Revista Colombiana de Computación
Vol. 25, No. 1. January – June 2024, pp. 29-38
e-ISSN: 2539-2115, https://doi.org/10.29375/25392115.5055
Selected paper previously presented at the Latin America High Performance Computing Conference (CARLA 2023), an event held in Cartagena de Indias, Colombia, September 18-22, 2023.

REVISTA COLOMBIANA DE COMPUTACIÓN

Containerization on a Self-supervised active foveated approach to Computer Vision

Dario Dematties[1], Silvio Rizzi[2], George K. Thiruvathukal[3]

[1] Northwestern Argonne Institute of Science and Engineering, Evanston, IL, United States
[2] Argonne National Laboratory, Lemont, IL, United States
[3] Loyola University Chicago, Chicago, IL, United States
ddematties@anl.gov, srizzi@alcf.anl.gov, gkt@cs.luc.edu

(Received: 6 February 2024; accepted: 18 June 2024, Published online: 30 June 2024)

Abstract. Scaling complexity and appropriate data sets availability for training current Computer Vision (CV) applications poses major challenges. We tackle these challenges finding inspiration in biology and introducing a Self-supervised (SS) active foveated approach for CV. In this paper we present our solution to achieve portability and reproducibility by means of containerization utilizing Singularity. We also show the parallelization scheme used to run our models on ThetaGPU–an Argonne Leadership Computing Facility (ALCF) machine of 24 NVIDIA DGX A100 nodes. We describe how to use mpi4py to provide DistributedDataParallel (DDP) with all the needed information about world size as well as global and local ranks. We also show our dual pipe implementation of a foveator using NVIDIA Data Loading Library (DALI). Finally we conduct a series of strong scaling tests on up to 16 ThetaGPU nodes (128 GPUs), and show some variability trends in parallel scaling efficiency.

Keywords: Singularity Containerization, NVIDIA DALI, Data Loading and Pre-processing Library, High Performance Computing, Strong Scaling.

1 Introduction

Today's CV main implementation challenges has two sources. First, inference requires substantial processing power. Second, training procedures demand extremely large high-quality labeled data sets. On the one hand, near real-time analytics surpasses the latency limitations found on centralized cloud computing by deploying models to the edge (Preuss, 2018). Nevertheless, CV models are big, therefore difficult to fit in edge devices. On the other hand, as is the case in medical imaging, data sets collection and labelling could require extremely skillful staff resulting in data scarcity to train the models (Castro et al., 2020). In order to overcome such difficulties we propose a strategy which gains inspiration from biological and developmental aspects found in humans an other mammals.

From a biological point of view, the retinas of some mammals do not sample the scene uniformly (Provis et al., 1998). Instead, they sample the visual field with a very high resolution in a tiny portion called the fovea and with very low resolution in the periphery (Purves et al., 2004). Thanks to foveated vision it is not necessary for the brain to process all the scene at high resolution, reducing computational (metabolic) cost. In order to compensate possible information losses, foveated animals need to scan images utilizing successive saccades and fixations.

From a developmental viewpoint, certain changes affect the characteristics of visual attention throughout early childhood in humans (Ross-Sheehy et al., 2015). Basically, important changes are produced in the saccadic behavior from childhood to adulthood, which range from reflexive saccades to the acquisition of the ability to control volitional saccadic eye movements (Johnson, 1995; Canfield & Haith, Young infants' visual expectations for symmetric and asymmetric stimulus sequences, 1991; Canfield & Kirkham, Infant Cortical Development and the Prospective Control of Saccadic Eye Movements, 2001). Furthermore, adults volitional saccades show some characteristics that differ from those of infants showing even further development and control towards adulthood (Weber & Daroff, 1972; Salapatek et al., 1980).

In Fig. 1 we show the strategy followed in this work. First of all we developed a foveated system and utilized it as a natural augmentation approach for self-supervised learning (Fig. 1 A). We advanced a strategy utilized in SimCLR (Chen et al., 2020), where a new approach to contrastive self-supervised learning algorithm is proposed. As shown in Fig. 1 A, we train a network called Backbone to classify different foveated fixations as the same when they come from the same image. When fixations come from

©2024 Universidad Autónoma de Bucaramanga - UNAB. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY-NC-SA 4.0) license (https://creativecommons.org/licenses/by-nc-sa/4.0/).

different images, the agreement in the output from the backbone has to be minimum. This strategy is called Contrastive Learning (CL). Originally, CL uses different augmentations such as crop-resize, Gaussian blur, Gaussian Noise, Color distortions, flipping, rotations, cutouts, etc. Our novel contribution here is that we use foveation as a bio-inspired augmentation mechanism in CL. Foveation and random fixations, acting in tandem with a proprioceptive system could result in a productive augmentation strategy in biological agents to understand the visual world around them. At the same time this scheme drastically mitigates the demands on high quality labelled data sets. As some animals do in their early lives, the backbone in Fig. 1 A learns visual features without labels, randomly wandering around scenes with erratic saccades and fixations.

