2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C) | 978-1-6654-4393-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ACSOS-C52956.2021.00029

Implementing CUDA Unified Memory in the PyTorch Framework

Jake Choi
Department of Computer Engineering
Seoul National University
Seoul, Korea
kidcoder@snu.ac.kr

Heon Young Yeom
Department of Computer Engineering
Seoul National University
Seoul, Korea
yeom@snu.ac.kr

Yoonhee Kim
Department of Computer Science
Sookmyung Woman's University
Seoul, Korea
yulan@sookmyung.ac.kr

***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. In this paper, we propose a modification that utilizes CUDA Unified Memory (UM) to expand GPU memory to the available host memory space so that practicality for the programmer can increase, and OOM memory errors will not result for any workload. We also pinpoint performance issues that result from our modifications to the framework, and outline future plans like reducing redundant memory copies, prefetching, and memory advising techniques to improve upon our design. Our implementation shows that PyTorch UM performance overheads are minimal when the data footprint is below GPU memory capacity.**

***Index Terms*—CUDA, Unified memory, PyTorch, framework**

I. 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 [13], PyTorch [27]), 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) [18] 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 II provides related work, background knowledge about CUDA UM, PyTorch, and insights about our design. Section III discusses the experimental setup and the implementation of UM on PyTorch. Section IV evaluates the performance of PyTorch with UM on the machines and the current limitations of the implementation. Section V concludes this work with future insight.

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2021R1A2C1003379). (Corresponding Author: Yoonhee Kim).

II. Related Work and Background

A. 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 [12] 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 scheduling runtime for training deep non-linear

978-1-6654-4393-7/21/$31.00 ©2021 IEEE
DOI 10.1109/ACSOS-C52956.2021.00029

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 [14], TensorFlow, Caffe [15], or MXNet [16].

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 [17] 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 [20] 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.

B. 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 SM, which stalls any future translations until all page faults are resolved [23]. 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 [22]. 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% [24].

C. 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 [28] 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 of the 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, the PyTorch developers think that UM will have too much of a performance overhead. Therefore other explicit methods of memory management that we already outlined in Section I are used instead. For graph neural network (GNN) use cases, a unified tensor implementations for PyTorch exist [30]. 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.

D. Design Differences with Other Frameworks

OC-DNN [29] 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 [31] 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.

III. Implementation and Experimental Setup

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. Figure 1 shows a simplified view of the PyTorch

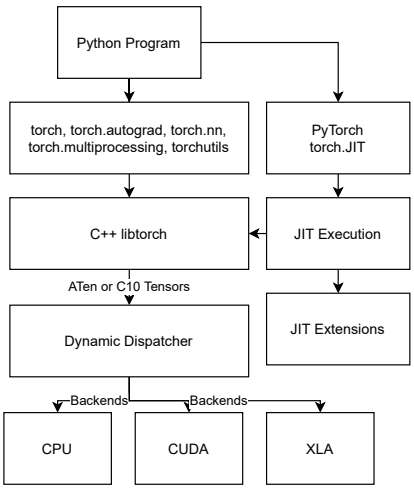

Fig. 1. Brief PyTorch Architecture

architecture. We specifically modified the CUDA caching allocator in the CUDA backend of the c10 library. The caching allocator is used to speed up 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

When profiling the results from Listing 1 using *nvprof* [25] 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 2 shows a simplified diagram of what NVIDIA Visual Profiler [26] outputs when Listing 1 is executed. In the actual profiled results, 18 additional PyTorch CUDA kernels are also executed, but have very low significance that they are omitted from Figure 2. 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.

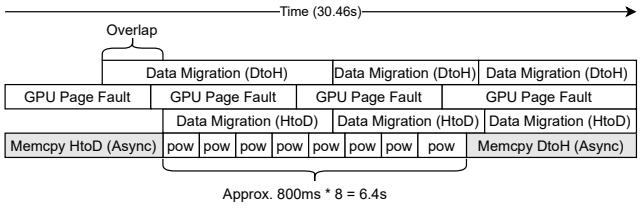

Fig. 2. Profiling UM page faults with *nvvp* (16 GB data oversubscribed)

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 3), 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 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.

