U.P.B. Sci. Bull., Series C, Vol. 86, Iss. 2, 2024 ISSN 2286-3540

PERFORMANCE ANALYSIS OF MEDICAL IMAGING WORKFLOWS

Cosmin-Gabriel Samoilă[1], Maria Predescu[2], Emil-Ioan Sluşanschi[3]

Magnetic resonance imaging (MRI) is a non-invasive imaging technology that produces three-dimensional detailed anatomical images. Analyzing a detailed image of the brain must be done in order to discover disorders associated with structural changes of the brain. Typically, this type of analysis was done manually by a well-trained anatomist or technician, and consuming valuable time. Technology improvements also impacted this particular domain, with medical image analysis being significantly enhanced by the detailed whole brain segmentation technique, which employs deep learning algorithms. This paper focuses on the SLANT Brain Segmentation application - which uses deep neural networks in order to segment whole brain images. The goal of this paper is to analyse and improve the performance of SLANT in different scenarios, in order to discover the most suitable setup to maximize application efficiency on modern CPU and GPU architectures.

Keywords: image segmentation, optimization, deep learning, high performance computing

1. **Introduction**

Detailed whole brain segmentation is an essential technique in medical image analysis, providing an easy way of measuring brain regions from a magnetic resonance image (MRI). For a long time, the manual analysis of the brain structure, which is a resource and time intensive operation, has been regarded as the standard. Nowadays, automated machine learning algorithms are available to reduce the manual effort [4]. Recently, deep convolution neural network (CNN) has been a method used for whole brain segmentation. The Spatially Localized Atlas Network Tiles (SLANT) method is the approach proposed by *SLANTbrainSeg* that brings into the spotlight the advantages of distributing multiple independent 3D fully convolutional networks (FCN) for high-resolution whole brain segmentation [9].

SLANTbrainSeg – Deep Whole Brain High Resolution Segmentation – is a whole brain segmentation pipeline applied on structural magnetic resonance images which is essential for understanding neuroanatomical-functional relationships [10]. The pipeline consists of three parts: pre-, deep-, and post-processing, as exhibited in Figure 1. All three components are contained in a Docker image that can be

[1]PhD Student, SDIALA Doctoral School, National University of Science and Technology Politehnica Bucharest, Romania, e-mail: `cosmin.samoila@upb.ro`

[2]MSc, National University of Science and Technology Politehnica Bucharest, Romania, e-mail: `predescumaria97@gmail.com`

[3]Professor PhD Eng, National University of Science and Technology Politehnica Bucharest, Romania, e-mail: `emil.slusanschi@upb.ro`

executed independently. The containerized solution has two versions, a Docker image for running the application exclusively on CPU, and a Docker image for running the application on a single GPU instance.

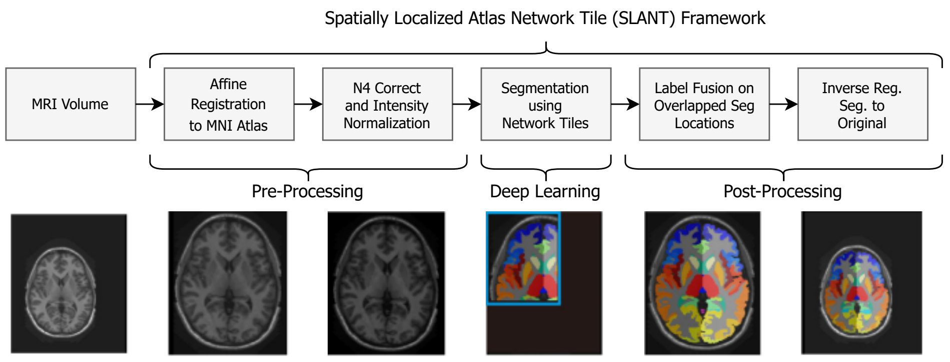

Spatially Localized Atlas Network Tile (SLANT) Framework
MRI Volume
Affine Registration to MNI Atlas
N4 Correct and Intensity Normalization
Segmentation using Network Tiles
Label Fusion on Overlapped Seg Locations
Inverse Reg. Seg. to Original
Pre-Processing
Deep Learning
Post-Processing

Fig. 1. Spatially Localized Atlas Network Tile (SLANT) Framework

SLANT is available in two versions, one that only uses the CPU, and one that can be run using one GPU. Our main objective is to find out the best way to run this application in order to increase its throughput, beginning with the comparison between the existing versions of the program, and exposing the required changes necessary so that it can be executed on multiple GPUs within the same computational node. Subsequently, the application is planned to be deployed in a highly distributed environment.