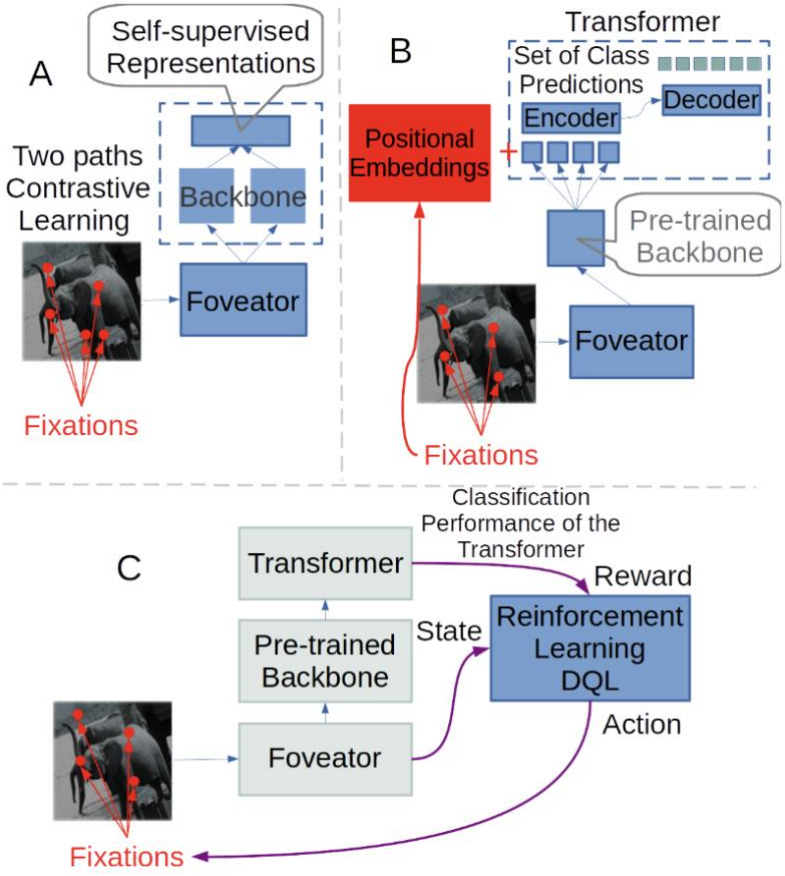

Figure 1. Active foveated developmental strategic scheme to solve CV problems such as the labeled data sets scarcity and the high computational complexity demanded by the models implementation. (A) Self-supervised contrastive learning approach using foveated vision as a biologically-inspired augmentation strategy. With this strategy we aim to mitigate labeled data sets scarcity. (B) A transformer architecture processing a sequence of outputs from a pre-trained backbone which process foveated fixations. Positional embeddings are determined by individual fixations and not by image patches, which saves great computing power demands from the transformer architecture perspective. (C) A RL architecture–a DQN–is added to the architecture to learn the saccadic behavior which is supposed to generate more effective fixation coordinates in order to increment the classification performance from the transformer.

Following this developmental trend found in biological agents, in Fig. 1 B we tested which additional features a sequence of fixations could bring to performance in CV tasks. We incorporated a transformer architecture utilizing our pre-trained backbone as a pre-processing stage of each successive fixation. To that end we adapted the original architecture developed by Facebook AI Research Group called DEtection TRansformer (DETR) (Carion et al., 2020). We eliminated several losses concerning detection, keeping only losses concerned with classification. In DETR, the transformer is used to encode an image pixel by pixel from the backbone output. We also adapted the positional encoder of the network and instead of encoding the position of every component from the backbone output we encoded the position of each fixation from our pretrained backbone.

Self-attention mechanisms have quadratic complexity (Vaswani et al., 2017). In a patch by patch scenario–as is the case in image Transformers (Dosovitskiy et al., 2021)–this situation represents a great

obstacle in the implementation of these kinds of architectures when trying to process big or 3D high resolution images. We address this by changing the strategy of giving each patch a position. We instead give each fixation a position in the network. The number of fixations will be considerable smaller than the number of patches in an image. For instance, for humans only two fixations suffice to recognize faces (Hsiao & Cottrell, 2008). This is a huge advantage in computational load terms.