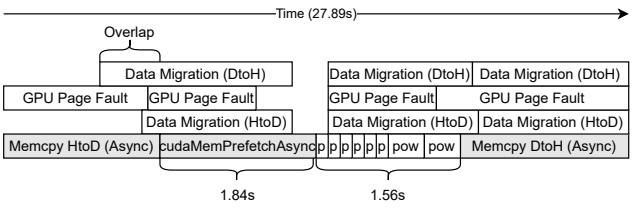

Fig. 3. Profiling UM after prefetch optimization (16 GB data oversubscribed)

The experimental setup is a single server machine equipped with 4 NVIDIA Titan XP GPUs. Each GPU has 12 GB of memory available. The server has a total of 125 GB of host memory available. We use pytorch 1.9.0 in developer mode for implementation purposes, and also use Anaconda [36] 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 [32], Resnet [33], and VGG [34]. The dataset used is the Tiny Imagenet 200 [35].

IV. Evaluation

We executed the Resnet-18 workloads with batch size 256 and 512 with and without UM and profiled the results using *nvprof*. Training using built-in deep learning models along with the Tiny Imagenet dataset in PyTorch invokes a vast number of cuDNN, GEMM, and backend CUDA operator kernels. We limited the number of iterations in the PyTorch Imagenet dataset to 5 in one epoch and ended the training afterwards because the training pattern does not change. Figure 4 shows the execution time for the workloads. We notice that the execution time for PyTorch with UM is similar to the base

execution time for all batch sizes which fit in GPU memory. Once GPU oversubscription occurs, the base case results in an OOM because the program fails to `cudaMalloc`. The bar is missing from the graph for the cases where this occur. One exception is that UM-Resnet-18 BS 512 takes much longer than Resnet-18 BS 512 because the GPU is oversubscribed. The UM implementation causes slightly more memory to be allocated on the GPU resulting in an unintended oversubscription of the GPU. In addition to these factors, using UM also increases CPU host memory allocation by the equivalent amount of data transferred to the GPU because of redundant copies which we mentioned in Section III.

Table I shows the top 5 kernels that take up the most time during execution of the profiled Resnet-18 workloads for 5 training iterations. When identical workloads are run, the total number of kernel invocations does not change. Application of UM changes the number and order of kernel invocations which take up the most time. Also, when memory is oversubscribed, the total amount of kernels increases significantly compared to the non-UM case with the same batch size. Even when limited to only 5 total iterations (an actual training phase would contain thousands of iterations with many epochs), total program execution time increases significantly and kernel overhead is also greatly increased. The last row shows the oversubscribed case, in which total execution time almost doubles from 27.1s to 50.68s and compute kernel time more than triples from 11.58s to 36s. This is because the kernels are not aware of the location of the intermediate data and are stalled. Even though it is not shown in the table, total compute utilization (percentage of time spent in kernels divided by total session time) increases from 42.4% to 67.2% from the base case to the oversubscribed case.

