IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 23, 2021, accepted April 6, 2021, date of publication April 9, 2021, date of current version April 21, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3072231

FastPathology: An Open-Source Platform for Deep Learning-Based Research and Decision Support in Digital Pathology

ANDRÉ PEDERSEN[1,3], MARIT VALLA[1,3,4], ANNA M. BOFIN[1], JAVIER PÉREZ DE FRUTOS[2], INGERID REINERTSEN[2,5], AND ERIK SMISTAD[2,5]

[1]Department of Clinical and Molecular Medicine, Norwegian University of Science and Technology, 7491 Trondheim, Norway
[2]SINTEF Medical Technology, 7465 Trondheim, Norway
[3]Clinic of Surgery, St. Olavs Hospital, Trondheim University Hospital, 7030 Trondheim, Norway
[4]Department of Pathology, St. Olavs Hospital, Trondheim University Hospital, 7030 Trondheim, Norway
[5]Department of Circulation and Medical Imaging, Norwegian University of Science and Technology, 7491 Trondheim, Norway

Corresponding author: André Pedersen (andre.pedersen@ntnu.no)

This work was supported in part by The Liaison Committee for Education, Research and Innovation in Central Norway (Samarbeidsorganet), and in part by the Cancer Foundation, St. Olavs Hospital, Trondheim University Hospital (Kreftfondet).

ABSTRACT Deep convolutional neural networks (CNNs) are the current state-of-the-art for digital analysis of histopathological images. The large size of whole-slide microscopy images (WSIs) requires advanced memory handling to read, display and process these images. There are several open-source platforms for working with WSIs, but few support deployment of CNN models. These applications use third-party solutions for inference, making them less user-friendly and unsuitable for high-performance image analysis. To make deployment of CNNs user-friendly and feasible on low-end machines, we have developed a new platform, *FastPathology*, using the FAST framework and C++. It minimizes memory usage for reading and processing WSIs, deployment of CNN models, and real-time interactive visualization of results. Runtime experiments were conducted on four different use cases, using different architectures, inference engines, hardware configurations and operating systems. Memory usage for reading, visualizing, zooming and panning a WSI were measured, using FastPathology and three existing platforms. FastPathology performed similarly in terms of memory to the other C++-based application, while using considerably less than the two Java-based platforms. The choice of neural network model, inference engine, hardware and processors influenced runtime considerably. Thus, FastPathology includes all steps needed for efficient visualization and processing of WSIs in a single application, including inference of CNNs with real-time display of the results. Source code, binary releases, video demonstrations and test data can be found online on GitHub at https://github.com/SINTEFMedtek/FAST-Pathology/.

INDEX TERMS Deep learning, neural networks, high performance, digital pathology, decision support.

I. INTRODUCTION

Whole Slide microscopy Images (WSIs) used in digital pathology are often large, and images captured at $\times 400$ can have approximately $200k \times 100k$ color pixels resulting in an uncompressed size of $\sim$ 56 GB [1]. This exceeds the amount of Random-Access Memory (RAM) and Graphics Processing Unit (GPU) memory on most computer systems. Thus, special data handling is required to store, read, process and display these images.

The associate editor coordinating the review of this manuscript and approving it for publication was Asad Waqar Malik.

With the integration of digital pathology into clinical practice worldwide [2], [3], there is a need for tools that can assist clinicians in their daily practice. Deep learning, and particularly Convolutional Neural Networks (CNNs), currently represent the state-of-the-art in automated and semi-automated analysis. CNNs are a class of artificial neural networks that can learn spatial features in the input data and are thus widely used in a range of computer vision tasks, including radiology and digital pathology. In the analysis of WSIs, the use of CNNs has resulted in accuracies surpassing traditional image analysis techniques [4]–[6]. Still, deploying CNNs requires computer science expertise, making it difficult for clinicians

and non-engineers to implement these methods into clinical practice. Thus, there is a need for an easy-to-use software that can load, visualize and process large WSIs using CNNs.

There are several open-source softwares available for visualizing and performing traditional image analysis on WSIs such as QuPath [7] and Orbit [8]. Still, most of these do not support deployment of CNNs. Most developers working with CNNs train their models in Python using frameworks like TensorFlow [9] and Keras [10]. Thus, platforms intended for use in digital pathology should support deployment of these models. A solution to this may be to deploy models in Python directly, using the same libraries, as done in Orbit. Inference in Python is optimized, because the actual Inference Engines (IEs), such as TensorFlow, are usually written in C and C++, and use parallel processing and GPUs. However, the Python language itself is not optimized and is thus unfit for large scale, high performance software development. Most existing platforms use Java/Groovy as the main language. Despite boasting good multi-platform support and being a modern object-oriented language, the performance of Java compared to C and C++ is debated [11], [12]. It is possible to deploy TensorFlow-based models in Java, with libraries like DeepLearning4J [13], but its support for layers and network architectures is currently limited.

We argue that due to the high memory and computational demands of processing and visualizing WSIs, modern C++ together with GPU libraries such as OpenCL and OpenGL are better suited to create such a software. We therefore propose to use and extend the existing high-performance C++ framework FAST [14] to develop an open-source platform for reading, visualizing and processing WSIs using deep CNNs. FAST was introduced in 2015 as a framework for high performance medical image computing and visualization using multi-core Central Processing Units (CPUs) and GPUs [14]. In 2019 [15], it was extended with CNN inference capabilities using multiple inference engines such as TensorFlow, OpenVINO [16] and TensorRT [17]. In this article, we describe a novel application *FastPathology* based on FAST which consists of a Graphical User Interface (GUI) and open trained neural networks for analyzing digital pathology images. We also outline the components that have been added to FAST to enable processing and visualization of WSIs. Four different neural network inference cases, including patch-wise classification, low-resolution segmentation, high-resolution segmentation and object detection, are used to demonstrate the capabilities and computational performance of the platform. The application runs on both Windows and Ubuntu Linux Operating Systems (OSs) and is available online at https://github.com/SINTEFMedtek/FAST-Pathology/.

