Global Journal of Computer Science and Technology: C
Software & Data Engineering
Volume 22 Issue 2 Version 1.0 Year 2022
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Capillary X: A Software Design Pattern for Analyzing Medical Images in Real-time using Deep Learning

By Maged Abdalla Helmy Abdou, Paulo Ferreira, Eric Jul & Tuyen Trung Truong

University of Oslo

Abstract- Recent advances in digital imaging, e.g., increased number of pixels captured, have meant that the volume of data to be processed and analyzed from these images has also increased. Deep learning algorithms are state-of-the-art for analyzing such images, given their high accuracy when trained with a large data volume of data. Nevertheless, such analysis requires considerable computational power, making such algorithms time- and resource-demanding. Such high demands can be met by using third-party cloud service providers. However, analyzing medical images using such services raises several legal and privacy challenges and do not necessarily provide real-time results. This paper provides a computing architecture that locally and in parallel can analyze medical images in real-time using deep learning thus avoiding the legal and privacy challenges stemming from uploading data to a third-party cloud provider. To make local image processing efficient on modern multi-core processors, we utilize parallel execution to offset the resourceintensive demands of deep neural networks. We focus on a specific medical-industrial case study, namely the quantifying of blood vessels in microcirculation images for which we have developed a working system.

GJCST-C Classification: *DDC Code: 020.3 LCC Code: Z1006*

Strictly as per the compliance and regulations of:

Capillary X: A Software Design Pattern for Analyzing Medical Images in Real-time using Deep Learning

Maged Abdalla Helmy Abdou [α], Paulo Ferreira [σ], Eric Jul [ρ] & Tuyen Trung Truong [ω]

Abstract- Recent advances in digital imaging, e.g., increased number of pixels captured, have meant that the volume of data to be processed and analyzed from these images has also increased. Deep learning algorithms are state-of-the-art for analyzing such images, given their high accuracy when trained with a large data volume of data. Nevertheless, such analysis requires considerable computational power, making such algorithms time- and resource-demanding. Such high demands can be met by using third-party cloud service providers. However, analyzing medical images using such services raises several legal and privacy challenges and do not necessarily provide real-time results. This paper provides a computing architecture that locally and in parallel can analyze medical images in real-time using deep learning thus avoiding the legal and privacy challenges stemming from uploading data to a third-party cloud provider. To make local image processing efficient on modern multi-core processors, we utilize parallel execution to offset the resourceintensive demands of deep neural networks. We focus on a specific medical-industrial case study, namely the quantifying of blood vessels in microcirculation images for which we have developed a working system. It is currently used in an industrial, clinical research setting as part of an e-health application. Our results show that our system is approximately 78% faster than its serial system counterpart and 12% faster than a master-slave parallel system architecture.

I. Introduction

Early attempts to address the problem of running demanding computational algorithms in tightly constrained environments emerged in the 1980s [1]. Performance limitations became apparent with the rise of processing big data using deep learning (DL) techniques because DL requires large amounts of computational power [2], [3]. Such limitations included the under-utilization of the available computing resources to execute processes introducing undesirable delays [4]. These limitations are still prominent when real-time results are desired in tightly constrained environments (i.e., clinical environments). Furthermore, using third-party cloud services to rent computing resources is risky due to General Data Protection Regulation (GDPR) [5]. These regulations effectively limit clinicians to local computing resources, such as laptops and PCs approved for use at hospitals.

Author α σ ρ: Department of Informatics University of Oslo, Norway. e-mail: magedaa@uio.no
Author ω: Department of Mathematics University of Oslo, Norway.

This paper aims to design, implement, and evaluate a software package that can analyze medical images using deep learning in a local environment as to mitigate the risk of breaching GDPR rules while still getting results in real-time. We focus on a specific industrial, medical case study: the quantification of blood vessels in microcirculation images captured by using in-clinic, hand-held cameras with microscope lenses. The quantified value is called capillary density or blood vessel density. This value is of high clinical relevance because the fluctuation of this value can be used as an early marker to indicate an organ failure, and the severity of the change might predict the chances of the patient surviving [6]–[12].

The requirements of our system were established by interviewing a set of medical doctors and surgeons who spent several years in the microcirculation analysis field (associated with ODI Medical AS, a MedTech company responsible for the e-health industrial application). The main requirements for a production-grade system for the quantification of blood vessels analysis captured by a real-time camera are:

1. The system must be able to analyze a microcirculation image (1920x1080) in real-time (one second or less);
2. The system must have low power consumption so that it can be used in battery-powered devices in hospitals; and
3. The system must be built on top of a popular, widely used programming language and framework (e.g., Python and Tensorflow) running on standard hardware.

To the best of our knowledge, no previous work on microcirculation analysis reported using parallel frameworks to calculate the capillary density in under 1 second for a frame with a resolution of 1920x1080 on a CPU using deep learning with an accuracy of ~85%. The medical doctors proposed this accuracy value to outperform the accuracy achievable by a trained clinician. Previous systems in the literature that achieve a comparable or higher accuracy needed a GPU that is not available in typical low-power computers approved for use in hospitals. The developed system runs in an industrial, clinical environment on a standard low cost

computer utilizing all the available resources and meets the requirements listed above. This paper does not focus on developing the deep learning algorithm but rather the deployment of the deep learning algorithm. The algorithm used in this paper achieves an accuracy of~85%, and is described in a previous paper [13].

In Section II, we present the work related to our paper including a literature survey on the relevant parallel frameworks and existing systems that were built to analyze micro-1 circulation images. In Section III, we present the proposed architecture for our package along with two baseline systems that we used to benchmark our proposed architecture against. In Section IV, we present how we implemented our system. In Section V, we present the evaluation criteria that have been used to evaluate our system and benchmark our proposed system against a baseline serial system and a baseline parallel system with the presented criteria and discuss our results. In Section VI, we present our conclusion.