The idea behind the incorporation of a transformer in the scheme is to give the system the capacity of attributing to a set of fixations a sequential nature. The transformer would extract any possible statistical structure if it exists inside a sequence. This is a step forward in our bio-inspired developmental approach since, in this stage, fixations evolve from being mere isolated entities to be part of a more complex structure–such as a sequence of fixations.

Nevertheless, fixation locations in our models are random so far. Yet the behavioral patterns found in saccades of biological systems are far from random (Ross-Sheehy et al., 2020; Castro et al., 2015; Spotorno et al., 2014; Alahyane et al., 2016). Optimization mechanisms from oculomotor behavior emergence in biological systems seems to be behind the reward dopamine system (Kato et al., 1995; Hikosaka et al., 2006; Ikeda & Hikosaka, 2003). Hence, RL in saccadic behavior is supported by biological evidence. As shown in Fig. 1 C, we incorporated a RL mechanism in our model with the aim of learning an effective saccadic behavior. Basically we trained a DQN, which treated the dynamic of the Transformer classifier as the environment. The observation of the state of the environment was the output from our foveated system, the actions taken by our network were the coordinates of the next fixation which gave rise to the next state from our foveator. Finally the classification performance from DETR was taken as the reward in this RL scenario.

2 Parallelization scheme

2.1 Pre-processing (foveation)

This stage in the Machine Learning (ML) workflow involves data loading and pre-processing. We incorporate fovetion here given its appropriateness for the pre-processing (augmentation) stages in the workflow. We use the NVIDIA Data Loading Library (DALI)1. DALI is a library for data loading and pre-processing to accelerate Deep Learning (DL) applications. DALI provides a collection of highly optimized building blocks for loading and processing image, video and audio data.

DL applications require complex, multi-stage preprocessing pipelines, (i.e., loading, decoding, cropping, resizing, and many other augmentations). Such data preprocessing stages are usually executed on Central Processing Units (CPUs). Hence they entail a bottleneck, limiting the performance and scalability of ML workflows in general.

Basically, DALI approaches the problem of the CPU bottleneck by offloading data pre-processing stages into Graphical Processing Units (GPUs). Additionally, DALI maximizes the throughput of the input pipeline autonomously, pre-fetching information, conducting parallel execution and by means of batch processing strategies which are all handled transparently for the user.

As Fig. 2 B shows, we developed our foveator as a dual DALI pipeline structure. One pipeline for reading an decoding a batch of images and the other for foveating and augmenting such batch. This is not the general strategy recommended in DALI in which the idea is to deploy a unique pipeline which gathers, decodes and applies all the necessary augmentations to the batch. In this way DALI applies all the optimization strategies such as pre-fetching information, parallel execution of several batches, etc. In our case, we use Pipe 1 to bring a batch and then Pipe 2 is the one which produces several foveated fixations and augmentations to the batch of images brought by Pipe 1.

Our main idea is to bring only one batch, and then apply several fixations to such a batch. Pipe 1 and Pipe 2 do not run in parallel. Such a feature will be implemented in the future.

With Pipe 1 we load and decode the images from the data set. As can be seen in Fig. 2 A, DALI split the data set in s shards–one shard per rank–each shard has m batches which are loaded and pre-processed sequentially by DALI.

For each batch loaded and decoded by Pipe 1, Pipe 2 produces a series of f foveations for each image in each batch. Each foveation has its own angle and fixation position in the image. Nevertheless, beyond the

foveation, Pipe 2 conducts a series of random augmentation on each image in the batch. For instance, before the foveation Pipe 2 produces crop and resize, flipping, color distortion, Gaussian noise and grid mask.

For each epoch, the shards are composed by a different set of batches of images as seen in the different colors chosen for different epochs in Fig. 2 A. This strategy is used to achieve certain variance in the input of each model from the data set, i.e., each model replica in the each rank will use all the images in the data set for training. At each epoch, each model replica will receive a different shard, in a rotation of the shards epoch by epoch.