A. RELATED WORK

QuPath [7] is a popular software for visualizing, analyzing and annotating WSIs. It is a Java-based application that supports reading WSIs using open-source readers such as Bio-Formats [18] and OpenSlide [19]. QuPath can be applied directly using the GUI, but it also includes an integrated script editor for writing Groovy-based code for running more complex commands and algorithms. Its annotation tool supports multiple different, dynamic brushes, and it can be used for various structures at different magnification levels. Using QuPath, it is possible to create new classifiers directly in the software, e.g. using Support Vector Machines (SVMs) and Random Forests (RFs). Quite recently, attempts to support deployment of trained CNNs have been made through StarDist [20], using TensorFlow to deploy a deep learning-based model for cell nucleus instance segmentation. Currently, the user cannot deploy their own trained CNNs in QuPath. However, it is possible to import external predictions from disk and save them as annotations.

The software **ASAP** [21] supports visualization, annotation and analysis of WSIs. Unlike QuPath, ASAP is based on C++. ASAP can also be used in Python directly through a wrapper, which is suitable as most machine learning researchers develop and train their models in Python.

Orbit [8] is a recently released software. It includes processing and annotating tools similar to QuPath. However, it is possible to deploy and train CNNs directly in the software. Orbit is written in Java, but the deep learning-module is written in Python, and executed from Java. For computationally intensive tasks, such as training of CNNs, Orbit uses a Spark infrastructure, which makes it possible to relax the footprint on the local hardware.

Due to the large size of WSIs, utilizing algorithms on these has a high computational cost. **Cytomine** [22] is a platform that solves this by running analyses through a web interface using a cloud-based service. It has similar options for visualization, annotation and analysis to QuPath. Its core solutions are open-source, however more advanced modules are not free-to-use. It also lacks options for CNN inference.

B. CONTRIBUTIONS

Following our previous work on neural network inference in FAST [15], which demonstrated basic WSI processing functionality, the following are the main contributions of this article:

- A new, free-to-use and open-source application called *FastPathology* for deep-learning based digital pathology, consisting of a user-friendly GUI and a collection of open trained neural networks.
- A new system for creating, reading and writing arbitrary large images and segmentations with low latency and memory footprint, using a tiled image pyramid approach with memory mapping implemented in FAST.
- A new GPU-based renderer for visualization of high-resolution neural network predictions as transparent colored overlays on top of WSIs interactively in real-time.
- A text pipeline method in FAST, enabling users to create, modify and use complex processing and visualization pipelines without programming.

- GPU-based methods for fast detection and rendering of bounding boxes detected on WSIs in FAST.
- Improved storage capabilities of predictions in FAST, making neural network deployment generalizable and scalable as demonstrated on four WSI specific use cases.
- Quantitative memory usage comparison of four open-source digital pathology applications.

II. METHODS

In the existing FAST framework, several components needed to be created to read, visualize and process WSIs. This section first describes how these components were designed to handle WSIs on a computer system with limited memory and computational resources. Then, the FastPathology application itself is described, including how it was designed to enable users without programming experience to apply deep learning models on WSIs.

A. READING WHOLE SLIDE IMAGES

WSIs are usually stored in proprietary formats from various scanner vendors. The open-source, C-based library OpenSlide [19] can read most of these proprietary formats. Since these images are very large, they are usually stored as tiled image pyramids. OpenSlide was added to FAST to enable reading of these files, thereby accessing the raw color pixel data. OpenSlide uses the virtual memory mechanisms of the operating systems. Thus, by streaming data on demand from disk to RAM, it is possible to open and read large files without exhausting the RAM system memory.

B. CREATING ARBITRARY LARGE IMAGES

When performing image analysis tasks such as segmentation on high-resolution image planes of WSIs, it is necessary to create, write and read large images while performing segmentation using a sliding window approach. To facilitate this, a tiled image pyramid data object was added to FAST, enabling the creation of images of arbitrary sizes. Given an image size of $M \times N$, FAST creates L levels where each level has the size $\frac{M}{2^l} \times \frac{N}{2^l}$ with l ranging from 0 to $L-1$. Levels smaller than 4096×4096 are not created. This limit was chosen to keep the number of levels low, as it affects storage usage, while still having a low-resolution image level which can fill the entire screen without looking blurred. Storing all levels in memory of a $\times 400$ WSI, would require an extremely large amount of memory. Thus the operating system's native file-based memory mapping mechanisms are used, which on Linux is the 64 bit mmap function and on Windows the file mapping mechanism. These file mapping mechanisms essentially create a large file on disk, and virtually map it to RAM, thus streaming data back and forth. Reading and writing data in this manner is slower compared to using the RAM only. Furthermore, the speed is affected by disk speed, and it requires additional disk space. Levels that use less than a threshold of 512 MB are stored in RAM instead without memory mapping. This was done to increase performance of loading the first levels at the cost of RAM usage up the specific threshold. The threshold corresponds to storing a level of $\sim 11.5k \times 11.5k$ pixels with 4 bytes per pixel.

C. RENDERING A WSI WITH OVERLAYS

High performance interactive image rendering with multiple overlays, colors and opacity usually requires a GPU implementation. Since GPUs also have a very limited memory size, WSIs will not fit into the GPUs memory. There is no native virtual memory system on GPUs, thus a virtual memory system for WSIs was implemented for GPUs in FAST, using OpenGL. From the image pyramid, only the required tiles at the required resolution in the image pyramid are transferred to the GPU memory as textures. To further reduce GPU memory usage, the tiles are stored using OpenGL's built-in texture compression algorithms (GL_COMPRESSED_RGBA). The tiles and resolution required at a given time are automatically determined based on the current position and amount of zoom of the current view of the image. Reading tiles from disk and streaming them to the GPU is time consuming. Therefore, the tile textures are processed and uploaded to the GPU in a separate thread. When the tile texture is done it is stored in a first-in-first-out (FIFO) cache. The user can manually specify the maximum cache size in bytes. Every time a tile is used, it is placed at the back of the cache. When the cache exceeds its limit, tiles are removed from the front of the cache and their textures deleted. This is done to limit the amount of GPU memory used for rendering. Before a given tile is ready to be rendered, the next-best resolution tiles already cached are displayed. The lowest resolution of the image pyramid is always present in GPU memory. Thus, the WSI and any overlays will always be displayed, even when higher resolution tiles are being loaded. The prediction overlays are continuously updated while the neural network is processing. For low-resolution predictions, this is done by updating the visualization on the GPU in real-time. When rendering high-resolution predictions stored as tiled image pyramids (e.g. whole WSI segmentation (use case 3)), a more advanced process is needed. Every time the high-resolution prediction is updated, e.g. a new patch has been processed, the patch prediction is written to level 0 of the image pyramid and every pixel changed is propagated upwards in the image pyramid. Every modified tile is marked as *dirty* in the GPU-based renderer, thereby triggering the system to update the tile's texture with the latest predictions.