II. Related Work

This section presents the literature review on current parallel frameworks and existing systems built to calculate capillary density.

a) Parallel Frameworks

Hadoop [14] gained recognition in 2004 and provides a framework for distributed storage and the processing of big data. It splits large blocks of data into a Hadoop Distributed File System (HDFS) which is based on Google's file system (GFS) and stores data across clusters [15], [16]. HDFS uses data locality, allowing clusters and nodes to manipulate data, making it faster than conventional high-performance computing [17].

MapReduce then processes the data stored on HDFS [18]. MapReduce has a master job tracker and one per cluster to schedule jobs, manage resources, and re-execute processes when a node fails [17]. HDFS and MapReduce are two modules built to store and process big data reliably. However, the main drawback of Hadoop is that it cannot deal with big data real-time stream processing; therefore, Apache Spark was introduced [19] was introduced in 2010.

Some benchmarks show that Spark is three times faster than Hadoop [20]. This increase is because Spark can load and process data using RAM instead of the two-stage access paradigm introduced by MapReduce [19]. Spark outshines MapReduce when it comes to real-time processing [21], [22]. Furthermore, the ease of programming on Spark with Scala [23], Java [24] and Python [22] makes it relatively easy to adapt instead of MapReduce, which can be programmed only in Java. Spark provides a unified processing system instead of several isolated applications that do not share the state amongst each other [25]. Although Spark was designed to outperform MapReduce processing, its the fundamental limitation is the complexity involving asynchronous execution and the compatibility issues introduced when integrating it into the deep learning lifecycle [26].

Dask [27] was introduced in 2014 and is a parallel computing library that uses dynamic task scheduling to leverage multi-core processors and High-Performance Computing (HPC) clusters. Instead of loading all data into RAM, Dask pulls data into RAM in chunks and throws away intermediate values as soon as possible, freeing more memory to process more data [27]. While Spark can be seen as an extension to the MapReduce paradigm, Dask is a generic task scheduling system that handles complex dimensional arrays [28]–[30]. Both Dask and Spark leverage acyclic graphs, but the map stage of Dask can represent more complex algorithms than Spark [31]. Thus, Dask can parallelize sophisticated algorithms without excess memory usage [29]. Moreover, Spark does not natively support multi-dimensional arrays as Dask does [30], [32]. This advantage makes Dask lightweight and smaller than Spark, and Dask integrates natively with the numeric Python ecosystem. However, Dask is not fully compatible with TensorFlow, and deep learning algorithms as the framework focuses on Data science libraries like Pandas and Numpy.

Orleans [33] is an actor system that provides highly available concurrent distributed systems. The main drawback is how the system reacts to a data failure event. Developers must manually create checkpoint actor states and intermediate responses to restore stateful actors [34]. While this does not affect the performance, it can bring some overhead when developing a system to handle failure events.

Tensorflow [35] is an ecosystem of machine learning and deep learning tools that leverage CPUs and GPUs while training. However, it provides limited support when deploying it to serve users because it does not fully support responses when a task is completed or when a fault is detected. One way to perform this activity is to wrap the Tensorflow Model in a flask service and serve the model [36]. However, this becomes unmanageable when scaling with different models. Tensorflow serving [37] was introduced to deploy models in production environments but has to be used in conjunction with traditional web servers, which introduces additional latency.

With the introduction of deep learning techniques [38], which consists of several millions of parameters to compute, wrapping deep learning models with traditional servers is no longer sufficient. Compared to traditional models, deep learning models are computationally intensive and have a response time of tens of a millisecond or greater [39]; thus, there is a need for efficient parallelizing to reduce the response time.

Frameworks such as MapReduce [17] and Spark [40] are not suitable for models serving in real-time because they were designed and built for batch processing. Furthermore, they are not suitable for large numbers of small transactions because of the considerable time overhead that they require for instantiation. Dask [27] and Tensorflow [35] provide a complex and very little support for model serving [26].

It is possible to set up different parts of different frameworks together to have a system that can serve a deep learning model. However, the compatibility and maintenance of these different frameworks increase the technical complexity. Unfortunately, deploying deep learning models into production is still not a straightforward endeavor.

b) Existing Microcirculation Analysis Systems

This section presents the current work on systems that calculate capillary density from microcirculation images.

As briefly mentioned at the end of the introduction, none of the existing works mentioned on microcirculation analysis reported using parallel frameworks to calculate the capillary

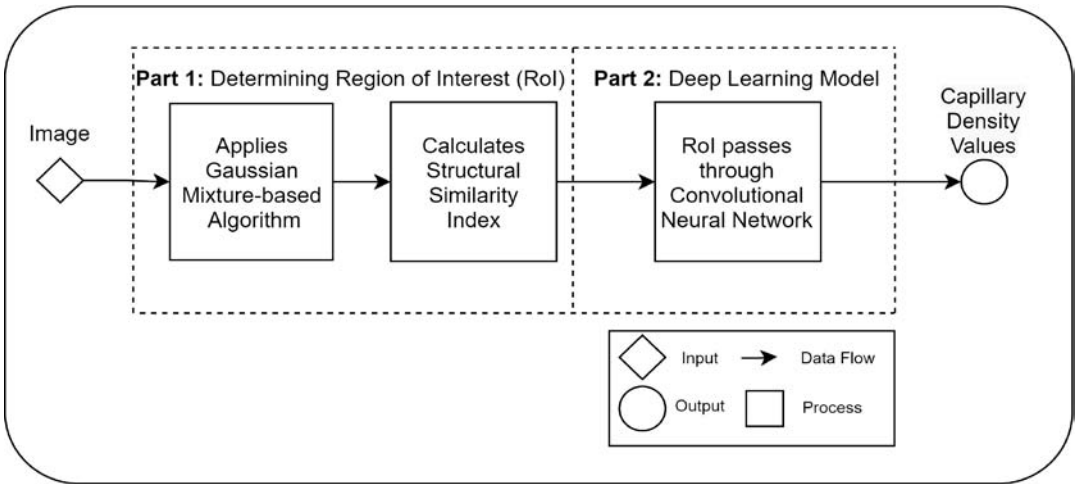


Fig. 1: The diagram shows the code encapsulated in a core on a computer. This code is replicated across each core to achieve parallelism. This code calculates the capillary density from a microcirculation image. The architecture consists of two parts: i) first determining the RoI using traditional computer vision algorithms and ii) then using deep learning to classify if the RoI contains a capillary

density in under ~1 second for a frame with a resolution of 1920x1080 on a CPU using deep learning with an accuracy of ~85%. Those who exceeded this accuracy used a GPU which is not readily available in a clinical environment.