The distribution of the load was carefully monitored using suitable profiling tools to decide if the resources of the GPU are efficiently utilized when processing the MRIs on multiple GPUs. After the results indicated that processing a single image doesn't utilize all the compute and memory-bandwidth capacity available on the GPU, we tried a different approach to improve the throughput of the application, executing multiple images in parallel on the same GPU

The structure of this paper is as follows: in Section 2 we give an outline of the related work with respect to GPU processing of image segmentation applications as well as execution frameworks on GPU architectures; Section 3 offers an analysis of CPU vs. GPU performance of the SLANT Framework; Section 4 shows the way in which the scheduling of image fragments is deployed on a GPUs using the CUDA programming framework; and in Section 5 we conclude with a discussion of the results as well as a way forward for the current study.

2. **Related work**

The graphical processing unit (GPU) has become an important tool when it comes to reducing the processing time of complex tasks requiring processing of significant amounts of data. Many applications from the medical field can benefit from its advantages. With the vast increase of medical imaging and treatment machines, a huge amount of data is generated in medical physics. The 3D MRIs are more and more accurate due to their increased resolution. Increased resolution means increased processing time, which creates the need for more sophisticated and optimized processing tools.

Image reconstruction and image processing are two computationally challenging tasks suitable for GPU acceleration. Hence, algorithms that could be optimally performed on GPUs are deployed in order to be able to efficiently parallelize the desired workload. For example, the algebraic reconstruction technique (ART) was not convenient for GPUs as each iteration only processes one projection line. To adapt to the GPU capabilities, simultaneous ART (SART) can be used, as the image is updated after the back-projection of an entire projection view [5].

`SLANTbrainSeg` is executed over large datasets of pre-processed MRIs of small sizes and it relies on deep learning frameworks, such as TensorFlow [7] to optimize their execution on a single GPU.

As we will see in the case of *SLANTbrainSeg*, many real-world applications have a low GPU utilization. Most of the time, a single process uses the context of one GPU and blocks the compute resources of the GPU, using them on its own. Sharing these resources among multiple kernels could improve the GPU utilization and would result into an improved performance of the application and also into an increased throughput. To share the context of one GPU, recent NVIDIA GPU architectures implement two features: Hyper-Q and MPS (Multi-Process Service). Hyper-Q allows CUDA kernels to be processed concurrently on the same GPU, while MPS is a feature that enables the overlapping of kernel and memcopy operations from different processes on the same GPU [1]. Knowing about these two features supported by NVIDIA-GPU, we can talk about Slate [3], a workload-aware GPU multiprocessing framework which represents a solution that takes advantage of GPU resource sharing while also managing the contention that may be harmful for the application performance. Slate is a cost-effective software solution for enabling GPU resources sharing and also implements workload-awareness to minimize resource contention.

Another framework that facilitates GPU resources sharing is Mystic, a scheduler that enables co-execution of applications on GPU-based clusters and cloud servers [8]. A particularity of Mystic is that it detects interference between applications using learning-based analytical models.

In the current research we propose an approach that uses CUDA MPS to schedule multiple image fragments over the same GPU, thus leading to an increased throughput of the GPU architecture, unlike the approaches offered in the Hyper-Q and Mystic frameworks described in this section.

3. **Performance Analysis of SLANT**

As mentioned in the previous section, *SLANTbrainSeg* application can run both on CPU and single GPU. In this section, we will begin by comparing the results for the two existing versions of the app, continuing by describing the optimization approaches we analysed during the research process. Among the options we considered and tested we can mention running the deep learning module on multiple GPUs, analysing the impact of limiting the CPU power on the GPU version, testing on various input images, and running multiple images in parallel on the same GPU using CUDA Multi-Process Service.