TABLE I
PROFILING KERNEL INVOCATIONS BY COMPUTE TIME FOR EACH MODEL

Top 5 Kernel Invocations Per Model			
	Kernel Name	***Calls (Time)***	***%***
Batch Size 256 Resnet-18 (20.76s)	maxwell_scudnn_wi...	207 (871ms)	12.6%
	vector_elementwise_ker...	884 (592ms)	8.6%
	cudnn::bn_fw_inf...	800 (486ms)	7.1%
	maxwell_gcgemm_32...	360 (410ms)	6.0%
	maxwell_scudnn_128...	46 (337ms)	4.9%
	Total	10449 (6.89s)	100%
Batch Size 256 Resnet-18-UM (24.79s)	maxwell_scudnn_wi...	207 (913ms)	9.6%
	cudnn::cnn::im2col4d...	22 (750ms)	7.9%
	maxwell_gcgemm_32...	354 (669ms)	7%
	vector_elementwise_ker...	782 (493ms)	5.2%
	maxwell_scudnn_128...	800 (485ms)	5.1%
	Total	10339 (9.5s)	100%
Batch Size 512 Resnet-18 (27.1s)	maxwell_gcgemm_32...	667 (806ms)	7.0%
	vector_elementwise_ker...	544 (754ms)	6.5%
	maxwell_scudnn_wi...	76 (691ms)	6%
	maxwell_scudnn_wi...	91 (600ms)	5.2%
	maxwell_scudnn_128...	26 (576ms)	5%
	Total	9057 (11.58s)	100%
Batch Size 512 Resnet-18-UM (50.68s)	vector_elementwise_ker...	102 (5.6s)	15.8%
	cudnn::bn_bw_1C11...	120 (2.79s)	7.8%
	at::native::_GLOBAL...	26 (2.62s)	7.3%
	maxwell_scudnn_wi...	126 (2.57s)	7.2%
	maxwell_cgemm_32...	10973 (2.55s)	7.1%
	Total	18435 (36s)	100%

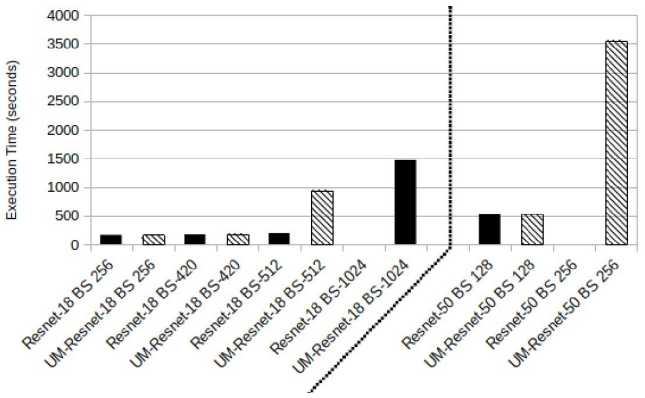

Fig. 4. Comparing execution time results for Resnet-18 and Resnet-50 on PyTorch with and without UM for different batch sizes

In Figure 5, we notice that there is no evidence of data migration between host and device, unlike Figure 2 and 3. Individual kernel execution times for certain kernels have increased significantly, such as `vector_elementwise_kernel` (shown in the figure as _ZN2at6n) taking a whopping 5.6s of time in only 5 iterations. Large amounts of intermediate data are created from the tensors already located in device memory which causes data migration activity to not show up the profiler. This is similar to the situation in Figure 2. Each iteration further exacerbates execution time because the same pattern is repeated. In addition to this, further analysis shows that data loaded through batches is also pinned in host memory using `cudaHostAlloc` and is read directly from the GPU without migration. In future work, redundant explicit data transfers that UM does not require need to be eliminated.

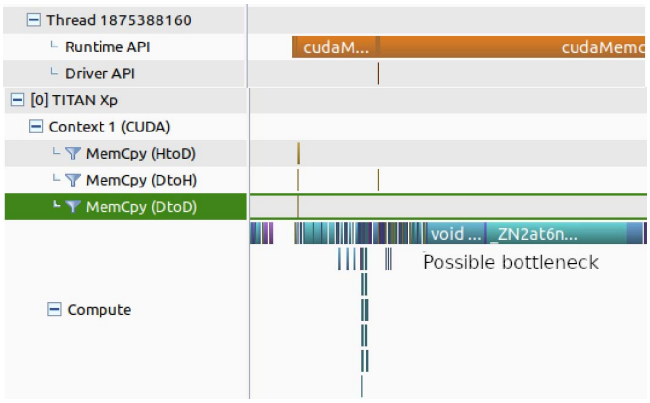

Fig. 5. Visual Profiling Resnet-18-UM with 512 batch size

V. CONCLUSION AND FUTURE WORK

In this paper, we have implemented UM in the PyTorch framework in order to remove the OOM memory problem at the bare minimum. We have noticed that under workloads that fit inside GPU memory, performance has not changed much.