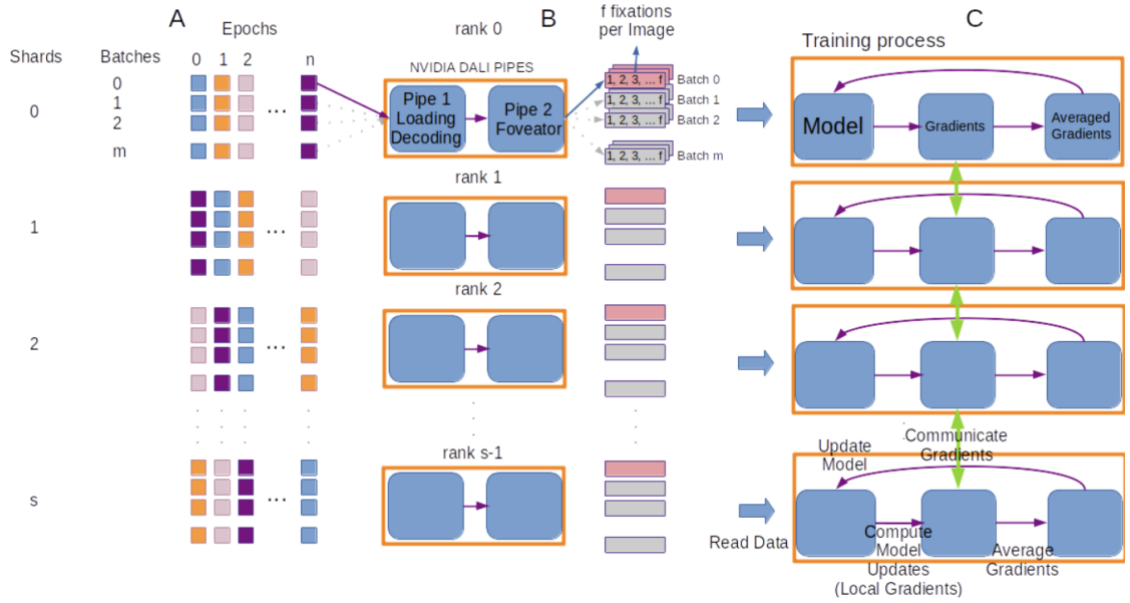

Figure 2. This is the parallelism scheme generally used in pytorch. The idea is to produce a model replica for each rank and to split the data set in different shards, each of which is processed by each model replica independently. Each square in A represents a batch of images. From one epoch to the following, the set of batches belonging to a shard rotates in a way that each model replica will process all batches in the data set. B shows how our foveator is build using two DALI pipes. Pipe 1 loads and decodes a complete batch of images while Pipe 2 produces a set of f fixations from such a batch. C shows how training proceeds. First, each model replica produces a forward pass of its respective batch and compute its loss and gradients, second, the gradients are communicated among all ranks and averaged in each rank. Consequently, each model replica will have the same gradients even when they processed different mini-batches. Using the averaged gradients, each model replica updates its weights and the process repeats.

2.1 DistributedDataParallel (DDP)

Once the model is build and occasionally brought from a checkpoint it has to be wrapped by DDP. By simply replacing the model with DDP(model), DDP will synchronize the weights and, during training, all reduce gradients before applying updates. DDP handles it transparently for the user. In our repository we use exactly the following code:

```
model = DDP(model, device_ids=[args.gpu], output_device=args.gpu)
```

This code sentence applies data parallelization. As shown in Fig. 2 C, with this scheme, all the ranks have their own replica of the model. As already shown in Fig. 2 A, the global batch of data is split into multiple mini-batches, and processed by different ranks. Each rank computes the corresponding loss and gradients with respect to the data it processes. Before updating the parameters at each epoch, the loss and gradients are communicated and then averaged among all the ranks through a collective operation. This scheme is relatively simple to implement. MPI_Allreduce is the communication operation used to communicate all gradients to all ranks in the run.

3 Implementation Technical Challenges

3.1 Containerization

Containerization is the procedure of building a light-weight executable file–called container–which contains only the operating system as well as some libraries and dependencies to run code consistently on any infrastructure. Singularity is a containerization platform that provides the capability of packaging software in a portable and reproducible way. The general procedure is to build a container on a local machine–on a computer where the researcher or developer has administrator (i.e. super user or root) privileges–and then to use its portability to run it on any largest High Performance Computing (HPC) system. One of the most important features of Singularity containers is that they run fluently using GPUs, high speed networks and parallel file systems on clusters. In order to achieve this portability, containers systems, such as Singularity, have a user space which is platform independent, containing all the necessary programs, custom configurations and environment needed to run in any kernel space.

For our case we first converted an NVIDIA NGC Docker Image to Singularity:

```
sudo singularity pull
pytorch_21.04-py3.sif
docker://nvcr.io/nvidia/pytorch:21.04-py3
```

Then we adapted our Singularity image to our needs. Basically we created a writeable sandboxed directory from our pytorch_21.04-py3.sif image, entered, modified, and finally created an updated version of our image. For instance adding a package–here mpi4py–can be done with:

```
sudo singularity build --sandbox
pytorch_21.04-py3 pytorch_21.04-py3.sif
sudo singularity shell --writable
pytorch_21.04-py3/
Singularity pytorch_21.04-py3/:~> pip
install mpi4py
Singularity pytorch_21.04-py3/:~> exit
sudo singularity build
pytorch_21.04-py3_v2.sif pytorch_21.04-py3
```

At this point, the container called pytorch_21.04-py3_v2.sif has all the packages we need to run our workflows.