In this section, we will analyse each version of the SLANTbrainSeg application. For the CPU version, the program was started using a different number of threads to determine the most efficient alternative. On the other hand, for the GPU version, the application was modified so that it could run on multiple GPUs, and the impact of this change will also be described. Because the input image resolutions may vary, we will also discuss the impact of the image size on the total execution time.

3.1. Impact of Input Size

Regarding the input that SLANT works with, it receives MRI volumes of the brain that are firstly pre-processed using MatLab. Next in the pipe comes the deep learning component that we are trying to optimize, and finally, the result is post-processed, also using MatLab, to obtain the final image of the segmented brain. This kind of image can be found in the OASIS (The Open Access Series of Imaging Studies) database [6]. For the tests described up to this point in the paper, a $37MB$ MRI was utilized as input. As stated before, for this input size, the load was not big enough to see an improvement in the application performance when running on multiple GPUs.

Using a Python script, the size of the images was increased, and we managed to run the pre-processing component on the scaled image. After the pre-processing, the output had the same size as the original image after the pre-processing step. This is due to the normalization process performed as part of the first component. The step was introduced because MRI is a non-scaled imaging technique, meaning the intensities of acquired scans vary across different scanners, and even different scans from the same scanner. Therefore, a regression-based intensity normalization was added to the application [9].

3.2. CPU Version

For the CPU version, the user is able to set the number of threads. Figure 2 illustrates how increasing the number of threads improves the execution time. It has to be mentioned that all the data below refer only to the training time, excluding the pre- and post-processing parts.

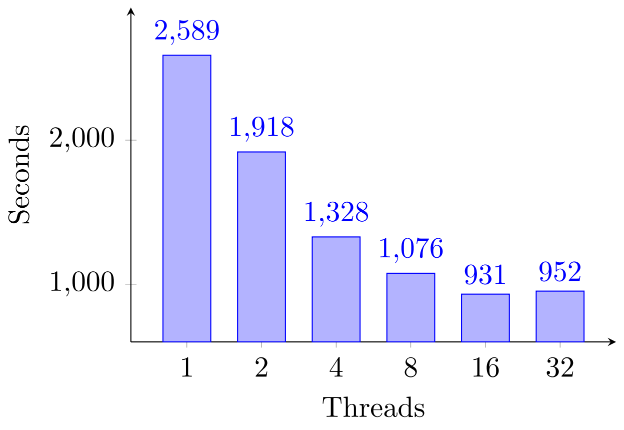


Fig. 2. Processing time for one image - CPU version with threads

The number of threads was increased until we noticed that no improvement was brought to the performance of SLANT. The execution was performed on the NCIT (National Center for Information Technology) cluster of the NUST Politehnica Bucharest on machines which provided 40 threads using hyper-threading (20 physical cores) and $64GB$ of RAM. The best execution time was obtained for the execution with 16 threads, signaling the best trade-off between CPU-load and the size of the data to be processed. A higher number of threads only introduced a parallel overhead without any further performance increase.

3.3. GPU Version

The deep learning module is implemented using the PyTorch framework, an open-source machine learning library based on the Torch library. PyTorch has two ways of splitting models and data across multiple GPUs:

- `nn.DataParallel` splits the model and data between different GPUs and coordinates the training. It uses one process to compute the model and then distributes it to each GPU during each batch. It is easier to use - just wrap the model and run the training script - but it also requires that all the GPUs are on the same node.
- `nn.parallel.DistributedDataParallel` [11] parallelizes the processing of the given module by splitting the input across the specified devices by chunking in the batch dimension and allows the GPUs to be on the same node or spread across multiple nodes. It duplicates the model across multiple GPUs, each of them being controlled by one process.