The user can easily pan and zoom to visualize all parts of a WSI with low latency and a bounded GPU memory usage. In FAST, multiple images and objects can be displayed simultaneously with an arbitrary number of overlays. This enables high and low-resolution segmentations, patch-wise classifications, and bounding boxes to be displayed on top of a WSI with different colors and opacity levels. These can also be changed in real-time while processing.

Figure 1 shows an example of how the predictions can be visualized at different resolutions as overlays on top of

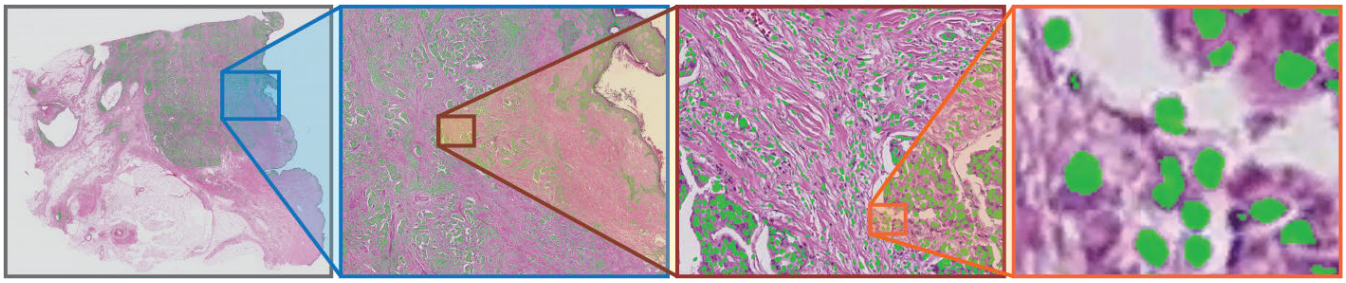

FIGURE 1. Illustration of how predictions, in this case segmented cell nuclei (green), can be visualized on top of a WSI in the viewer on different magnification levels.

the WSI. This illustrates the large size of these WSIs and why a tiled image pyramid data structure is required to visualize and process these images.

D. TISSUE SEGMENTATION

Since WSIs are so large, applying a sliding window method across the image might be time consuming, especially when using CNNs. Thus, removing regions other than the tissue sample would be an advantage. In FAST, a simple tissue detector was implemented which segments the WSI by thresholding the RGB image color space, using a predefined threshold. The image level with lowest resolution is segmented based on the Euclidean distance between a specific RGB triplet and the color white. Morphological closing is then performed to bias sensitivity in tissue detection. The default parameters, such as the threshold value, were empirically determined and tuned on WSIs from a series of breast cancer tissue samples. The tissue segmentation method was implemented in OpenCL to run in parallel on the GPU or on the multi-core CPU.

E. NEURAL NETWORK PROCESSING

Inference of neural networks is done through FAST by loading a trained model stored on disk as described in [15]. FAST comes with multiple inference engines including: 1) Intel's OpenVINO which can run on Intel CPUs as well as their integrated GPUs, 2) Google's TensorFlow which can run on NVIDIA GPUs with the CUDA and cuDNN framework, or directly on CPUs, and 3) NVIDIA's TensorRT which can run on NVIDIA GPUs using CUDA and cuDNN. FAST will automatically determine which inference engines can run on the current system depending on whether CUDA, cuDNN or TensorRT are installed or not.

In many image analysis solutions the WSI is tiled into small patches of a given size and magnification level. A method is then applied to each patch independently, and the results are stitched together to form the WSI's analysis result. FAST uses a *patch generator* to tile a WSI into patches in a separate thread on the CPU. Thus, a neural network can simultaneously process patches while new patches are being generated. Due to the parallel nature of GPUs, it can be beneficial to perform neural network inference on batches of patches, which can be done in FAST using the *patch to batch generator*. Finally, the *patch stitcher* in FAST takes the stream of patch-wise predictions to form a final result image or tensor which can be visualized or further analyzed. For methods which generate objects such as bounding boxes, an *accumulator* is used instead which simply concatenates the objects into a list. Since computations and visualizations are run in separate threads in FAST, the predictions can be visualized on top of the WSI, while the patches are being processed, simultaneously.

It is also possible to run different analyses on different threads in FAST. However, as amount of memory and threads are limited, running multiple processes simultaneously might affect the overall runtime performance.

Results are stored differently depending on whether one is performing patch-wise classification, object detection or segmentation. For patch-wise classification, predictions are visualized as small rectangles with different colors for different classes and varying opacity dependent on classification confidence level. For object detection, predictions are visualized as bounding boxes, where the color of the box indicates the predicted class. For semantic segmentation, pixels are classified, and given a color and opacity depending on the predicted class and confidence level.

To enable introduction of new models and generalizing to different multi-input/output network architectures, each model assumes that it has a corresponding *model description text-file*. This file includes information on how the models should be handled. For instance, for some inference engines, the input size must be set by the user, as it is not interpretable directly from the model.

F. GRAPHICAL USER INTERFACE

In order to use the WSI functionality in FAST without programming, a GUI is required. The GUI of FastPathology was implemented using Qt 5 [23]. The GUI was split into two windows. On the right side there is a large OpenGL window for visualizing WSIs and analysis results from CNN predictions. On the left, the user can find a dynamic taskbar with five sections for handling WSIs.

1) **Import:** Options to create or load existing projects and reading WSIs.