Any installations can be done into the image, including with apt-get (in the case of Debian GNU/Linux), pip (as we proceeded) or using the container image OS native build tools (such as gcc etc.) to manually build and install whatever software is needed.

3.1 Submitting jobs on Theta GPUs

Theta GPU nodes are NVIDIA DGX A100. The DGX A100 comprises eight NVIDIA A100 GPUs that provide a total of 320 gigabytes of memory for training Artificial Intelligence (AI) datasets, as well as high-speed NVIDIA Mellanox ConnectX-6 network interfaces.

The system used for submitting jobs at ALCF is Cobalt. Cobalt is a scheduler and resource manager for HPC systems. Cobalt jobs may run either as script jobs or interactive mode jobs. We run our jobs using three scripts. The first script uses the Cobalt environment variables to launch the jobs. Typical Cobalt variables are the number of nodes, the time allocated, the project name, etc.

It is important to know that data access between centre storage such as /home or /projects and the Theta GPU compute nodes is not suitable for I/O intensive loads of small random-read/write character common in for instance ML. Accessing data sets from network file systems is slow and undetermined. Consequently, the Theta GPU nodes are equipped with local scratch solid state drives, which are faster compared to other network file system Theta nodes. These disks are suitable for this type of I/O load and are available to each

job under /local/scratch like all scratch disks on Theta GPU. Just as on the rest of Theta GPU, this scratch space is volatile, accessible on a per-job basis, and will be cleared after each Cobalt job.

Transferring data sets to the node local scratch space should be done in large contiguous chunks to maximize transfer speed. A good strategy to achieve this is to store your data set on centre storage (i.e., /projects) in an uncompressed tar archive, transfer this to scratch and finally unpacking there in the Cobalt job on the allocated GPU node.

Therefore, the first script launches a second script which copies all the data set–imagenet ILSVRC 2012–on a local node scratch during run. The task for this second script is to copy all the data set into such a space. Initially the script copies all the tar balls in the scratch space and finally untars all the images there.

First, copying the train tarballs to scratch is something like:

```
cp -rv /lus/theta-fs0/software
/datascience/datasets/tar_balls
/train/*.tar ./train
```

Then untaring the train tarballs in scratch is like:

```
cd train
for f in `ls *.tar`; do echo $f; mkdir
${f%.tar}; tar -xf $f -C ${f%.tar}; done
cd ..
```

The first script launches one second script per node to copy all the data sets on the respective scratch file drives in each node. In order to do so, the first script uses mpirun command in the following way:

```
mpirun -n $N_NODES -hostfile
${COBALT_NODEFILE} -map-by node
$SSD_SCRIPT
```

Where N_NODES is the number of nodes and SSD_SCRIPT is the second script, the one that copies the data set on the scratch space. This command launches one process (i.e., rank) per node, and each rank runs the second script which copies and untars the data set on the scratch space.

Finally the first script launches the third script containerized, like:

```
mpirun -n $N_RANKS -hostfile
${COBALT_NODEFILE} -map-by
node singularity run --nv -B
/lus:/lus,/raid:/raid $CONTAINER $SCRIPT
```

where `N_RANKS` is the number of ranks, which in this case is

```
N_RANKS=${RANKS_PER_NODE}*${N_NODES}.
```

Cobalt script (the first script) which runs on one node, and uses mpirun, launches n singularity instances, in our case 8 per node as can be seen in Fig. 3. Each singularity instance launches the same shell script (the third script). Each shell script sets up a virtual environment, and then executes the main python script. The variable CONTAINER holds the container path and SCRIPT holds the path to the third script.

3.2 Model parallelization

DDP from Pytorch and mpi4py were used to manage data set parallel processing. Basically we distributed n Message Passing Interface (MPI) processes (ranks) in n GPUs as shown in Fig. 3. For collective communication, we set DDP to use NCCL on GPUs–gloo can be used on CPUs. DDP replicates the whole model in each rank. Each model replica processes a different part of the data set. Gradients computed during the backward pass in each model replica are communicated, averaged and used to conduct weight adjustments in each model.