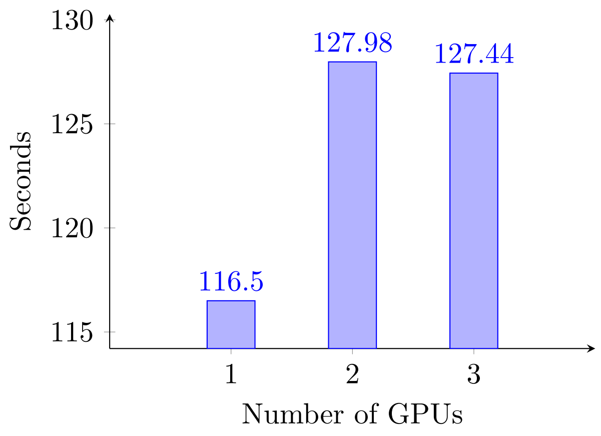


Fig. 3. Processing time for one image - single node, multiple GPUs

The data in Figure 3 is collected after the deep learning module was executed on the `hp-sl.q` queue on a Tesla $K40$ GPU with a memory of $11GB$, and the numbers refer to the computation time for one input image. The changes in performance when increasing the number of GPUs are not visible in this case due to the overhead of data distribution. For more accurate results the program must receive more images as input to be processed in parallel, not just a single fragment. For a better analysis of the results, and especially to discover why the performance of the application is not improved when running on multiple GPUs, we decided to go into detail and analyse the load distribution using a profiling tool.

The breakdown was done for the version using two GPUs, working with the NVIDIA Visual Profiler. By performing this analysis we discovered that for a $37MB$ input image, the load was executed entirely on one GPU, the second one not being used at all. For this input, the load on the used GPU was most of the time below 50%. This result may indicate that the size of the input is not big enough to be divided on both GPUs and the scheduling algorithm used by the `nn.DataParallel` method does not distribute it on the second GPU, using just the first one. We will analyse later on the input options we have and how we could push the limits of the load.

Following these results, two approaches may be taken into consideration: to execute the processing on larger images or to process multiple images in parallel, on the same GPU because processing a single image doesn't utilize all the compute and memory-bandwidth capacity available on the GPU.

Even if the processing time of a single image increases by an approximate 10% when we use a configuration with more than one GPU, having the workload distributed across multiple GPUs is desired. This observation is important if we take into account the workload setup, where batches of multiple images are processed, hence the throughput is increased *close-to-linear* with the number of GPUs.

3.4. Analysis of CPU impact on GPU execution

To better understand the impact of the CPU when the GPU version is running, the CPU power was limited in two ways:

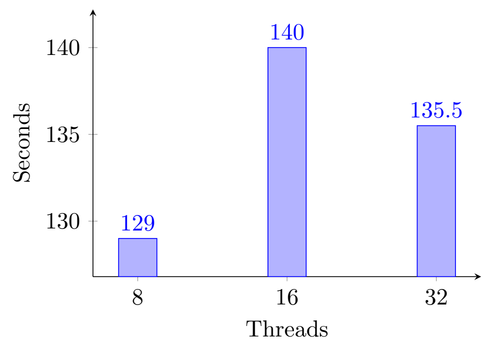
- Using different numbers of CPU threads during the GPU execution.
- Limiting the application to run just on a certain number of CPUs.

For the first option, to change the number of threads, a method from torch library was used, setting the number of threads in `test.py` using `set_num_threads` option provided by PyTorch. Modifying the number of threads did not seem to have a big impact on the execution time for the GPU version, as it can be observed in Figure 4a, the times for the 8, 16, and 32 threads versions are very similar.

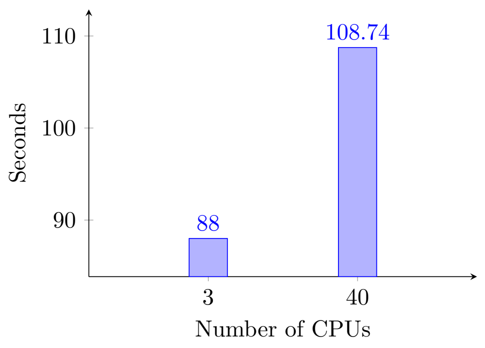
The second option was to limit the CPUs for the application. This was achieved by restricting the available CPUs for the application using the `taskset` command, where just three out of the 40 available CPUs were used to run the GPU version of the SLANT application. Setting the affinity of the process for certain CPUs doesn't affect the performance, the obtained time was better for this method, showing that, even without limiting the number of CPUs, the program was using the same CPU power.

