transfers are eviction transfers, and are done in 2MB chunks of data. They are evicted automatically by the UVM driver because of lack of space in GPU memory. In the case of HtoD transfers, the majority are speculative transfers performed to predict accesses to the data by the GPU, while a minor share comprises of actual coherence faults, needing the data to be transferred to ensure correctness. For page faults, read and write page faults are distinguished. Based on Figure 9 there is usually a greater portion of write GPU faults compared to read GPU faults. Other vision training models in the Tiny Imagenet 200 workload follow a similar pattern in terms of the ratios of the types of data transfers and page faults. We noticed that there are many tensors created with duplicate sizes, and although it is not shown in Figure 9, there are very few tensors that actually have a majority of read GPU faults. In comparison to this, there are actually a number of tensors that have only write GPU faults and no read operations. The total number of GPU tensors generated during a workload varies depending on the model type and the batch size.

Figures 10 and 11 show us the allocated GPU tensor count for each vision processing and NLP workloads,

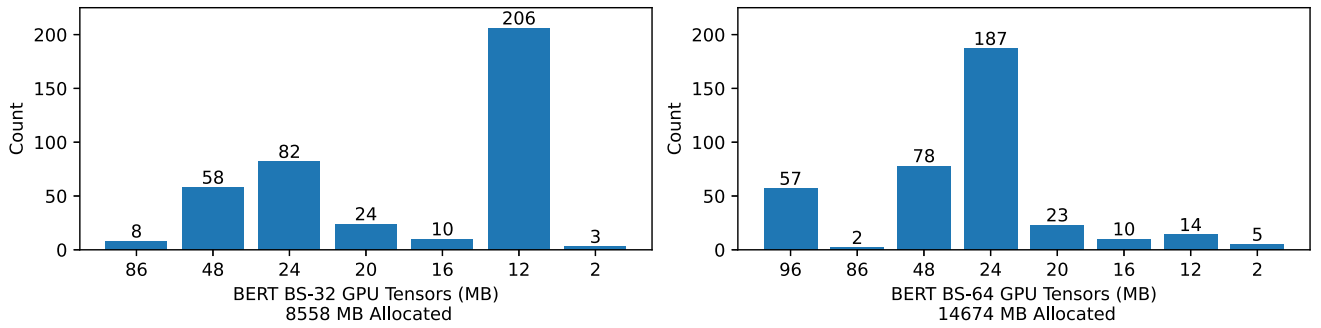


Fig. 11 PyTorch NLP Workload Allocated GPU Tensor Count by Size

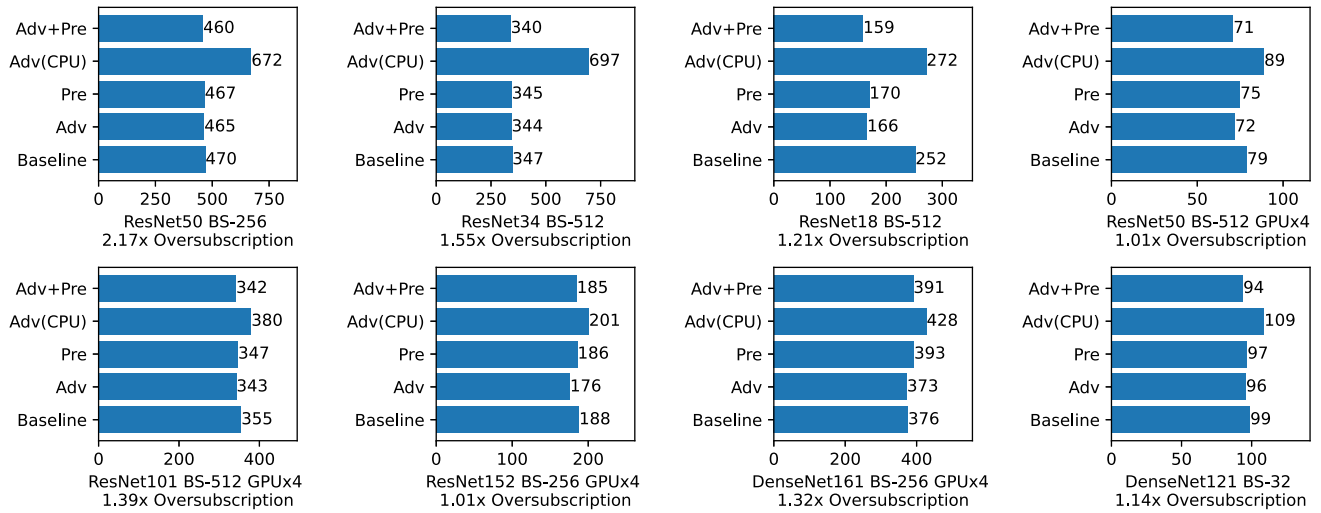


Fig. 12 PyTorch Vision Processing Workload Training Execution Time(s) for 50 Iterations on Titan XP GPUs

respectively. We noticed that for ResNet, doubling the batch size does not affect the number of different types of tensors; only the size of the individual tensors are mostly doubled, as shown in Figure 10. For DenseNet, the number of different types of tensors increases as the batch size changes. Total memory used by the GPU also increases as batch size is increased. DenseNet generally uses various smaller types of tensors, while ResNet uses larger size tensors. For BERT, doubling batch size does not really affect the number of different size tensors. However, there is a tensor size that is used much more heavily than other sizes, and that is size 12 MB for BERT BS-32 and size 24 MB for BERT BS-64. We also notice a trend that workloads using smaller tensor sizes tend to benefit more from our memory optimization techniques in our results that we will show next.

The results of training varying vision processing models on the Titan XP GPUs are shown in Figure 12. The labels on the x-axes explain the model name, batch size, and the amount of GPU tensor data oversubscribed in comparison to the total amount of GPU memory available. The labels on the y-axes show the memory techniques that we applied to the model in order to increase performance. Lower execution times mean better performance. We only show the **best case times** for the various memory optimization techniques we apply on tensors that we select based on heuristics and profiled information. We can perform the memory advise using several options, `AccessedBy`, `SetPreferredLocation`, and `SetReadMostly`, and we can also choose to advise the device we prefer the tensors to reside in (CPU or GPU). **Pre** refers to using