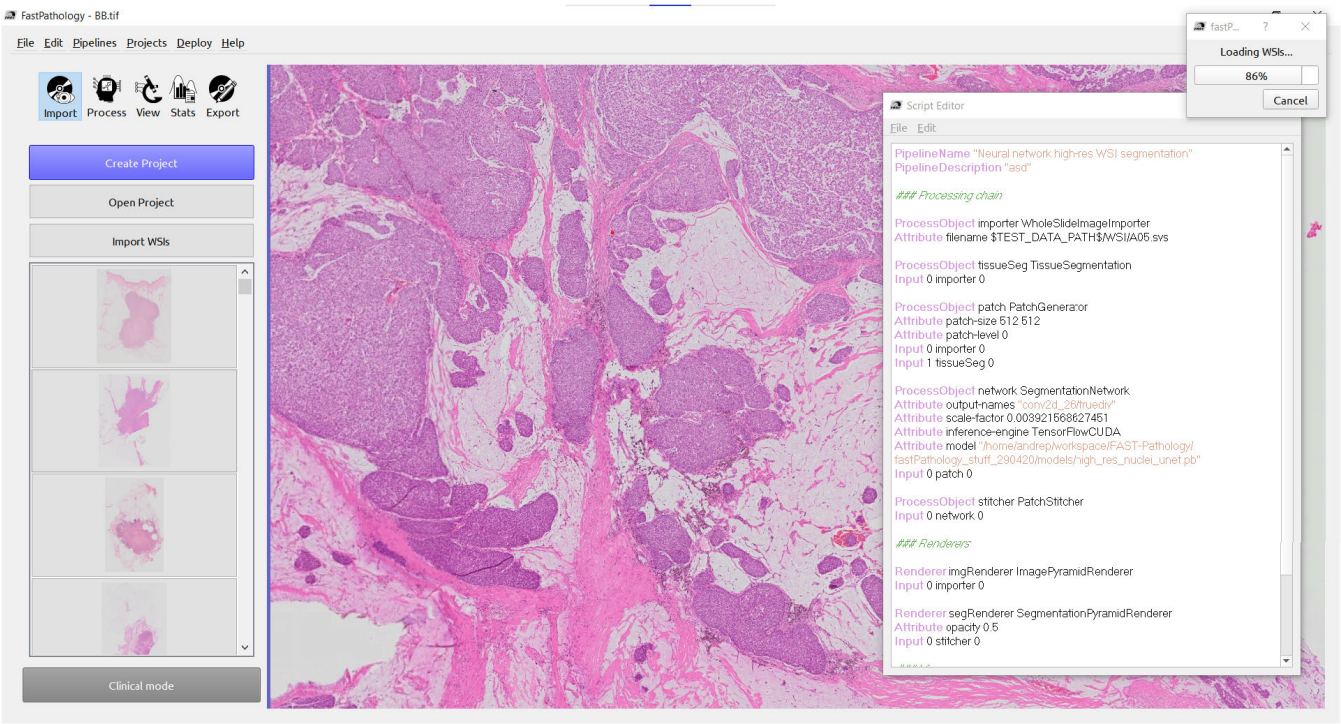

FIGURE 2. An example of FastPathology's GUI showing some basic functionalities. The task bar can be seen on the left side. The right side contains a OpenGL window rendering a WSI. On top of the window is a progress bar and a script editor containing a text pipeline.

2) **Process:** Selection of available processing methods, e.g. tissue segmentation or inference with CNN.
3) **View:** Viewer for selecting results to visualize, e.g. tumor segmentation, patch-wise histological prediction.
4) **Stats:** Extract statistics from results, e.g. histogram of histological grade predictions, final overall WSI-level prediction.
5) **Export:** Exporting results in appropriate formats, e.g..png or.mhd/.raw for segmentations and heatmaps, or .csv for inference results.

An example of the GUI can be seen in Figure 2. The View, Stats and Export widgets are dynamically updated depending on which results are available. In the View widget one can also change the opacity of the result or the class directly, and the color. Results can be removed and inference can be halted. Figure 3 shows how the user can interact with the GUI and how the different components relate.

G. TEXT PIPELINES

FAST implements *text pipelines*, a txt-file containing information regarding which components to use in a pipeline. These pipelines are deployable directly within the software. It is also possible to load external pipelines, or to create or edit pipelines using the built-in script-editor, as seen in Figure 2. To make the editor more user-friendly, text highlighting was added. This produces different colors for FAST objects and corresponding attributes, e.g. patch generator and magnification level. Using FastPathology, it is also possible to modify other text-files, such as the model description text-file.

H. ADVANCED MODE

An advanced mode was added to enable users to change and tune hyperparameters of algorithms and models. For tissue segmentation, the threshold and kernel size for the morphological operators can be set in the GUI. A dynamic preview of the segmentation is then updated in real time, to give the user feedback about the selected parameters.

I. PROJECTS

It may be convenient to run the same analysis on multiple WSIs. Therefore, a *Project* can be created, and several WSIs can be added to the project. By selecting a pipeline, and choosing *run-for-project*, the pipeline is run sequentially on all WSIs in the project. Results are stored within the project in a separate folder. This makes it possible to load the project including the results, and export the results to other platforms, e.g. QuPath.

J. STORING RESULTS

Storing results from different image analysis is an important part of a WSI analysis platform. Currently, it is possible to store the tissue segmentation and predictions on disk using the metaimage (.mhd/.raw) format in FAST. Tensors from neural networks are stored using the HDF5 format.