4. Towards High-Throughput Computing

Most of the time, when thinking about improving the performance of an application executed on GPUs, we think about the most efficient way to distribute the workload on multiple GPUs. But there are cases when a single application process doesn't utilize the full compute capacity of even one GPU node. SLANT Brain Segmentation is one of those applications, which for processing a single image, utilizes below 50% of the compute and memory-bandwidth capacity of a GPU with 11GB memory. To be able to use all the available resources and to maximize the throughput of $SLANTbrainSeg$, a couple of scenarios were analysed, including CUDA Multi-Process Service and scheduled processing of segments. We propose an approached meant to improve both the speedup as well as the memory usage of GPUs for the SLANT application.



(a) 2 GPUs changing the number of threads



(b) 2 GPUs limiting the CPUs

Fig. 4. CPU impact on the GPU execution

4.1. CUDA Multi-Process Service

The CUDA Multi-Process Service (MPS) is an alternative implementation of the CUDA Application Programming Interface (API) that allows multiple CUDA processes to share the context of a single GPU [2].

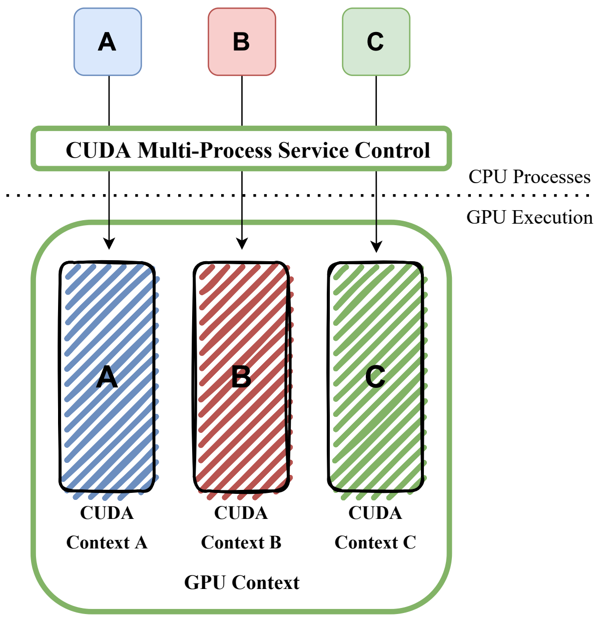


Fig. 5. Multi-Process Service (MPS)

As mentioned in the previous sections, the SLANT pipeline is divided into three parts: the pre-processing, the deep-learning, and the post-processing phases. The deep learning component receives as input a pre-processed image. After the images go through the first module, they have the same size as a result of the normalization process performed as part of this component. The step was introduced because magnetic resonance imaging is a non-scaled imaging technique, meaning the intensities of the brain scan may vary a lot. Therefore, a regression-based intensity normalization was added to the application. Knowing that the input size is constant for the deep learning module and that processing a single image doesn't utilize enough of the computing capacity of one GPU, we decided to explore the available options to maximize the throughput for SLANTbrainSeg by maximizing the resources utilization of one GPU. Using the CUDA Multi-Process Service, the GPU can run multiple independent kernels concurrently as long as there are enough resources available (e.g. registers, shared memory, thread blocks slots, etc.) while a single kernel is running.

4.2. Scheduled Processing of Fragments

To get better flexibility in scheduling the processing of the MRIs, a different method to process the images was taken into consideration. Each 3D MRI is divided into 27 pieces or fragments, as shown in Figure 6, which can be processed separately without affecting the integrity and correctness of the final results. This is a particularity of the SLANT-27 whole brain segmentation method where the segmentation is done using 27 network tiles. Because multiple $3D$ fully convolutional networks (FCN) are used, each one of them is specialized on one part of the brain with smaller spacial variations. As part of the last step of the SLANT pipeline using the label fusion technique, the final segmentation of the brain is obtained [9].