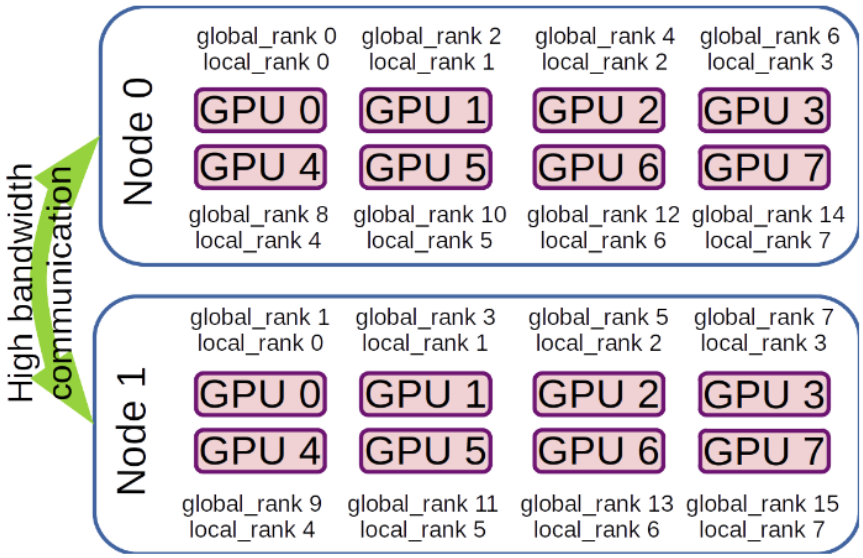

Figure 3. Distribution of global and local ranks on Theta GPU devices for 2 nodes. The organization is the same for more than 2 nodes i.e., ranks are distributed alternatively per compute node. There are 16 ranks, which are distributed per node, i.e., global rank 0 in node 0, global rank 1 in node 1, global rank 2 in node 0, global rank 3 in node 1, and so on. Local ranks numbers map device numbers matching their ids inside a node, i.e. local rank 0 in GPU 0, local rank 1 in GPU 1, and so forth.

We start DDP by calling pytorch's init_process_group function from the DistributedDataParallel (DDP) package. This function has some options, but the easiest is typically to use the environment variables. It is also needed to assign each script a rank and world size. Luckily, DDP is compatible with MPI, therefore all the information for each process identification can be obtained from mpi4py.

First, we used openmpi environment variables to read the local rank:

```
local_rank =
os.environ['OMPI_COMM_WORLD_LOCAL_RANK']
# Use MPI to get the world size
# and the global rank:
size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
```

Then set the Pytorch environment:

```
os.environ["RANK"] = str(rank)
os.environ["WORLD_SIZE"] = str(size)
os.environ['CUDA_VISIBLE_DEVICES'] =
str(local_rank)
```

Then get the hostname of the master node, and broadcast it to all other nodes.

```
if rank == 0:
master_addr = socket.gethostname()
else:
master_addr = None
master_addr =
MPI.COMM_WORLD.bcast(master_addr,
root=0)
```

Finally set the master address on all node's environments:

```
os.environ["MASTER_ADDR"] = master_addr
Port can be any open port.
os.environ["MASTER_PORT"] = str(2345)
```

After this, we use the `init_method='env:// argument`

```
in init_process, something like:
torch.distributed.init_process_group
(backend='nccl', init_method='env://')
```

4 Strong Scaling Tests

We tested the parallelization efficiency of our Contrastive Learning (CL) model by means of a strong scaling efficiency approach. Basically, we measured the number of images processed per second as a function of the number of nodes utilized for the run for different batch sizes. We used this metric to measure the scaling efficiency of the model processing different mini-batch sizes with different number of nodes. The efficiency is the quotient between the real and ideal speedup, `Speedup = p(n)/p(1)`, where `p(n)` is the performance–in processed images per second–when `n` nodes are used.

In Fig. 4 we can see the efficiency for different number of nodes and for different mini-batch sizes. As can be seen in the figure, on the one hand, as expected, efficiency declines as the number of nodes increases. It could be originated by the increase in the communication traffic among ranks located in different nodes (i.e., network originated delays).