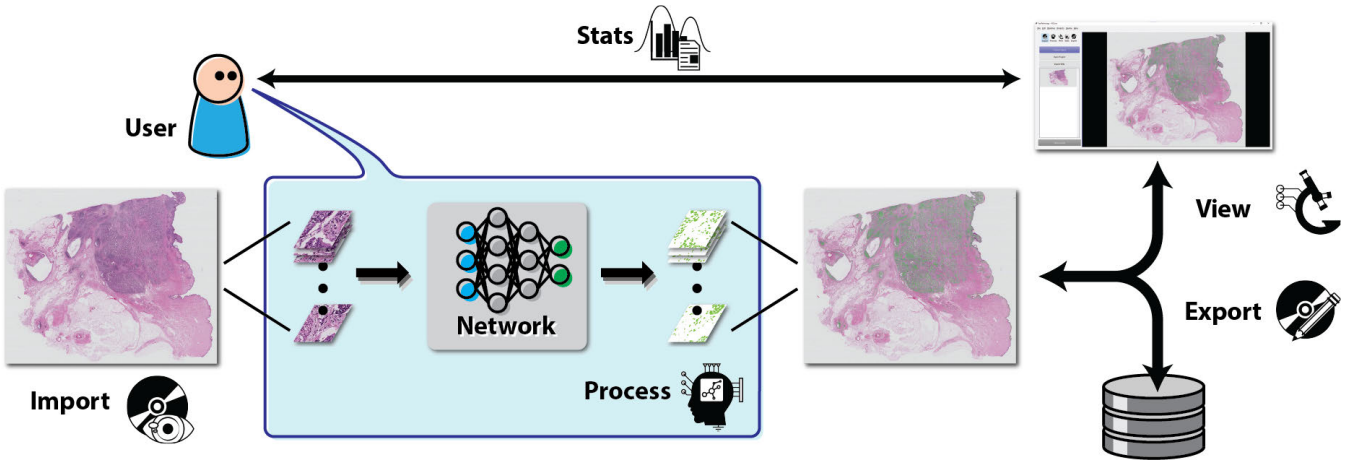

FIGURE 3. Illustration of the user workflow for analyzing WSIs in FastPathology. It also shows how each component in the GUI are related, and how the user can run a pipeline (Process) and get feedback from the neural network, either from the OpenGL window (View) or from the statistics summary (Stats). WSIs can be added through the Import widget and results are stored on disk using the Export widget.

K. INFERENCE USE CASES

Four different neural network inference cases were selected to demonstrate the capabilities and performance of the application. All models were implemented and trained using TensorFlow 1.13. For use cases 1, 3 and 4, the tissue segmentation method was used to limit the neural network processing to tiles with tissue only. All models were trained as a proof-of-concept for the platform, not to achieve the highest possible accuracy.

1) USE CASE 1 - PATCH-WISE CLASSIFICATION

This use case focuses on patch-wise classification of WSIs. The image was tiled into non-overlapping tiles of size $512 \times 512 \times 3$ at $\times 200$ magnification level, and RGB intensities normalized to the range [0, 1]. The network used was a CNN with the MobileNetV2 [24] encoder pretrained on the ImageNet dataset [25]. The classifier part contained a global average max pooling layer and two dense layers with 64 and 4 neurons respectively. Between the dense layers, batch normalization, ReLU and dropout with a rate of 0.5 were used. In the last layer a softmax activation function was used to obtain the output probability prediction for each class. The model has $\sim$ 2.31M parameters. It was trained on the Grand Challenge on Breast Cancer Histology Images (BACH) dataset [26]. The model classifies tissue into four classes: normal tissue, benign lesion, in situ, and invasive carcinoma. A patch stitcher is used to create a single heatmap of all the classified patches. The heatmap is visualized on top of the WSI with a different color for each class. The opacity reflects the confidence score of the class as shown in Figure 4a).

2) USE CASE 2 - LOW-RESOLUTION SEGMENTATION

This task focuses on semantic segmentation of WSIs by segmenting pixels of the entire WSI using the pyramid level with the lowest resolution. Thus, this use case does not process patches, but the entire image. The network uses a fully convolutional encoder-decoder scheme, based on the U-Net architecture [27]. From images of size $1024 \times 1024 \times 3$, the network classifies each pixel as tumor or background. All the convolutional layers in the model are followed by batch normalization, ReLU and a spatial dropout of 0.1. However, in the output layer, the softmax activation function was used. The total number of parameters was $\sim$ 11.56M. The dataset used is a subset of a series of breast cancer cases curated by the Breast Cancer Subtypes Project [28]. The subset comprises Hematoxylin-Eosin (H&E)-stained full-face tissue sections (4μ thick) from breast cancer tumors. WSIs were captured at $\times 400$ magnification. The result is visualized on top of the WSI with each class having a different color. The opacity reflects the confidence score of the class. Figure 4b) shows the results of this use case, where the segmented tumor region is shown in transparent red, whereas the background class is completely transparent.

3) USE CASE 3 - HIGH-RESOLUTION SEGMENTATION

We used the same U-Net-architecture as in use case 2, to perform segmentation on independent patches. The image was tiled as in use case 1. Tiles of size $256 \times 256 \times 3$ were used. Patches from varying image planes were extracted (around $\times 200$), but higher resolution tiles were preferred. The PanNuke dataset [29], [30] was used to train the model. PanNuke is a multi-organ pan-cancer dataset for nuclear segmentation and classification. It contains 19 different tissue types and five different classes of nuclei: inflammatory cell, connective tissue, neoplastic, epithelial, and dead (apoptic or necrotic) nuclei. We only trained the model to perform nuclear segmentation, regardless of class. The total number of parameters was $\sim$ 7.87M. The segmentation of each patch was stitched together to form a single, large segmentation image. This image has the same size as the image pyramid level it is processing, and the result is formed into a new image segmentation pyramid as described in section II-B.