However with workloads that are oversubscribed, performance has dropped. We propose in future work to remove all redundant memory copies and operations and implement UM allocation in the CPU side in order to make the framework more aware of UM. We have shown evidence that performance overheads during oversubscription can be alleviated by using memory optimization techniques like `cudaMemAdvise` and `cudaMemPrefetchAsync`. We propose to improve upon this to apply more general memory optimization techniques to work at appropriate situations where we can predict the data access patterns of kernels. We also want to effectively utilize UM across multiple GPUs when parallel processing is performed under the framework.

REFERENCES

[1] B. Ebubekir and D. Banu, 2018. Performance Analysis and CPU vs GPU Comparison for Deep Learning. 1-6. 10.1109/CEIT.2018.8751930.

[2] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. 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. DOI:https://doi.org/10.1145/3373376.3378530

[3] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 1737–1746.

[4] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In Proceedings of the 2016 International Conference on Supercomputing (ICS'16). Association for Computing Machinery, Article 23.

[5] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal,Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive Residual Gradient Compression for Data-parallel Distributed Training. In AAAI 2018.

[6] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. arXiv preprint arXiv:1712.01887, 2017.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv preprint arXiv:1604.06174 (2016).

[8] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. 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.

[9] Jain, A., Phanishayee, A., Mars, J., Tang, L., and 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., A. Garg and P. Kulkarni, "Dynamic Memory Management for GPU-Based Training of Deep Neural Networks," 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 200-209, doi: 10.1109/IPDPS.2019.00030.

[11] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. 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).

[12] NVIDIA, "cuDNN: GPU Accelerated Deep Learning," 2016.

[13] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In OSDI (2016), vol. 16, pp. 265–283.

[14] Collobert, R., Bengio, S., and Mariéthoz, J. Torch: a modular machine learning software library. Tech. rep., Idiap, 2002.

[15] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia (2014), ACM, pp. 675–678.

[16] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015).

[17] Lawrence Davis. 1991. Handbook of genetic algorithms. (1991).

[18] NVIDIA. Beyond GPU Memory Limits with Unified Memory on Pascal, 2016. URL https://developer.nvidia.com/blog/beyond-gpumemory-limits-unified-memory-pascal/.

[19] A. Awan, C. Chu, H. Subramoni, X. Lu, and D. Panda. 2018. 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).

[20] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda. 2019. 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. DOI:https://doi.org/10.1145/3300053.3319419

[21] Ren, J., Rajbhandari, S., Aminabadi, R.Y., Ruwase, O., Yang, S., Zhang, M., Li, D., & He, Y. (2021). ZeRO-Offload: Democratizing Billion-Scale Model Training. ArXiv, abs/2101.06840.

[22] 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

[23] Sakharnykh N (2017) Maximizing unified memory performance in cuda. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

[24] Li W, Jin G, Cui X, See S (2015) 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. https://doi.org/10.1109/CCGrid.2015.105

[25] "NVIDIA Profiler nvprof", https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[26] "Nvidia Profiler User's Guide." https://docs.nvidia.com/cuda/profiler-users-guide/.

[27] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[28] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints, abs/1605.02688, May 2016.

[29] A. A. Awan, C. Chu, H. Subramoni, X. Lu and D. K. Panda, "OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training," 2018 IEEE 25th International Conference on High Performance Computing (HiPC), 2018, pp. 143-152, doi: 10.1109/HiPC.2018.00024.

[30] Seungwon Min, Kun Wu, Sitao Huang, Mert Hidayetoglu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, Wen-Mei W. Hwu: PyTorch-Direct: Enabling GPU Centric Data Access for Very Large Graph Neural Network Training with Irregular Accesses. CoRR abs/2101.07956 (2021)

[31] "Caffe2," https://caffe2.ai/.

[32] 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.

[33] 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.

[34] Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. arXiv 2014, arXiv:1409.1556.

[35] Barnes, Z. (2017). Techniques for Image Classification on Tiny-ImageNet.

[36] Anaconda Software Distribution. (2020). Anaconda Documentation. Anaconda Inc. Retrieved from https://docs.anaconda.com/