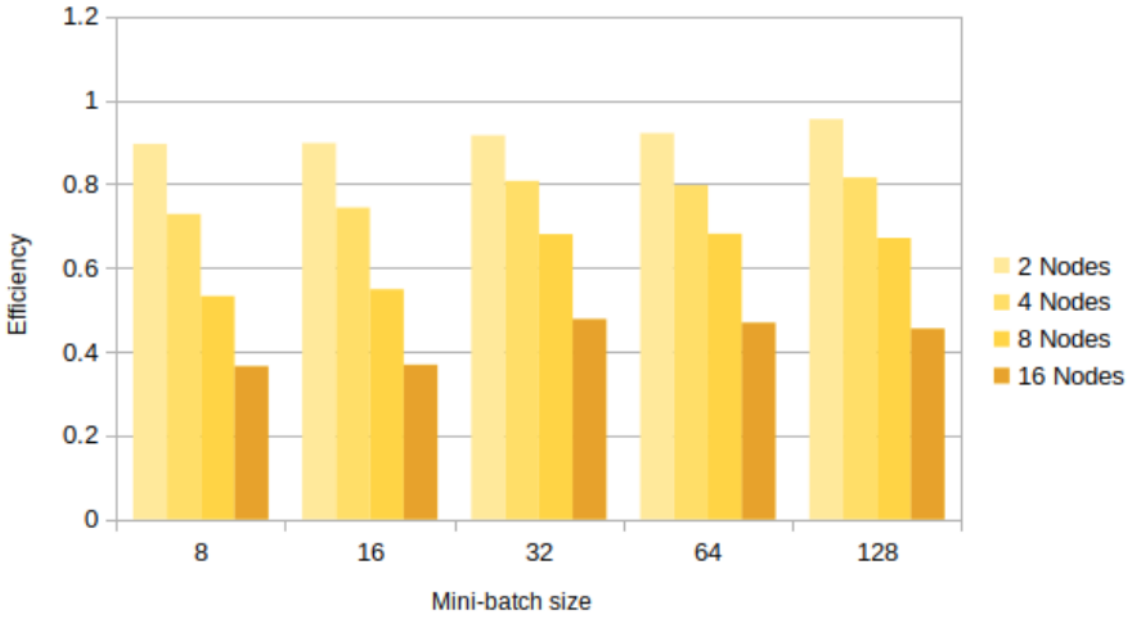

Figure 4. This is a chart showing strong scaling efficiency plots as a function of the mini-batch size with different series for different number of nodes. Efficiency tends to increase with the mini-batch size and decreases when the number of nodes increases.

On the other hand, efficiency tends to improve as the mini-batch size increases. The reason behind such improvement could be related to the I/O bandwidth available in the scratch solid state drives. That is, the larger the batch sizes, the larger the continuous chunks of data to transmit from the disks and the better the data transmission rate.

5 Conclusion

In this paper we introduced our biologically inspired proposal to solve major problems found in today's CV applications. We showed how to achieve portability and reproducibility of our experimental profile

using containerization. We run our applications on up to 16 NVIDIA DGX A100 nodes, each with several GPUs, combining mpi4py with DistributedDataParallel (DDP) and utilizing one solid state scratch device per node in order to accelerate executions. We also showed our approach to develop a foveator using NVIDIA Data Loading Library (DALI) with two pipes: one for loading plus decoding and the other for augmenting. Finally we run a series of strong scaling efficiency tests using the number of images processed per second by our implementation as a performance metric. Results show clear variation trends with respect to the number of nodes and the mini-batch sizes which will be further analysed in future works utilizing profiling tools.

Acknowledgement

The authors would like to thank NVIDIA Data Loading Library (DALI) support staff for all the invaluable assistance in the implementation of their ideas using the library. The authors would also like to thank the data science staff at the Argonne Leadership Computing Facility (ALCF), especially Corey Adams and Huihuo Zheng for their invaluable help in implementing these research models on Argonne machines. This work used resources of the Argonne Leadership Com- puting Facility, which is a U.S. Department of Energy, Office of Science User Facility supported under Contract DE-AC02-06CH11357.

ORCID iD

Dario Dematties https://orcid.org/0000-0002-8726-7837
Silvio Rizzi https://orcid.org/0000-0002-3804-2471
George K. Thiruvathukal https://orcid.org/0000-0002-0452-5571

References

Alahyane, N., Lemoine- Lardennois, C., Tailhefer, C., Collins, T., Fagard, J., & Doré-Mazars, K. (2016, January). Development and learning of saccadic eye movements in 7- to 42-month-old children. *Journal of Vision, 16*(1), 6, 1-12. https://doi.org/10.1167/16.1.6

Canfield, R. L., & Haith, M. M. (1991). Young infants' visual expectations for symmetric and asymmetric stimulus sequences. *Developmental Psychology, 27*(2), 198-208. https://doi.org/10.1037/0012-1649.27.2.198

Canfield, R. L., & Kirkham, N. Z. (2001). Infant Cortical Development and the Prospective Control of Saccadic Eye Movements. *Infancy, 2*(2), 197-211. https://doi.org/10.1207/S15327078IN0202_5

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. (2020, May 28). *arXiv:2005.12872v3 [cs.CV]. End-to-End Object Detection with Transformers.* https://doi.org/10.48550/arXiv.2005.12872

Castro, D. C., Walker, I., & Glocker, B. (2020). Causality matters in medical imaging. *Nature Communications, 11*(3673), 1-10. https://doi.org/10.1038/s41467-020-17478-w