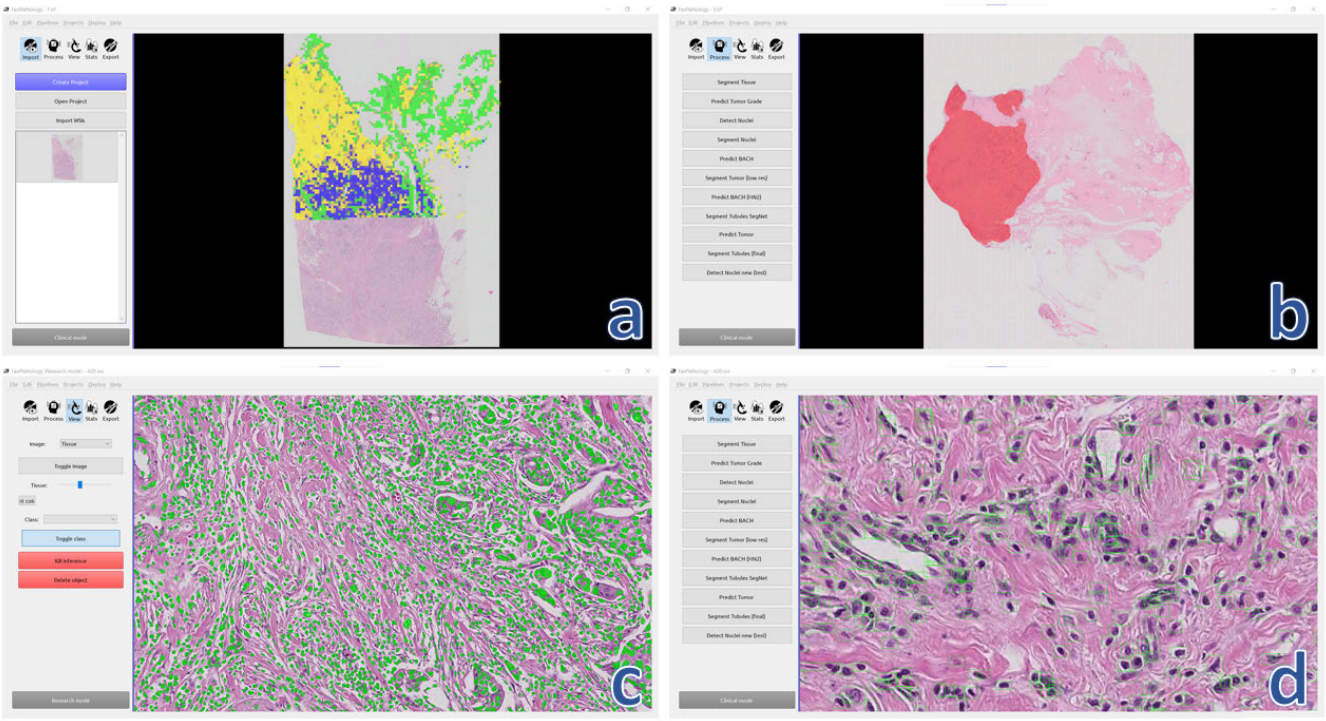

FIGURE 4. Illustrating the resulting predictions of each use case on top of a WSI. a) patch-wise classification of tissue, b) low-resolution segmentation of breast cancer tumor, c) high-resolution segmentation of cell nuclei, and d) object detection of cell nuclei.

The result is visualized on top of the WSI with each class having a different color (see Figure 4c).

4) USE CASE 4 - OBJECT DETECTION AND CLASSIFICATION

We used the same tiling strategy as for use case 3, with the same image planes and input size. However, in this case we performed object detection using the Tiny-YOLOv3-architecture [31]. Implementation and training of Tiny-YOLOv3 was inspired by the specified GitHub repository.[1] The model was pretrained on the COCO dataset [32], and fine-tuned on the PanNuke dataset. Transfer learning was performed due to the challenge of training an object detector from scratch, where the greatest observed benefit was in convergence speed. Bounding box coordinates with corresponding confidence and predicted class for all predicted candidates were made. The total number of parameters was $\sim$ 8.67M. Non-maximum suppression was performed to handle overlapping bounding boxes. From all patches, these were then accumulated into one large bounding box set, visualized as colored lines with OpenGL, where the color indicates the predicted class (see Figure 4d).

III. EXPERIMENTS

A. RUNTIME

To assess speed, we performed runtime experiments using the four use cases. The experiments were run on a single Dell desktop with Ubuntu 18.04 64 bit operating system, with 32 GB of RAM, an Intel i7-9800X CPU and two NVIDIA GPUs, GeForce RTX 2070 and Quadro P5000. We measured runtimes using the four inference engines: TensorFlow CPU, TensorFlow GPU (v1.14), OpenVINO CPU (v2020.3) and TensorRT (v7.0.0.11). TensorRT was only used in use cases 1 and 4, where an UFF-model was available. All U-Net models contained spatial dropout and upsampling layers that were not supported by TensorRT, and thus could not be converted. For each inference engine, a warmup run was done before 10 consecutive runs were performed. Runtimes for each module in a pipeline were reported. The warmup was done to avoid measurements being influenced by previous runs. The experiments were run sequentially.

From these experiments, the population mean ($\bar{X}$) and standard error of the mean ($S_{\bar{X}}$) were calculated. Multiple Shapiro-Wilk tests [33] were conducted to state whether the data were normal. The Benjamini-Hochberg false discovery rate method [34] was used to correct for multiple testing. For all hypothesis tests, a significance level of 5 % was used. Only six out of 32 variables had small deviations from the normal distribution, thus a normal distribution was assumed. The mean and 95%-confidence intervals were reported. In addition, multiple pairwise tests were performed using Tukey's range test [35] to evaluate whether there were a significant difference between any of the total runtimes (see supplementary material for the p-values).

[1] https://github.com/qqwweee/keras-yolo3

All experiments were done on the A05.svs ×200 WSI from the BACH dataset. Measurements were in milliseconds, if not stated otherwise. To simplify the measurements, rendering was excluded in all runtime measurements. The OpenGL rendering runtime is so small it can be regarded as negligible. The real bottleneck is inference speed and patch generation. To make all experiments directly comparable, a batch size of one was used during inference.

For all runtime measurements we reported the time used for each component (patch generator, neural network input and output processing, neural network inference, and patch stitcher), and the combined time in a FAST pipeline. Neural network input processing includes resizing the images if necessary and intensity normalization (0-255 → 0-1).

B. MEMORY

We monitored memory usage on selected tasks and compared them to the QuPath (v0.2.3), ASAP (v1.9) and Orbit (v3.64) platforms. All experiments were run on the same Dell desktop as used in Section III-A (using the RTX 2070 GPU). The WSI used was the TE-014.svs from the Tumor Proliferation Assessment Challenge 2016 [36], since it is a large, openly available ×400 WSI.

In this experiment, memory usage was measured after starting the program, after opening the WSI, and after zooming and panning the view for 2.5 minutes. Both RAM and GPU memory usage was measured. To make the comparison fair, we attempted to make similar movements and zoom for all platforms.

Orbit initializes the WSI from a zoomed region, in contrast to the three other which initializes from a low-resolution overview image. In order to achieve the same overview field of view for all, it was necessary to zoom out initially when using Orbit. This, however spiked the RAM usage for Orbit. Thus, to make comparison fair, we only measured memory usage after the initial image was displayed when opening a WSI for all applications.