Cynthia Cheng et al. [41] takes a three-step approach to quantify capillary density. First, they apply an image enhancement process to darken the capillaries and lighten the background. They then flatten the image using 2D filters and raise the image's contrast. The image is then despeckled using a 7x7 filter. They then adjust the histogram of the image to a best-fit model The second step involves manually selecting the capillary as a target object. They then select the background as a reference. The algorithm then selects the rest of the capillaries and excludes the images. A macro is then created from this process, which can be applied to other images with similar characteristics. As described, this involves several steps, including the manual user intervention; therefore cannot provide results in less than 1 second.

A. Tama et al. [42] uses binarization followed by skeleton extraction and segmentation to quantify the capillaries. The first step involves extracting a reference image. The image has to be then manually cropped by the user. The green channel is then extracted from the image to have the highest probability of vessels in it. They then apply a top-hat transform to remove unevenness in the background. They then apply the Wiener filtering, a lowpass filter followed by Gaussian smoothing. They then apply Otsu thresholding to segment the image from the background and apply a skeleton extraction method to quantify the capillary. The authors do not report the speed needed to perform these steps.

Sherry G.Clendenon et al. [43] uses a manual method to segment the microvascular structure. The authors do not report the speed or accuracy of their method.

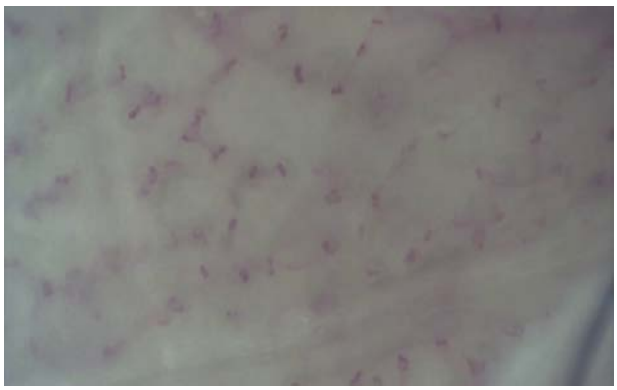
Pavle Prentắsic et al. [44] used a custom neural network to segment the foveal microvasculature. Their neural network consists of three Convolutional Neural Network (CNN) blocks coupled with max-pooling and a dropout layer followed by two dense layers. They reported accuracy of 82.4% at 2 minutes.

R Nivedha et al. [45] used a non-linear Support Vector Machine [46] to classify images. They first started by extracting the green channel since it contains the

relevant information to detect blood vessels. They then performed manual cropping and used adaptive histogram equalization to improve the image's contrast. They then used image enhancement to segment the image using a Gaussian filter followed by OTSU thresholding. They then used PrincipalComponent Analysis(PCA) to extract the features. A Support Vector Machine then performed the classification. They reported accuracy of 83.3% but not the time needed for automated analysis.

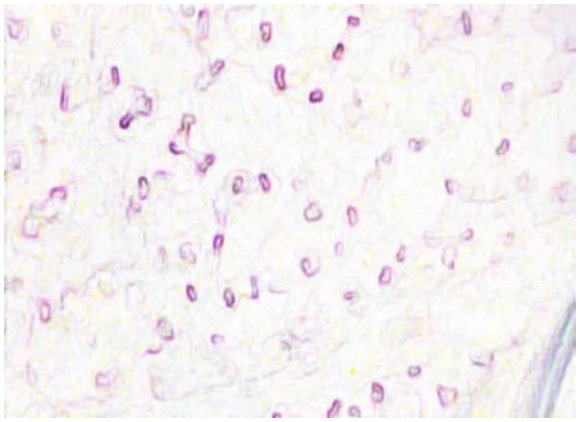
KV Suma et al. [47] used Fuzzy Logic Kernels to classify the images. They started by Fuzzification of the input, followed by the Application of the Fuzzy operator, then aggregating the consequents across the rules, ending with results. They reported an overall accuracy of 83.3% but not the time needed for automated analysis. In their next paper [48], they experimented with different types of machine learning techniques, including Random Forests Classifier, Multinomial Logistic Regression, and CNNs. However, they do not report the timing needed for classifying the blood vessels.

Perikumar Java et al. [49] used a custom form of ResNet18 [50] to quantify capillaries. They used a 10-layer architecture and resized the images to input 224x224x3. They

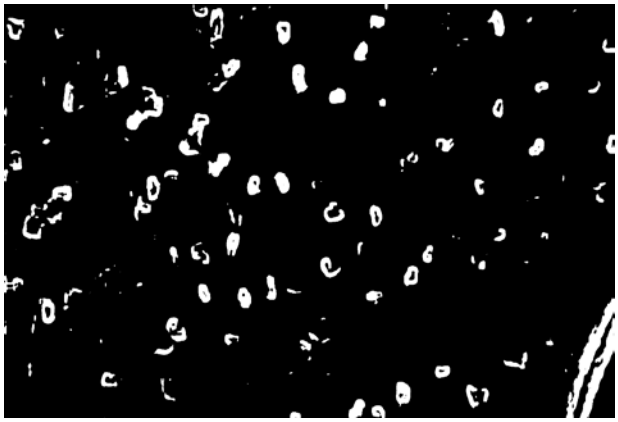


(a)