prefetching with selected tensors based on counter information, and the location that we perform prefetching on is (determined through trial and error) before the tensors are accessed in the code. **Adv** refers to using memory advise on selected tensors that have high GPU page fault counts or large amounts of data transfers from CPU to GPU. The default option for memory advise that we perform if nothing is mentioned is `SetPreferredLocation`. We also try to prefetch tensors in advance before coherence faults occur in a way that would improve performance by placing the prefetch operations in sections in code that would provide such benefit. Again, where we place such prefetch operations is completely decided by trial and error. **Adv+Pre** refers to combining memory advise and prefetching. **Adv(CPU)** is an experimental case where we

place tensors on host memory to be directly accessed by GPU in order to avoid page faults and transfers to GPU memory. Oversubscription rate is calculated by the total memory of GPU tensors allocated divided by the total memory of all GPU devices in use.

The trend that we noticed was that the *lower* the oversubscription rate, the more performance gain we were able to achieve using memory techniques. In the case of ResNet18 with BS-256, we were able to reduce execution time from 252 seconds to 159 seconds, which is approximately a 38% reduction. In comparison to this, when oversubscription rate was over 2 for ResNet50 BS-256, execution time was only roughly the same, slightly better, or even worse no matter what technique we applied on various combinations of tensors. This isn't to say that there is no way to reliably improve performance, it is just that we could not determine the exact timing in code to prefetch the appropriate tensors to apply memory optimizations on based on the profiled data. Because of the incredibly large amount of permutations that we can perform on memory advise and prefetching, there is no way of knowing the optimal permutation of the application of memory techniques on the tensors. Therefore we rely on heuristics and the profiled data that we receive from the CUPTI profiler after the first few iterations of training to decide which tensors to apply memory techniques on. However, the profiled data for most deep learning workloads showed that in the case of oversubscription, all tensors in their entire size were evicted into CPU host memory (in chunks of data that we summed) and recalled back into GPU memory using page faults every single iteration in the training loop. Applying memory advise and prefetching on too many tensors at once also causes an adverse effect on performance. Therefore each individual tensor should be more actively monitored at the exact point in time in which they are accessed during training. Simply profiling the accesses across the entire training iteration and counting totals for GPU faults and transfers is not enough to determine the access pattern of every single tensor. Directly placing tensors in host memory to avoid GPU faults (**Adv(CPU)**) also usually results in adverse performance decreases for virtually every single workload that we tested in this paper.