The physical memory usage was monitored using the interactive process viewer *htop* on Linux. Due to this, if a process used ≥ 10 GB of RAM, htop would report it in the format 0.010 TB, which meant that we had lower resolution on these measurements. The graphical memory usage was monitored using the NVIDIA System Management Interface (*nvidia-smi*).

C. MODEL AND HARDWARE CHOICE

To further assess how different neural network architectures could affect inference speed on a specific use case, we ran use case 1 with a more demanding InceptionV3 model [15]. This model is available in the FAST test data.[2] The model should have a classifier part identical to the one used for our MobileNetV2 model.

The same use case was also run on a low-end HP laptop with Windows 10 64 bit operating system, 16 GB of RAM, an Intel i7-7600 CPU, and Intel HD Graphics 620 integrated GPU, to show how runtimes could differ between low- and high-end machines.

To compare difference in runtime between operating systems, we also run the same experiments using a high-end Razer laptop with Windows 10 64 bit operating system, 32 GB of RAM, an Intel i7-10750H CPU, an Intel UHD graphics integrated GPU, and NVIDIA GeForce RTX 2070 Max-Q GPU. To our understanding, the performance of both the CPU and GPU should be comparable to that of the Dell desktop computer used in the experiments. During experiments with both Windows laptops, the machines were constantly being charged and real-time anti-malware protection was turned off. For all machines, all experiments were performed using a Solid State Drive (SSD).

IV. RESULTS

A. RUNTIME

Comparing the choice of inference engine, Tables 2 - 5 show that inference with TensorFlow CPU was the slowest alternative, for each respective use case, especially using TensorFlow CPU (see supplementary material for the p-values). Inference with GPU was the fastest, with TensorRT slightly faster than TensorFlow CUDA. However, no significant difference was found between TensorFlow CUDA and TensorRT in any of the runtime experiments. The OpenVINO CPU IE had comparable inference speed with the GPU alternatives, even surpassing TensorFlow CUDA on the low-resolution segmentation task. However, no significant difference was observed. Thus, there was no benefit of using the GPU for low-resolution segmentation. We also ran inference with two different GPUs using TensorRT, and found negligible difference in terms of inference speed between the two hardwares. Also, more complex tasks such as object detection and high-resolution segmentation resulted in slower runtimes than patch-wise classification and low-resolution segmentation, across all inference engines.

B. MEMORY

With regards to memory, there was a strong difference between the C++ and the Java-based applications (see Table 1). Both C++-based platforms used considerably less memory across all experiments.

Using nvidia-smi we observed that FastPathology was the only platform that ran both computation and graphics on the GPU (C+G). FAST uses OpenCL for computations and OpenGL for rendering. The two Java-based softwares (QuPath and Orbit) only ran graphics on GPU, either using DirectX or another non-OpenGL form of rendering. ASAP and Orbit did not use any GPU, whereas QuPath used a negligible amount. Hence, FastPathology was the only platform capable of exploiting the advantage of having a GPU available for both computations and rendering. It was observed that both C++ applications (FastPathology and ASAP) opened their WSIs

[2] https://github.com/smistad/FAST/wiki/Test-data

TABLE 1. Memory measurements of reading, panning and zooming the view of a ×400 WSI. All memory usage values are in MB.

Memory usage	FastPathology		QuPath		Orbit		ASAP	
	RAM	GPU	RAM	GPU	RAM	GPU	RAM	GPU
Application startup	205	101	497	86	373	0	84	0
Opening WSI	268	111	989	88	817	0	173	0
Zooming and panning	1,544	1,203	∼11,000	89	9,903	0	1,185	0

TABLE 2. Runtime measurements of use case 1 - Patch-wise classification, using the MobileNetV2 encoder performed on the Ubuntu desktop. Each row corresponds to an experimental setup. Each cell displays the average runtime and 95 % confidence interval limits for 10 successive runs.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-9800X	29.7 ± 0.0	1.4 ± 0.0	16.7 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	145.0 ± 0.2
TensorFlow CPU	Intel i7-9800X	34.0 ± 0.0	1.1 ± 0.0	35.6 ± 0.4	0.0 ± 0.0	0.0 ± 0.0	176.4 ± 2.0
TensorFlow CUDA	Quadro P5000	21.3 ± 0.4	1.5 ± 0.0	9.3 ± 0.3	0.0 ± 0.0	0.0 ± 0.0	103.8 ± 2.2
TensorRT	GeForce RTX 2070	20.2 ± 0.1	1.3 ± 0.0	1.2 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	98.9 ± 0.5
	Quadro P5000	20.4 ± 0.1	1.3 ± 0.0	1.3 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	99.8 ± 0.6

TABLE 3. Runtime measurements of use case 2 - low-resolution semantic segmentation.

Inference engine	Processor	Runtime (ms)				
		Image read	NN input	NN inference	NN output	Total (s)
OpenVINO CPU	Intel i7-9800X	9.4 ± 3.7	5.3 ± 0.1	149.3 ± 3.1	4.6 ± 5.7	0.17 ± 0.01
TensorFlow CPU	Intel i7-9800X	8.5 ± 0.7	3.1 ± 0.1	1101.9 ± 11.5	2.7 ± 0.1	1.12 ± 0.01
TensorFlow CUDA	Quadro P5000	10.7 ± 1.7	3.3 ± 0.2	998.9 ± 12.8	7.2 ± 1.3	1.0 ± 0.0

TABLE 4. Runtime measurements of use case 3 - high-resolution semantic segmentation.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-9800X	37.6 ± 0.1	0.8 ± 0.0	16.2 ± 0.0	0.2 ± 0.0	66.3 ± 0.1	400.7 ± 0.7
TensorFlow CPU	Intel i7-9800X	37.3 ± 0.2	1.0 ± 0.0	38.5 ± 0.2	0.3 ± 0.0	73.2 ± 0.1	542.3 ± 1.2
TensorFlow CUDA	Quadro P5000	29.9 ± 0.5	0.8 ± 0.0	5.3 ± 0.0	0.3 ± 0.0	76.1 ± 0.3	396.2 ± 1.6

instantly (< 1 s), whereas both Java-based softwares (QuPath and Orbit) were slower (3-4 s).