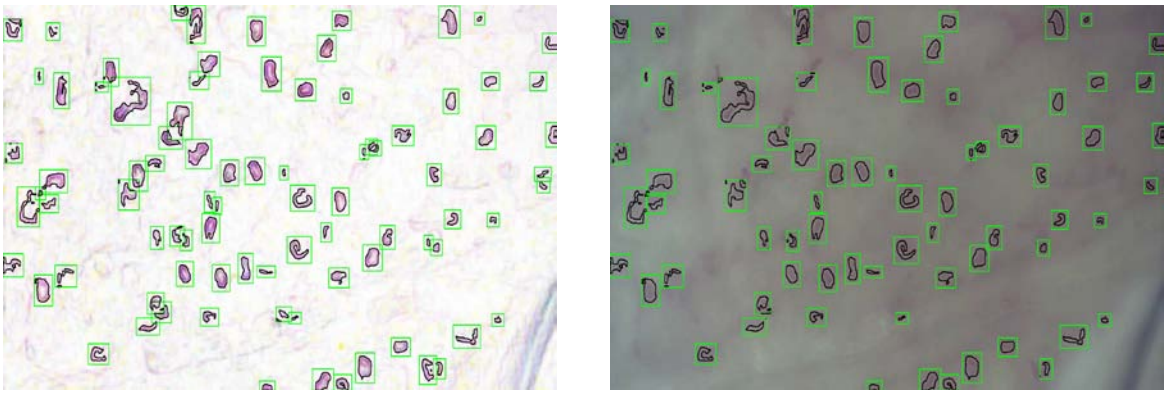
(b)



(c)



(d)



(e) (f)

Fig. 2: (a) This presents a sample of a microcirculation image that is taken as an input to the system (b) The background image calculated using a Gaussian Segmentation Algorithm (c) The segmented area formed by calculating the difference between the original image and the background (d) The Structural Similarity Index calculated from the original frame and the background image (e) The modified image with the capillary area highlighted in black encapsulated within the green bounding box (f) The original image with the capillary area highlighted in black encapsulated within the a bounding box

applied the Adam optimizer and cross-entropy as loss metrics. They trained the NVIDIA GeForce GTX Titan X algorithm and used the PyTorch library. They reported accuracy of 88%. However, such an algorithm is not suitable for a clinical environment due to the high-end GPU required to run it.

F Ye et al. [51] utilized the concept of transfer learning and used the Inception Single Shot Multibox Detector (SSD) [52] to build their neural network. They build their system using Python and Tensorflow with an image resolution of 744 × 482 pixels. They applied data augmentation to the image to increase the number of datasets. The SSD architecture requires GPU to produce results in real-time, making it unsuitable to be used in a clinical requirement with only CPUs available.

YS Hariyani et al. [53] used U-net architecture combined with a dual attention module. They introduced a new method called DA-CapNet, which can analyze microcirculation images. It consists of the encoder and decoder parts. The encoder downsamples the dimension of the information in an image while increasing the number of channels. This step increases the spatial information dimension. They then combine it with a dual attention module which increases the accuracy. The dual attention uses the squeeze and excitation process to extract the blood vessels in the image. The authors resized the image to 256×256 to reduce the processing time and used a Gaussian threshold method with a median blurring filter of kernel size five. The authors reported accuracy of 64% but not the time taken for analyses.

G Dai et al. [54] used a custom neural network similar to Pavle Prentăsic et al. for segmentation. However, G Dai et al. used five CNN blocks instead of three. Hang-Chan Jo et al. [55] used a Attention-UNet architecture [56]. Their method starts by using the CLAHE method and computes several histograms. They then apply the Gamma correction and pass it to the deep neural network. The reported accuracy was 73.20%, but not the time is taken for analysis.

III. Proposed System

This section presents the system's architecture to analyze medical images in parallel, specifically, to calculate the capillary density in a microcirculation image. We start by presenting the DL part (which is based on OpenCV [57] and Tensorflow [35]) and the architecture of our system's parallel part (which is based on Ray [58]).

a) The Deep Learning Algorithm part of the Proposed System

The outline of the deep learning architecture is shown in Figure 1. It consists of two main parts: i) determining the regions of interest (RoIs) where capillaries might exist, and ii) using a CNN for predicting whether these RoIs contain a capillary or not.

The original frame is shown in Figure 2a. The position of the capillaries is determined by first removing the background from the original frame using a Gaussian Mixture-based Background/Foreground Segmentation Algorithm [59]. The background removed is shown in Figure 2b. The structural similarity index measure (SSIM) [60], [61] is applied between the original frame shown in Figure 2a and background image shown in Figure 2b resulting in Figure 2c and Figure 2d. Bounding boxes are formed around the red areas using OpenCV contour method [62]. These bounding boxes are then passed to the CNN for prediction. The RoIs that have been predicted as capillaries have a green bounding box around each one of them along with a black line to highlight the shape of

the capillary. This is shown in Figure 2e and the original image in Figure 2f. The number of pixels within the encapsulated black contour line is summed up and divided by the total number of pixels resulting in the value of the capillary density.

Bounding boxes are formed around the predicted bounding boxes using the OpenCV contour method [62]. These bounding boxes are then passed to the CNN for prediction. The RoIs predicted as capillaries have a green bounding box around them and a black line to highlight the capillary shape. This is shown in Figure 2c and Figure 2d. The number of pixels within the encapsulated black contour line is summed

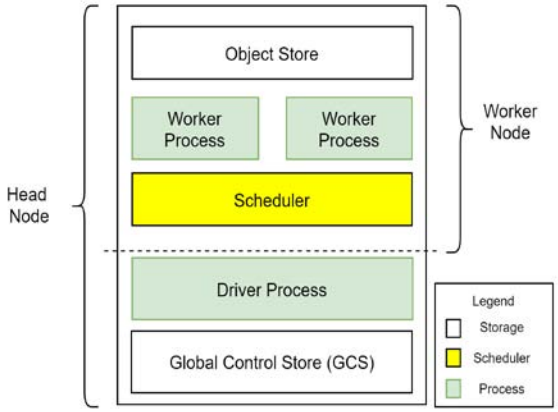


Fig. 3: A Breakdown of the Building Blocks Used to Built the Proposed System

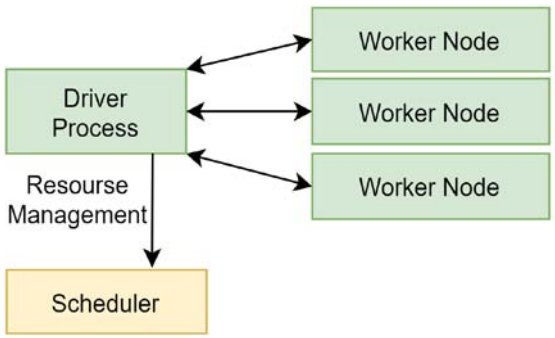


Fig. 4: The data flow view of how the driver process coordinates with the driver process and the workers

up and divided by the total number of pixels resulting in the value of the capillary density.