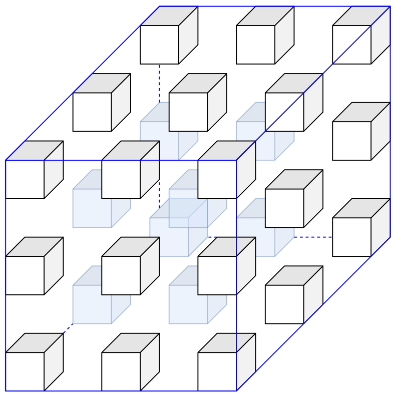


Fig. 6. SLANT-27 Network Tiles

In the approaches described up to this moment, to process one image, we used a script provided in the Docker image of SLANTbrainSeg, that executed sequentially the deep learning module for each piece. To obtain more control over when a certain piece of one image is processed, a custom scheduling algorithm for processing the MRIs so that the throughput is maximized has to be implemented.

5. Conclusion and further work

The findings described in this paper represent a good starting point in the GPU optimization of the applications that perform processing of magnetic resonance images as they have a similar structure. The fact that GPU computing capabilities are not yet efficiently utilized in most data centers, typically incur significant financial costs, that can be reduced with better resource utilization policies. When a single process utilizes a small number of resources of a GPU, a lot of computing power can be wasted as those resources could be used to run other processes as well. The CUDA Multi-Process Service is an alternative implementation of the CUDA API that solves this problem by allowing operations from different processes to overlap on the same GPU.

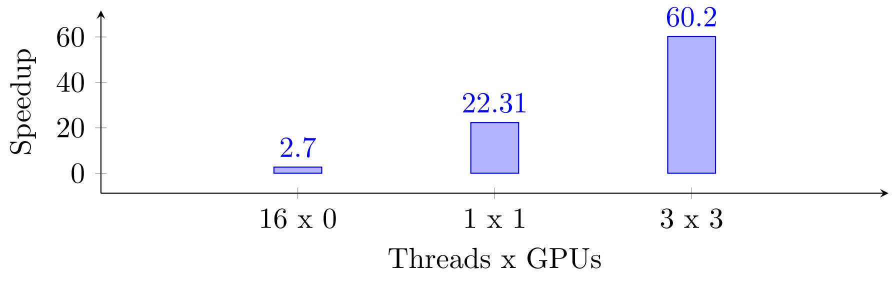


Fig. 7. Speedup for processing 6 images compared to serial execution

As show in Figure 7, comparing the results of the CPU and GPU executions for six images, the multi-threaded execution is 2.7 times faster than the serial one (one thread and no GPUs). If we offload the execution to all three available GPUs of a single machine, the execution time decreases approximately 60 times compared to the serial run. As a reference, the serial processing of a batch of six MRI images, on the machine that we have previously described, takes 4 hours and 20 minutes and the fastest configuration that uses three threads and three GPUs takes 4.5 minutes.