Castro, M., Expósito-Casas, E., López-Martín, E., Lizasoain, L., Navarro-Asencio, E., & Gaviria, J. L. (2015, February). Parental involvement on student academic achievement: A meta-analysis. *Educational Research Review, 14*, 33-46. https://doi.org/10.1016/j.edurev.2015.01.002

Chen, T., Kornblith, S., Norouzi, M., & Hinton, G. (2020, February 13). A Simple Framework for Contrastive Learning of Visual Representations. In H. Daumé III, & A. Singh (Ed.), *Proceedings of the 37 th International Conference on Machine Learning, PMLR 119*, *119*, pp. 1597-1607. Vienna, Austria. https://doi.org/10.48550/arXiv.2002.05709

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., . . . Houlsby, N. (2021, October 22). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *International Conference on Learning Representations ICLR 2021* (pp. 1-21). Vienna, Austria: OpenReview. https://doi.org/10.48550/arXiv.2010.11929

Hikosaka, O., Nakamura, K., & Nakahara, H. (2006). Basal Ganglia Orient Eyes to Reward. *Journal of Neurophysiology, 95*(2), 567-584. https://doi.org/10.1152/jn.00458.2005

Hsiao, J. H.-W., & Cottrell, G. (2008). Two Fixations Suffice in Face Recognition. *Psychological Science, 19*(10), 998-1006. https://www.jstor.org/stable/40064836

Ikeda, T., & Hikosaka, O. (2003, August 14). Reward-Dependent Gain and Bias of Visual Responses in Primate Superior Colliculus. *Neuron, 39*(4), 693-700. https://doi.org/10.1016/S0896-6273(03)00464-1

Johnson, M. H. (1995). The inhibition of automatic saccades in early infancy. *Developmental Psychobiology, 28*(5), 281-291. https://doi.org/10.1002/dev.420280504

Kato, M., Miyashita, N., Hikosaka, O., Matsumura, M., Usui, S., & Kori, A. (1995, January). Eye Movements in Monkeys with Local Dopamine Depletion in the Caudate Nucleus. I. Deficits in Spontaneous Saccades. *The Journal of Neuroscience, 15*(1), 912-927. https://doi.org/10.1523/JNEUROSCI.15-01-00912.1995

Preuss, M. (2018, December). *Updated: 2018-12-27T08:37:12+00:00, Editorial: What is Edge Computing: The Network Edge Explained.* (J. Leavitt, Ed.) Cloudswards Web site: https://www.cloudwards.net/what-is-edge-computing/

Provis, J. M., Diaz, C. M., & Dreher, B. (1998, March). Ontogeny of the primate fovea:a central issue in retinal development. *Progress in Neurobiology, 54*(5), 549-581. https://doi.org/10.1016/S0301-0082(97)00079-8

Purves, D., Augustine, G. J., Fitzpatrick, D., Hall, W. C., Lamantia, A.-S., McNamara, J. O., & Williams, S. M. (Eds.). (2004). *Neuroscience* (Third ed.). Sunderland, Massachusetts, USA: Sinauer Associates. https://pages.ucsd.edu/~mboyle/COGS107a/pdf-files/Neuroscience.pdf

Ross-Sheehy, S., Reynolds, E., & Eschman, B. (2020). Evidence for Attentional Phenotypes in Infancy and Their Role in Visual Cognitive Performance. *Brain Science, 10*(9), 605, 1-24. https://doi.org/10.3390/brainsci10090605

Ross-Sheehy, S., Schneegans, S., & Spencer, J. P. (2015). The Infant Orienting With Attention Task: Assessing the Neural Basis of Spatial Attention in Infancy. *Infancy, 20*(5), 467-506. https://doi.org/10.1111/infa.12087

Salapatek, P., Aslin, R. N., Simonson, J., & Pulos, E. (1980, December). Infant Saccadic Eye Movements to Visible and Previously Visible Targets. *Child Development, 51*(4), 1090-1094. https://doi.org/10.2307/1129548

Spotorno, S., Malcolm, G. L., & Tatler, B. W. (2014, February). How context information and target information guide the eyes from the first epoch of search in real-world scenes. *Journal of Vision, 14*(2), 7, 1-21. https://doi.org/10.1167/14.2.7

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017, June 12). Attention Is All You Need. *arXiv*(1706.03762 [cs.CL]), 15. https://doi.org/10.48550/arXiv.1706.03762

Weber, R. B., & Daroff, R. B. (1972, March). Corrective movements following refixation saccades: Type and control system analysis. *Vision Research, 12*(3), 467-475. https://doi.org/10.1016/0042-6989(72)90090-9