The CNN consisted of three blocks of Conv2D. The first Conv2D consisted of 32 filters, the second Conv2D consisted of 64 filters, and the third Conv2D consisted of 128 filters with a block of Maxpooling2D. All the Conv2D blocks have a filter of 3x3 shape. Two dense layers of 128 neurons follow the Conv2D blocks, 64 neurons, and two neurons. The Rectified Linear Unit (ReLu) [63] activation function is used for the whole network except the last neuron layer, which used a softmax activation function [64]. This network has been trained on ~11,000 images of capillaries captured by trained professionals in a clinical setting[1]1. The details and the specificity of the algorithms and data used to train the algorithm can be found in a previous paper by the same authors [13].

b) The Parallel System part of the Proposed System

This architecture has two types of nodes: the worker nodes and a head node. A worker node consists of the worker process(es), the scheduler, and the object-store. A worker node and a head node anatomy are shown in Figure 3 and the data flow within the components is shown in Figure 4.

A worker process encapsulates the code to be executed and is responsible for task submission and execution of tasks. In our system, the worker node encapsulates the deep learning algorithms. It receives the image to be analyzed and replies whether this image contains a capillary (blood vessel) or not. The scheduler is the resource manager of the worker node. The object store stores and transfers object larger than 100KB. The head node has a Global Control Store (GCS) and a driver process. The GCS is a key-value server that contains objects, actors, and tasks. The driver process submits tasks to the scheduler and keeps track of the objects created with all the nodes. When the code is initiated, an instance of a head node is created. The maximum number of worker processes within this head node is based on the number of parallel modules in the architecture instantiated and the maximum number of cores. Each worker performs both stages: suggesting RoIs and detecting capillaries using the CNN loaded. Each worker returns a single object that contains the frame's density value and is stored in the object-store. The code execution of this architecture is scheduled using the scheduler, and the tasks are performed over a general-purpose Remote Procedure call to the worker processes on top of the Python interpreter. The scheduler then communicates the results via an object transfer protocol. For error handling and fault tolerance, the scheduler retries executing it on the worker processor, if a task fails due to a worker process ending unexpectedly.

Thus, one of the main differences between the proposed system and the baseline parallel system is that the former uses a driver process to manage the workers while the latter uses a controller and a router to manage the worker's tasks. A baseline parallel system uses some controller and router to prevent the worker's potential overloading with tasks, which can cause it to fail. However, these two components (controller and router) can occupy up to two cores for the management of the workers without performing any code execution.

[1] The sponsoring company provided a device that was used to capture this data.

While the proposed system does not reserve any cores to manage the drivers but rather re-executes the code if a worker fails [65].

When the code is instantiated in our proposed system, the worker node loads the CNN as a Tensorflow model. Each worker occupies a logical processor, thread, or core, depending on the CPU architecture; we assume it is a core and instantiate a worker per core. As the number of cores increased, the number of images processed in parallel increased with the number of cores.

We have shown that by combining the deep neural network part with the parallel part, we can process several images at the same time, suggest RoIs and predict whether the bounding boxes have a capillary or not. Furthermore, the number of frames processed in parallel is determined by the maximum number of cores available or the pre-defined the value inserted by the user (assuming it does exceed the number of cores available).

IV. Implementation

Many programming languages can implement a parallel processing framework. Python is the fastest-growing programming language [66], [67] and the preferred programming language for deep learning with Tensorflow [68], [69]. This popularity stems from its design philosophy, where it emphasizes readability and simplicity [66]. Moreover, the number of libraries, various tools, and speedily expanding the industrial community supporting Python made the language attractive [70].

Thus, the proposed package was built on top of Python 3.7 [22], OpenCV 4.5.2[57], Scikit-learn 0.18[71], Ray 1.2[26] and Tensorflow 2.3[35]. The coding and evaluation were done in Pycharm Professional 2021.1 on a Windows 10 operating system. The system can be installed, modified, and used by following the instructions in the readme file on the Github repository (www.github.com/magedhelmy1/CCGRID 2022 parallel system for image analysis).

To use the system, the user can clone the package from the Github repository and import it in their Python environment.

V. Evaluation and Discussion

In this section, we compare the baseline serial architecture, the baseline parallel architecture, and the proposed system with each other using the following three metrics: execution time, speedup, and CPU usage. We show that the proposed Python system is 78% faster than the serial system and 12% faster than the baseline parallel architecture. These three metrics are standardized markers to quantify a system performance [72]. We use these three metrics to compare our proposed approach to a serial and parallel system with the same deep neural network. We show that the proposed system meets the requirements mentioned in Section I and supersedes both the baseline serial system and the baseline parallel system in execution time, speedup, and CPU usage. The proposed system, serial counterpart, and parallel counterpart had the same CNN model and microcirculation images. We evaluated the three systems by taking the average time to calculate capillary density per image for a set of 100 images, which is an arbitrary number we chose to reduce the margin of error and ensure our calculations' accuracy.

a) Execution Time

To calculate how much one architecture was faster compared to the other, we used Equation 1, where ET denotes execution time.

$$\frac{SlowerET - FasterET}{SlowerET} = \%Faster \quad (1)$$

The execution time metric measures the average time needed to calculate a single image's capillary density. We used 100 images in each architecture to reduce the measurement error margin. The execution time of each architecture is the following:

1. Baseline serial architecture — one second per frame;
2. Baseline parallel architecture — 0.25s per frame; and
3. The proposed system — 0.22s per frame. The average