For DenseNet161 using four GPUs, we noticed that performing different heuristics for prefetching and memory advise based on the profiled data actually decreased performance from baseline 376 seconds to 391 seconds using **Adv+Pre**. In this case, prefetching actually worsened performance more than advise. This could be due to improper timing of prefetches in the experiment. We did not distinguish between read faults and write faults in choosing the tensors to be advised in the GPU. Also oversubscription rate was quite high. In our results for DenseNet121 (tested on 1050 Ti instead of Titan XP), advising mostly write-only tensors improved execution time from 99 seconds to 96 seconds in 50 iterations. Prefetching did not make a significant difference in execution time, also possibly due to improper placement of prefetching operations. For multi-GPU ResNet152 and ResNet50, advising tensors to remain on their respective GPUs improved execution time performance from 188 seconds to 176 seconds and 79 to 72 seconds respectively. This could be attributed to low oversubscription rates of 1.01x.

Our solution is straightforward to use on single or multiple GPUs requiring minimal changes to application code when more devices are recruited. The reason for this was stated in Sect. 3.3, where tensors are already distributed and allocated in their respective GPU devices by the PyTorch framework. Therefore, we can simply store the **device ID** and use memory prefetching and advise on the corresponding device as a parameter to our API. Tensors located on certain GPU memory would not be prefetched and advised to remain on different GPU memory because the accessing device would be different and doing so would almost always result in performance slowdowns. If one GPU is oversubscribed, then other GPUs would also be oversubscribed because data tends to be allocated evenly in PyTorch. Therefore the performance results for multiple GPU tend to be similar to single GPUs.

We also test our solution on NLP models like BERT. Figure 13 shows the experimental execution times (in seconds) of NLP workloads. In the BERT-Base BS-32 MRPC workload, we set the `SetReadMostly` option for tensors where there are less than 2 GPU write page faults during profiling. However, the results show that this actually *worsens* performance by almost 70 seconds compared to the baseline of 283 seconds (not shown in graph because it is not a best case time) because the option creates duplicate data on both host memory and the GPU memory. The purpose of it is to reduce CPU page faults, but based on our profiled data there are no CPU page faults for any of the GPU tensors during training, nullifying the purpose. Forcing tensors to remain at host memory also reduces performance heavily, because this causes GPU accesses to access zero-copy memory, and forces data to travel across the PCIe bus every single time they are accessed. Because all tensors are usually accessed during every iteration similar to the vision processing workloads, this causes severe performance drawbacks. This resulted in poor execution times of 642 seconds for **Adv (CPU)**. It adviced the tensors to reside in host memory to be directly accessed from the GPU. The best results were obtained when we combined memory advising and prefetching (**Adv+Pre**) to obtain an execution time of 270 seconds compared to the baseline of 283.

For the other BERT workloads, we did not attempt needlessly place tensors in host memory because we knew