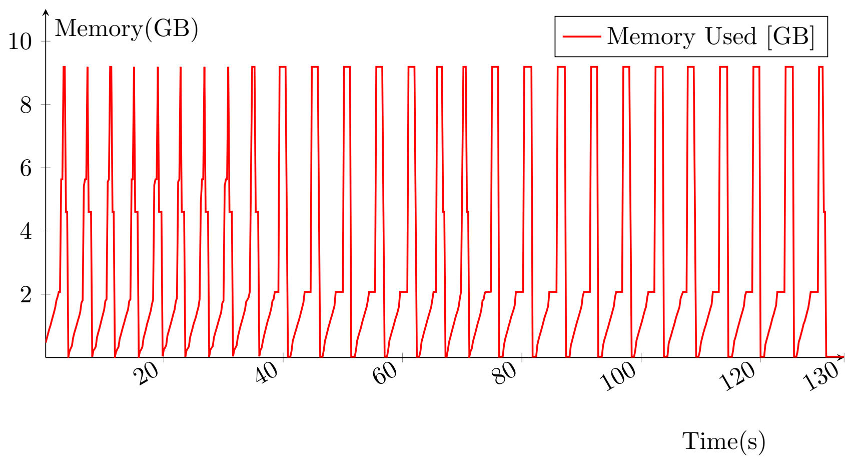


Fig. 8. Memory usage while running two images on a single GPU

As can be seen in Figure 8, at the peak point in the processing of one fragment of the image, more than 80% of the memory of the GPU is utilized. As a consequence of the memory being the bottleneck of our workload, the average GPU utilization is still rather low, hence the processing of multiple images in parallel will have a poor resource utilization or the execution will be halted because of memory-overflow. To solve this issue, we are planning to implement a custom scheduling algorithm for image fragments. The heuristic will have to take into account the instant load and memory utilization for each GPU when the processing of a new fragment will be scheduled. One might notice that processing independent MRI images is embarrassingly parallel so, having a *resource-aware* scheduling algorithm, will allow us to efficiently deploy the current workload in modern HPC environments.

Acknowledgment

We would like to acknowledge the contribution and fruitful collaboration with Dr. Ana Gainaru from the Oak Ridge Leadership Computing Facility.

REFERENCES

[1] Improving GPU Utilization with Multi-Process Service (MPS). https://on-demand.gputechconf.com/gtc/2015/presentation/S5584-Priyanka-Sah.pdf. Accessed: 2022-06-15.

[2] Nvidia Documentation: Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2022-06-15.

[3] *Allen, Tyler and Feng, Xizhou and Ge, Rong.* Slate: Enabling workload-aware efficient multiprocessing for modern gpgpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 252–261, 2019.

[4] *Bruce Fischl and David H. Salat and Evelina Busa and Marilyn Albert and Megan Dieterich and Christian Haselgrove and Andre van der Kouwe and Ron Killiany and David Kennedy and Shuna Klaveness and Albert Montillo and Nikos Makris and Bruce Rosen and Anders M. Dale.* Whole brain segmentation: Automated labeling of neuroanatomical structures in the human brain. *Neuron*, 2002.

[5] *Guillem Pratx and Lei Xing.* Gpu computing in medical physics: A review. In *Medical Physics - The International Journal of Medical Physics Research and Practice*. American Association of Physics in Medicine, 2011.

[6] *LaMontagne, Pamela J. and Benzinger, Tammie LS. and Morris, John C. and Keefe, Sarah and Hornbeck, Russ and Xiong, Chengjie and Grant, Elizabeth and Hassenstab, Jason and Moulder, Krista and Vlassenko, Andrei G. and Raichle, Marcus E. and Cruchaga, Carlos and Marcus, Daniel.* Oasis-3: Longitudinal neuroimaging, clinical, and cognitive dataset for normal aging and alzheimer disease. *medRxiv*, 2019.

[7] *Martín Abadi and others.* Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.

[8] *Ukidave, Yash and Li, Xiangyu and Kaeli, David.* Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362, 2016.

[9] *Yuankai Huo and Zhoubing Xu and Yunxi Xiong and Katherine Aboud and Prasanna Parvathaneni and Shunxing Bao and Camilo Bermudez and Susan M. Resnick and Laurie E. Cutting and Bennett A. Landman.* 3d whole brain segmentation using spatially localized atlas network tiles. *NeuroImage*, 194:105–119, 2019.

[10] *Yunxi Xiong and Yuankai Huo and Jiachen Wang and L. Taylor Davis and Maureen McHugo and Bennett A. Landman.* Reproducibility evaluation of SLANT whole brain segmentation across clinical magnetic resonance imaging protocols. In Elsa D. Angelini and Bennett A. Landman, editors, *Medical Imaging 2019: Image Processing*, volume 10949, pages 729 – 736. International Society for Optics and Photonics, SPIE, 2019.

[11] PyTorch Documentation: DataParallel and DistributedDataParallel methods. https://pytorch.org/docs/stable/nn.html#dataparallel-layers-multi-gpu-distributed. Accessed: 2022-06-15.