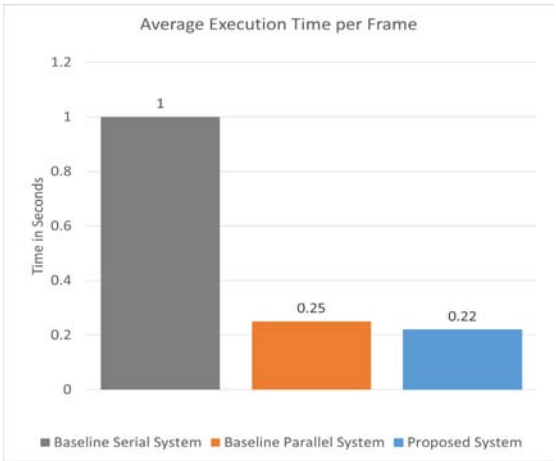


Fig. 5: The execution time of the proposed system against the baseline serial system and the baseline parallel system

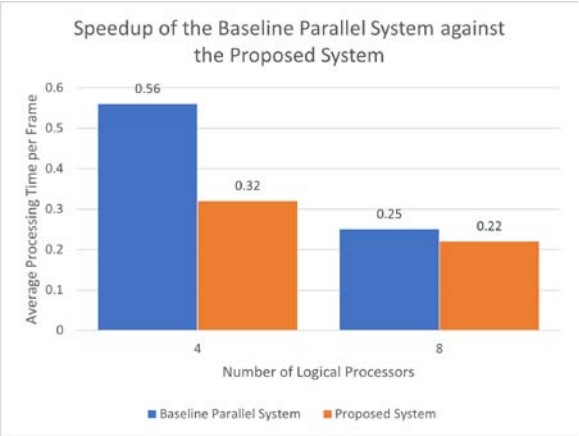


Fig. 6: This graph shows the speedup of the baseline parallel system and the proposed system as the number of cores increases

values were calculated by measuring the time to process a frame in a set of 100 microcirculation images. The execution time of the three architectures is presented in Figure 5.

In short, our results show that our proposed system is 12% faster than the baseline parallel architecture and 78% faster than its baseline serial architecture.

b) Speedup

This metric calculates the speed gain by the system as the number of cores increases. For the baseline serial architecture, the execution time is one second regardless of the number of cores available (indicating that the system is not scalable). The average execution time of processing one frame for the baseline parallel system and the proposed system is shown in Figure 6.

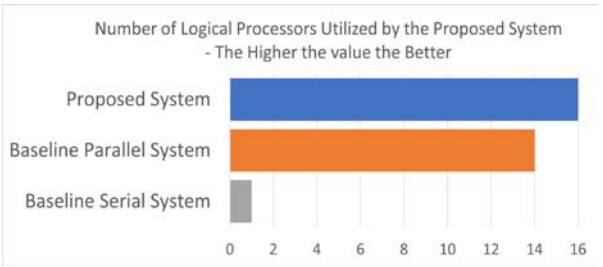


Fig. 7: This graph shows the number of cores used by each architecture to process a frame. The more used at any instance the better since this shows how efficient the system is at utilizing all the resources available to it.

The baseline parallel system processed a frame on average in 0.56 seconds with four cores, while the proposed system processed a frame in 0.32 seconds. The proposed system processes a frame 68% faster than the baseline serial architecture and ~43% faster than the baseline parallel architecture. As the number of cores doubles, the proposed system gains an additional ~31% during the baseline parallel architecture gains an additional ~55%. In both cases, the proposed system outperforms the baseline parallel architecture. One of the main reasons the proposed system outperformed a baseline parallel system with a master-slave architecture is that a masterslave architecture can reserve up to two cores to manage the other parallel cores. In contrast, the proposed system does not reserve any cores beforehand. In this way, we free up the computer cores to focus on processing images rather than purely handling requests. Thus, the proposed system gains more speedup than the baseline parallel system. We can conclude that the proposed system has the recommended architecture for running deep neural networks on a single machine.

c) System Resource Utilization - CPU Usage

This metric measures the number of cores used to process the medical images using deep learning.

With the baseline serial architecture, only one cores is utilized per frame due to the Python Global Interpreter Lock's limitation. With the baseline parallel architecture, it is always two less than the available number of cores because it always reserves these two for the management of the parallel workers. Each cores in the proposed system is allocated a task where each task processes a frame. Thus, the proposed system is most efficient on a single machine with a multi-core. A graph showing the number of cores used by each architecture is shown in Figure 7.

d) System Generalization

Our system functions and classes were built using modular design patterns. This design philosophy means that the user can replace the DL part of our system with their algorithm by simply pointing the function in our code to their algorithm. The details of this are highlighted in the README file in the GitHub repository. Thus, our package can be generalized to analyze images using a DL model of the user's choice in parallel. The system will automatically scale to the number of cores available without the user having to worry about experiencing issues with dependency, integration, resource utilization, and speedup.

VI. Conclusion

This paper presented a software package that can analyze medical images using DL locally. Our proposed system can efficiently use all local resources because it utilizes parallel execution to offset the resource-intensive demands of using a deep neural network. The proposed system is of high clinical relevance because monitoring changes in capillary density can be used to locate early markers indicating organ failure. The severity of the change in capillary density might predict whether or not the patient survives. Furthermore, clinical researchers do not risk uploading patient data to a third-party cloud provider to use a deep neural network to automatically analyze their images.

Our experiments show that our system provides an optimal design for using deep learning models running on a multicore single machine for image analysis. We benchmarked our system with a baseline serial architecture and a baseline parallel architecture using standardized evaluation metrics: execution time, speedup, and CPU usage. These metrics are used to calculate the performance of a system. Our results indicate that the proposed system is approximately 78% faster than its baseline serial system counterpart and 12% faster than a baseline parallel system.

As demonstrated by our evaluation criteria, our system exhibits an acceptable industrial performance compared to the other two presented baseline systems. This argument is further strengthened because our system is currently used as a product in an industrial setting to calculate and track capillary changes in patients with pancreatitis, COVID-19, and acute heart diseases. The clinical researchers welcomed using this system to analyze their medical images locally. This acceptance was mainly due to the system reducing analysis time and removing the risks of uploading the data to a thirdparty cloud provider.