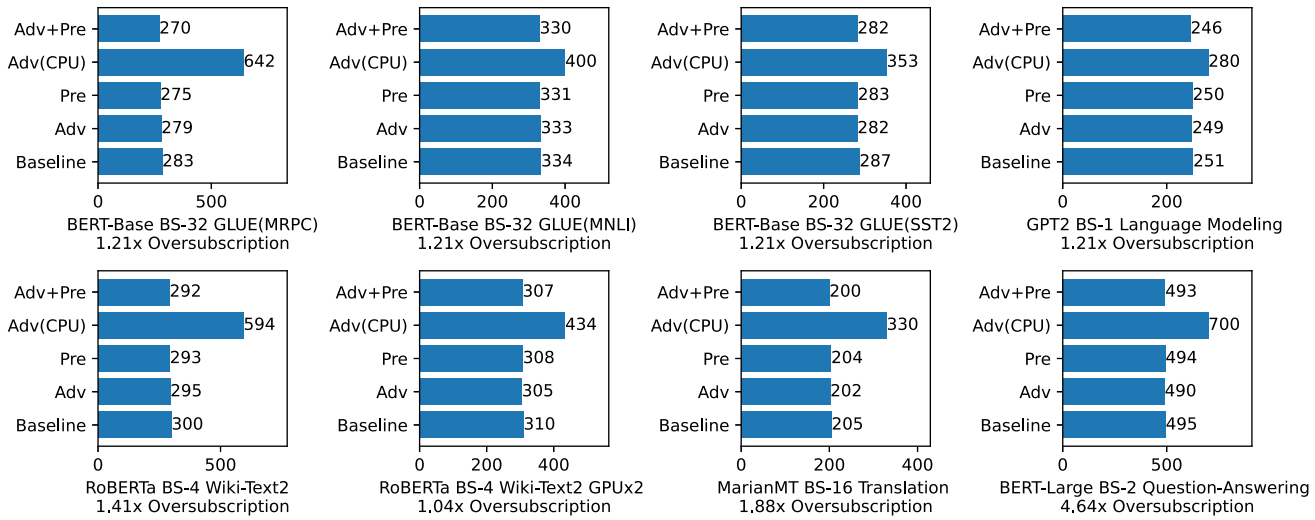


Fig. 13 PyTorch NLP Workload Training Execution Time(s) for 50 Iterations on 1050 Ti and 1660 Ti GPUs

already that it would result in poor execution times. These other BERT workloads use different task names like **MNLI** and **SST2** as a different method of training General Language Understanding Evaluation (GLUE) workload, where they differ in accuracy and speed. In both cases, prefetching and advising the top-3 tensors in terms of total page fault count gave slightly better results of 330 seconds compared to the baseline of 334 seconds, similar to the first example. All tasks had identical oversubscription rates. We tested fine training GPT2 and RoBERTa for language modeling on WikiText-2 [4] dataset, and achieved better results when advising and prefetching write-only tensors which have a majority of write GPU faults compared to read GPU faults. MarianMT was a translation workload and this approach was also effective towards it, even though it had a higher oversubscription rate. For BERT-Large, the oversubscription rate was 4.64x, so we tested it on 15 iterations instead of the usual 50 due to extremely long execution times on our machines, and were able to increase performance slightly by prefetching and advising write-only tensors as well. Overall, the results for both vision processing and language processing workloads showed similar straightforward patterns, with performance decreasing for **Adv(CPU)** scenarios and performance generally improving slightly in the best case when prefetch and memory advise is used on tensors.

5 Conclusion

In this paper, we created a fine-grained approach to selectively profile the activities for all GPU-allocated tensor in PyTorch using CUPTI, and created an API for the user to apply unified memory techniques like memory advise and prefetching on GPU-allocated tensors anywhere in the PyTorch application. We profiled activities like GPU/CPU read and write page faults, and DtoH and HtoD coherence, speculative, and eviction transfers. We dynamically profiled several vision processing and NLP workloads like ResNet, DenseNet, BERT, RoBERTa, GPT2, and MarianMT for a set amount of iterations, and then applied unified memory techniques on heuristically selected tensors. We were able to achieve minor performance improvements of up to roughly 8% on less oversubscribed workloads like multi-GPU ResNet152 and ResNet50. In addition to this, we were able to achieve performance improvements of up to 4% by prefetching and advising **write-only** tensors on the NLP workloads regardless of oversubscription rates. Workloads that used smaller tensor sizes like NLP and DenseNet also tended to benefit more from our unified memory techniques regardless of oversubscription rates.

Despite this, there were workloads where we were not able to achieve performance improvements. The reason was because based on our profiling results, all data ranges of GPU-allocated tensors in PyTorch are repeatedly used in every single training iteration making optimization difficult. In addition to this, prefetching needs to be timed optimally for tensors to be removed from GPU when they are not needed and retrieved beforehand in advance before they are used. This is difficult to perform in a fine-grained manner because of the large number of tensors that are allocated and used at different points of the training iteration. Also, simply advising large numbers of tensors to remain without being evicted in GPU is not effective because other tensors would inevitable have to be fetched from host memory. Instead of using heuristics to somewhat randomly guess which tensors are better suited to remain or

be prefetched in GPU memory, a better option would be to have pre-determined knowledge on the exact point in time each individual GPU tensor is required during each phase of training (through exact profiling of timestamps relative to the point in code). Then automatic application so memory advise and prefetch based on such knowledge would be helpful to solve this issue. This is what we want to pursue in future work.