Our code has been made public as an open-source project in a GitHub repository for testing and usage by other clinical users. The users can import the package into their Python environment and immediately start using it. Moreover, users who can clone the code from GitHub can swap our algorithm with theirs, showing that our architecture can be generalized and utilized in the context of other use cases that require image analysis running on a CPU in near real-time. Thus, the generality of our approach can be justified by several other use cases that require image analysis.

Acknowledgment

The authors would like to thank the Research Council of Norway for providing the necessary funds for this project. The research carried out was funded under these two projects; Industrial Ph.D. project nr: 305716 and BIA project nr: 282213. We would also like to thank ODI Medical AS for providing the requirements, testing the system, and integrating it as part of their e-health application.

References Références Referencias

1. R. Duncan, "A survey of parallel computer architectures, "Computer, vol. 23, no. 2, pp. 5–16, 1990.
2. Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," Information Fusion, vol. 42, pp. 146–157, 2018.
3. N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," arXiv preprint arXiv:2007.05558, 2020.
4. S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Parallel i/o optimizations for scalable deep learning," in 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2017, pp. 720–729.
5. J. M. M. Rumbold and B. Pierscionek, "The effect of the general data protection regulation on medical research," Journal of medical Internet research, vol. 19, no. 2, e47, 2017.
6. M. J.S. Parker and N. W. McGill, "The established and evolving role of nailfold capillaroscopy in Connective-Tissue disease," in Connective Tissue Disease – Current State of the Art [Working Title], IntechOpen, 2019.
7. V. Nama, J. Onwude, I. T. Manyonda, and T. F. Antonios, "Is capillary rarefaction an independent risk marker for cardiovascular disease in south asians?" en, J. Hum. Hypertens., vol. 25, no. 7, pp. 465–466, Jul. 2011.
8. B. Fagrell and M. Intaglietta, "Microcirculation: Its significance in clinical and molecular medicine," Journal of internal medicine, vol. 241, no. 5, pp. 349–362, 1997.
9. A. J. H. M. Houben, R. J. H. Martens, and C. D. A. Stehouwer, "Assessing microvascular function in humans from a chronic disease perspective," en, J. Am. Soc. Nephrol., vol. 28, no. 12, pp. 3461–3472, Dec. 2017.
10. T. Wester, Z. A. Awan, T. S. Kvernebo, G. Salerud, and K. Kvernebo, "Skin microvascular morphology and hemodynamics during treatment with venoarterial extra-corporeal membrane oxygen-nation," en, Clin. Hemorheol. Microcirc., vol. 56, no. 2, pp. 119–131, 2014.
11. A. L´opez et al., "Effects of early hemodynamic resuscitation on left ventricular performance and microcirculatory function during endotoxic shock," en, Intensive Care Med Exp, vol. 3, no. 1, p. 49, Dec. 2015.
12. G. Hernandez, A. Bruhn, and C. Ince, "Microcirculation in sepsis: New perspectives," Current vascular pharmacology, vol. 11, no. 2, pp. 161–169, 2013.
13. M. Helmy, A. Dykyy, T. T. Truong, P. Ferreira, and E. Jul, "Capillarynet: An automated system to analyze microcirculation videos from handheld vital microscopy," arXiv preprint arXiv: 2104.11574, 2021.
14. A. Holmes, Hadoop in practice. Manning New York; 2012, vol. 3.
15. D. Borthakur et al., "Hdfs architecture guide," Hadoop Apache Project, vol. 53, no. 1-13, p. 2, 2008.
16. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), Ieee, 2010, pp. 1–10.

17. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.
18. ——, "Mapreduce: A flexible data processing tool, "Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.
19. M. Saouabi and A. Ezzati, "A comparative between hadoop mapreduce and apache spark on hdfs," in Proceedings of the 1st International Conference on Internet of Things and Machine Learning, 2017, pp. 1–4.
20. S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using kmeans," International journal of computer applications, vol. 113, no. 1, 2015.
21. K. Aziz, D. Zaidouni, and M. Bellafkih, "Real-time data analysis using spark and hadoop," in 2018 4th international conference on optimization and applications (ICOA), IEEE, 2018, pp. 1–6.
22. G. Van Rossum et al., "Python programming language.," in USENIX annual technical conference, vol. 41, 2007, p. 36.
23. M. Odersky et al., "An overview of the scala programming language," 2004.
24. K. Arnold, J. Gosling, and D. Holmes, The Java programming language. Addison Wesley Professional, 2005.
25. A. Spark, "Apache spark," Retrieved January, vol. 17, p. 2018, 2018.
26. P. Moritz et al., "Ray: A distributed framework for emerging ai applications," in Symposium on Operating Systems Design and Implementation, 2018, pp. 561– 577.
27. M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in Proceedings of the 14th python in science conference, Citeseer, vol. 126, 2015.
28. M. Dugr´e, V. Hayot-Sasson, and T. Glatard, "A performance comparison of dask and apache spark for dataintensive neuroimaging pipelines," in 2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), IEEE, 2019, pp. 40–49.
29. P. Mehta et al., "Comparative evaluation of big-data systems on scientific image analytics workloads," arXiv preprint arXiv:1612.02485, 2016.
30. D. O. W. .-. D. S. analytics in Python. "Dask document-tation - comparison to spark." (Mar. 2021), [Online]. Available: https://docs.dask.org /en/latest/spark.html.
31. R. Nishihara et al., "Real-time machine learning: The missing pieces," in Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017, pp. 106–110.
32. H. Li, G. Fox, and J. Qiu, "Performance model for parallel matrix multiplication with dryad: Dataflow graph runtime," in 2012 Second International Conference on Cloud and Green Computing, IEEE, 2012, pp. 675–683.
33. S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011, pp. 1–14.
34. P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed virtual actors for programmability and scalability," MSR-TR-2014–41, 2014.
35. T. O. Webpage. "Tensorflow." (Dec. 2021), [Online]. Available: https://www.tensorflow.org/.
36. F. A. Aslam, H. N. Mohammed, and P. Lokhande, "Efficient way of web development using python and flask," International Journal of Advanced Research in Computer Science, vol. 6, no. 2, pp. 54–57, 2015.
37. C. Olston et al., "Tensorflow-serving: Flexible, high-performance ml serving," arXiv preprint arXiv: 1712.06139, 2017.
38. G. Litjens et al., "A survey on deep learning in medical image analysis," Medical image analysis, vol. 42, pp. 60–88, 2017.
39. S. Han, "Efficient methods and hardware for deep learning," Ph.D. dissertation, Stanford University, 2017.
40. M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in 9th Symposium on Networked Systems Design and Implementation 12), 2012, pp. 15–28.
41. C. Cheng, C. W. Lee, and C. Daskalakis, "A reproducible computerized method for quantitation of capillary density using nailfold capillaroscopy," Journal of visualized experiments: JoVE, no. 104, 2015.
42. A. Tama, T. R. Mengko, and H. Zakaria, "Nailfold capillaroscopy image processing for morphological parameters measurement," in 2015 4th International Conference on Instrumentation, Communications, Information Technology, and Biomedical Engineering (ICICI-BME), IEEE, 2015, pp. 175–179.
43. [43] S. G. Clendenon et al., "A simple automated method for continuous fieldwise measurement of microvascular hemodynamics," Microvascular research, vol. 123, pp. 7–13, 2019.
44. P. Prentaˇsi´c et al., "Segmentation of the foveal microvasculature using deep learning networks," Journal of biomedical optics, vol. 21, no. 7, p. 075 008, 2016.
45. R. Nivedha, M. Brinda, K. Suma, and B. Rao, "Classification of nailfold capillary images in patients with hypertension using non-linear svm," in 2016 International Conference on Circuits, Controls, Communications and Computing (I4C), IEEE, 2016, pp. 1–5.

46. W. S. Noble, "What is a support vector machine?" Nature biotechnology, vol. 24, no. 12, pp. 1565–1567, 2006.
47. K. Suma, K. Indira, and B. Rao, "Fuzzy logic based classification of nailfold capillary images in healthy, hypertensive and diabetic subjects," in 2017 International Conference on Computer Communication and Informatics (ICCCI), IEEE, 2017, pp. 1–5.
48. K. Suma, V. Sasi, and B. Rao, "A novel approach to classify nailfold capillary images in indian population using usb digital microscope," International Journal of Biomedical and Clinical Engineering (IJBCE), vol. 7, no. 1, pp. 25–39, 2018.
49. P. Javia, A. Rana, N. Shapiro, and P. Shah, "Machine learning algorithms for classification of microcirculation images from septic and non-septic patients, "in 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, 2018, pp. 607–611.
50. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.
51. F. Ye, S. Yin, M. Li, Y. Li, and J. Zhong, "In-vivo fullfield measurement of microcirculatory blood flow velocity based on intelligent object identification," Journal of biomedical optics, vol. 25, no. 1, p. 016 003, 2020.
52. C. Ning, H. Zhou, Y. Song, and J. Tang, "Inception single shot multibox detector for object detection," in 2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW), 2017, pp. 549–554. DOI: 10.1109/ICMEW.2017.8026312.
53. Y. S. Hariyani, H. Eom, and C. Park, "Da-capnet: Dual attention deep learning based on u-net for nailfold capillary segmentation," IEEE Access, vol. 8, pp. 10 543–10 553, 2020.
54. G. Dai et al., "Exploring the effect of hypertension on retinal microvasculature using deep learning on east asian population," PloS one, vol. 15, no. 3, e0230111, 2020.
55. H.-C. Jo, H. Jeong, J. Lee, K.-S. Na, and D.-Y. Kim, "Quantification of blood flow velocity in the human conjunctival microvessels using deep learning-based stabilization algorithm," Sensors, vol. 21, no. 9, p. 3224, 2021.
56. O. Oktay et al., "Attention u-net: Learning where to look for the pancreas," arXiv preprint arXiv:1804.03999, 2018.
57. G. Bradski and A. Kaehler, "Opencv," Dr. Dobb's journal of software tools, vol. 3, 2000.
58. R. Nishihara and P. Moritz, Ray v2.0.0.dev0. [Online]. Available: https : / /docs .ray. io/en /master /whitepaper. html.
59. P. KaewTraKulPong and R. Bowden, "An improved adaptive background mixture model for real-time tracking with shadow detection," in Video-based surveillance systems, Springer, 2002, pp. 135–144.
60. Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," IEEE signal processing magazine, vol. 26, no. 1, pp. 98–117, 2009.
61. Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," IEEE transactions on image processing, vol. 13, no. 4, pp. 600–612, 2004.
62. openCV, Contour approximation method. [Online]. Available: https : / /docs . opencv . org /3 . 4/d4/d73 /tutorial py contours begin.html.
63. V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in Proceedings of the 27th International Conference on International Conference on Machine Learning, ser. ICML'10, Haifa, Israel: Omnipress, 2010, pp. 807–814, ISBN: 9781605589077.
64. C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," arXiv preprint arXiv:1811.03378, 2018.
65. R. Nishihara and P. Moritz, Ray v1.0.1. [Online]. Available: https://docs.ray.io/en/ray-1.0.1/serve/faq.html.
66. K. Srinath, "Python–the fastest growing programming language," International Research Journal of Engineering and Technology, vol. 4, no. 12, pp. 354–357, 2017.
67. A. S. Saabith, M. Fareez, and T. Vinothraj, "Python current trend applications-an overview," International Journal of Advance Engineering and Research Development, vol. 6, no. 10, 2019.
68. S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," Information, vol. 11, no. 4, p. 193, 2020.
69. I. Staňcin and A. Jovi´c, "An overview and comparison of free python libraries for data mining and big data analysis," in 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2019, pp. 977–982.
70. G. Piatetsky, "Python leads the 11 top data science, "Machine Learning Platforms: Trends and Analysis, 2019.
71. F. Pedregosa et al., "Scikit-learn: Machine learning in python," the Journal of machine Learning research, vol. 12, pp. 2825–2830, 2011.
72. C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," Communications in Computational Physics, vol. 15, no. 2, pp. 285–329, 2014.