Author Contributions Jake Choi: Conceptualization, Software, Implementation, Writing - original draft, Review, Methodology, Validation. Heon Young Yeom: Supervision. Yoonhee Kim: Corresponding Author, Funding acquisition, Administration, Supervision.

Funding This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.NRF-2021R1A2C1003379)

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

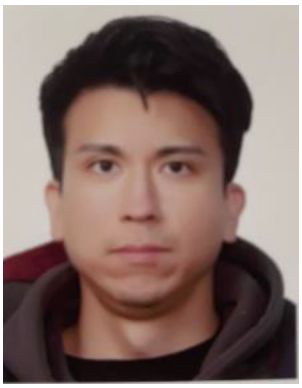
Competing interests The authors have no relevant financial or non-financial interests to disclose.

Informed Consent Written informed consent for publication of this paper was obtained from all authors.

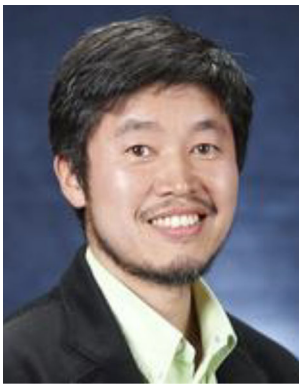
References

1. Ebubekir, B., Banu, D.: Performance Analysis and CPU vs GPU Comparison for Deep Learning. In: 2018 6th International Conference on Control Engineering & Information Technology (CEIT) (pp. 1–6). (2018). https://doi.org/10.1109/CEIT.2018.8751930
2. Huang, C., Jin, G., Li, J.: SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 1341–1355. (2020). https://doi.org/10.1145/3373376.3378530
3. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep Learning with Limited Numerical Precision. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). (2015). JMLR.org, 1737–1746
4. Judd, P., Albericio, J., Hetherington, T., Aamodt, T., Jerger, N., Moshovos, A.: Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In: Proceedings of the 2016 International Conference on Supercomputing (ICS'16). (2016). Association for Computing Machinery, Article 23
5. Chen, C., Choi, J., Brand, D., Agrawal, A., Zhang, W., Gopalakrishnan, K.: Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In: AAAI (2018)
6. Lin, Y., Han, S., Mao, H., Wang, Y., Dally, W.: Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. arXiv preprint arXiv:1712.01887 (2017)
7. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost (2016). arXiv preprint arXiv:1604.06174 (2016)
8. Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., Keckler, S.: vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'49). IEEE Press, Article 18 (2016)
9. Jain, A., Phanishayee, A., Mars, J., Tang, L., Pekhimenko, G.: Gist: Efficient data encoding for deep neural network training. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) (2018), IEEE, pp. 776–789
10. S. S.B., Garg, A., Kulkarni, P.: Dynamic Memory Management for GPU-Based Training of Deep Neural Networks. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 200–209 (2019). https://doi.org/10.1109/IPDPS.2019.00030
11. Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Leon Song, S., Xu, Z., Kraska, T.: Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18) (2018)
12. Abadi, M., Barham, P., Chen, J., Chen, Z., A, Davis, Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI, Vol. 16, pp. 265–283 (2016)
13. Collobert, R., Bengio, S., Mariéthoz, J.: Torch: a modular machine learning software library. Tech. rep, Idiap (2002)
14. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia, ACM, pp. 675–678 (2014)
15. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
16. Davis, L.: Handbook of genetic algorithms. (1991)
17. Awan, A., Chu, C., Subramoni, H., Lu, X., Panda, D.: OCDNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In: 25th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC) (2018)
18. Manian, K.V., Ammar, A.A., Ruhela, A., Chu, C.-H., Subramoni, H., Panda, D. K.: Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures. In: Proceedings of the 12th Workshop on General Purpose Processing Using GPUs (GPGPU '19). Association for Computing Machinery, New York, NY, USA, 43–52 (2019). https://doi.org/10.1145/3300053.3319419
19. Ren, J., Rajbhandari, S., R.Aminabadi, Y., Ruwase, O., Yang, S., Zhang, M., Li, D., He, Y.: ZeRO-Offload: Democratizing Billion-Scale Model Training. (2021). arXiv:abs/2101.06840
20. Knap, M., Czarnul, P.: Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. J. Supercomput. **75**, 7625–7645 (2019). https://doi.org/10.1007/s11227-019-02966-8
21. Sakharnykh, N.: Maximizing unified memory performance in cuda. (2017). https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
22. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unifed memory technology on nvidia gpus. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp 1092–1098. (2015). https://doi.org/10.1109/CCGrid.2015.105

23. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32 (pp. 8024–8035). (2019). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
24. Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints, abs/1605.02688, May (2016)
25. Awan, A.A., Chu, C., Subramoni, H., Lu, X., Panda, D.K.: OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), (2018), pp. 143–152, https://doi.org/10.1109/HiPC.2018.00024
26. Min, S., Wu, K., Huang, S., Hidayetoglu, M., Xiong, J., Ebrahimi, E., Chen, D., Hwu, W.: PyTorch-Direct: Enabling GPU Centric Data Access for Very Large Graph Neural Network Training with Irregular Accesses. CoRR abs/2101.07956 (2021)
27. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems; Curran Associates, Inc.: New York, NY, USA, (2012); pp. 1097–1105
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 26 June–1 July (2016); pp. 770–778
29. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv 2014, arXiv:1409.1556
30. Barnes, Z.: Techniques for Image Classification on Tiny-ImageNet. (2017)
31. Choi, H., Lee, J.: Efficient use of GPU memory for large-scale deep learning model training. Appl. Sci. **11**(21), 10377 (2021). https://doi.org/10.3390/app112110377
32. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., et al.: Transformers: state-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45, Online. Association for Computational Linguistics. (2020)
33. Merity, S., Xiong, C., Bradbury, J., Socher, R.: Pointer Sentinel Mixture Models. (2016)
34. Chen, C.L., Chen, C.C., Yu, W.H., et al.: An annotation-free whole-slide training approach to pathological classification of lung cancer types using deep learning. Nat. Commun. **12**, 1193 (2021). https://doi.org/10.1038/s41467-021-21467-y
35. Chuang, W.Y., Chen, C.C., Yu, W.H., et al.: Identification of nodal micrometastasis in colorectal cancer using deep learning on annotation-free whole-slide images. Mod. Pathol. (2021). https://doi.org/10.1038/s41379-021-00838-2
36. Choi, J., Yeom, H. Y., Kim, Y.: Implementing CUDA Unified Memory in the PyTorch Framework. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), (2021), pp. 20–25. https://doi.org/10.1109/ACSOS-C52956.2021.00029
37. Anaconda Software Distribution.: Anaconda Documentation. Anaconda Inc. Retrieved from https://docs.anaconda.com/ (2020)
38. Caffe2. https://caffe2.ai/
39. CUPTI. https://docs.nvidia.com/cuda/cupti/index.html
40. NVIDIA.: Beyond GPU Memory Limits with Unified Memory on Pascal, 2016. URL https://developer.nvidia.com/blog/beyond-gpumemory-limits-unified-memory-pascal/
41. NVIDIA, cuDNN: GPU Accelerated Deep Learning, 2016
42. NVIDIA Profiler nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html
43. NVIDIA Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/
44. PyTorch Documentation.: https://pytorch.org/docs/ stable/cpp_extension.html/ (2020)
45. CUDA-UVM-GPT2.: https://github.com/kooyunmo/cuda-uvm-gpt2/ (2020)



Jake Choi is a PhD candidate in the Distributed Computing Systems Laboratory at Seoul National University. He received his BS in Computer Science with a minor in Management Science from KAIST in 2013. His research interests include GPUs, systems, and computer security.



Heon Y. Yeom (Member, IEEE) received the B.S. degree in computer science from Seoul National University, in 1984, and the M.S. and Ph.D. degrees in computer science from Texas A &M University, in 1986 and 1992, respectively. From 1986 to 1990, he worked with the Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University, in 1993, where he currently is a Professor with the School of Computer Science and Engineering. He teaches and researches on distributed systems and transaction processing.

Yoonhee Kim She is the professor of the Computer Science Department at Sookmyung Women's University. She received her Bachelor's degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed and cloud computing. She is a member of IEEE and ACM, and she has served on a variety of program committees, advisory boards, and editorial boards.