C. MODEL AND HARDWARE CHOICE

Tables 2 and 6 show runtime measurements on use case 1 using two different networks, MobileNetV2 and InceptionV3. Due to the increase in complexity, we observed that inference using CUDA was faster than using all CPU alternatives, in use case 1. This example showed that having a GPU available for inference can greatly speed up runtime, especially when models become more complex. A similar conclusion can also be drawn from Table 5 where a complex U-Net architecture was used, in contrast to using a lightweight Tiny-YOLOv3 architecture as seen in Table 4.

Tables 7 and 8 show inference using the low-end laptop. There was a significant increase in runtime for all inference engines. The low-end laptop had an integrated GPU and thus we could run inference using OpenVINO GPU.

TABLE 5. Runtime measurements of use case 4 - object detection and classification.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-9800X	9.9 ± 0.0	0.9 ± 0.0	6.7 ± 0.1	0.1 ± 0.0	0.0 ± 0.0	193.5 ± 0.5
TensorFlow CPU	Intel i7-9800X	10.6 ± 0.2	1.2 ± 0.1	14.3 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	295.0 ± 3.0
TensorFlow CUDA	Quadro P5000	6.6 ± 0.0	1.1 ± 0.0	3.4 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	129.7 ± 0.4
TensorRT	Quadro P5000	6.6 ± 0.0	1.1 ± 0.0	1.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	129.4 ± 0.3

TABLE 6. Runtime measurements of patch-wise classification (use case 1), using the InceptionV3 encoder performed on the Ubuntu desktop.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-9800X	28.4 ± 0.1	1.2 ± 0.0	49.9 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	245.4 ± 0.3
TensorFlow CPU	Intel i7-9800X	34.9 ± 0.0	1.3 ± 0.0	53.5 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	263.0 ± 0.2
TensorFlow CUDA	Quadro P5000	21.2 ± 0.1	1.3 ± 0.0	23.3 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	118.8 ± 0.4

TABLE 7. Runtime measurements of patch-wise classification (use case 1), using the MobileNetV2 encoder performed on the low-end windows laptop.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-7600U	51.3 ± 2.3	3.6 ± 0.1	51.8 ± 2.4	0.0 ± 0.0	0.0 ± 0.0	268.4 ± 12.2
OpenVINO GPU	Intel HD Graphics 620	63.9 ± 0.9	4.3 ± 0.0	28.6 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	314.0 ± 4.1
TensorFlow CPU	Intel i7-7600U	104.9 ± 3.9	4.1 ± 0.1	218.2 ± 2.1	0.0 ± 0.0	0.0 ± 0.0	1065.4 ± 10.2

TABLE 8. Runtime measurements of patch-wise classification (use case 1), using the InceptionV3 encoder performed on the low-end windows laptop.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-7600U	48.6 ± 0.3	3.6 ± 0.0	299.2 ± 1.5	0.0 ± 0.0	0.0 ± 0.0	1449.4 ± 7.4
OpenVINO GPU	Intel HD Graphics 620	159.6 ± 0.4	5.7 ± 0.0	153.2 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	788.6 ± 2.0
TensorFlow CPU	Intel i7-7600U	100.62 ± 0.9	4.6 ± 0.0	448.4 ± 1.8	0.0 ± 0.0	0.0 ± 0.0	2167.3 ± 8.9

This alternative is only better when more demanding models are used. Here, a much larger difference in runtime can be seen between the two CPU alternatives, TensorFlow CPU and OpenVINO CPU. OpenVINO was superior in terms of runtime.

Tables 9 and 10 show runtime measurements of the same use case with both encoders using the high-end Windows laptop. In this case we achieved runtime performance similar to the performance using the Ubuntu desktop. We found no significant difference using TensorRT between the two high-end machines, and TensorRT on Windows and TensorFlow CUDA on Ubuntu. For CPU there was a significant drop in performance for all use cases and encoders.

V. DISCUSSION

In this paper, we have presented a new platform, FastPathology, for visualization and analysis of WSIs. We have described the components developed to achieve this high-performance and easy-to-use platform. The software was evaluated in terms of memory usage, inference speed, and model and OS compatibility (see supplementary material for the p-values). A variety of deep learning use cases, model architectures, inference engines and processors were used.

TABLE 9. Runtime measurements of patch-wise classification (use case 1), using the MobileNetV2 encoder performed on the high-end windows laptop.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-10750H	31.6 ± 0.4	2.2 ± 0.0	22.5 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	155.3 ± 1.8
OpenVINO GPU	Intel UHD graphics	37.4 ± 0.2	2.5 ± 0.0	28.3 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	184.0 ± 0.8
TensorFlow CPU	Intel i7-10750H	48.0 ± 0.1	2.4 ± 0.0	79.9 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	395.1 ± 1.2
TensorRT	RTX 2070 Max-Q	21.8 ± 0.1	2.2 ± 0.0	5.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	108.5 ± 0.4

TABLE 10. Runtime measurements of patch-wise classification (use case 1), using the InceptionV3 encoder performed on the high-end windows laptop.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	NN patch stitcher	Total (s)
OpenVINO CPU	Intel i7-10750H	32.8 ± 0.7	2.4 ± 0.0	118.2 ± 0.3	0.0 ± 0.0	0.0 ± 0.0	578.6 ± 1.4
OpenVINO GPU	Intel UHD graphics	33.7 ± 0.1	2.8 ± 0.0	111.3 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	547.3 ± 0.3
TensorFlow CPU	Intel i7-10750H	47.2 ± 0.2	2.5 ± 0.0	165.7 ± 0.6	0.0 ± 0.0	0.0 ± 0.0	805.6 ± 3.0
TensorRT	RTX 2070 Max-Q	22.2 ± 0.1	2.1 ± 0.0	11.2 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	110.6 ± 0.5

A. MEMORY USAGE AND RUNTIME

In the memory experiments, FastPathology performed similarly to another C++-based software (ASAP), whereas both Java-based alternatives (QuPath and Orbit) were more memory expensive, using a large amount of memory while zooming. We have presented a runtime benchmark. Among the CPU alternatives, OpenVINO CPU performed the best. Inference on GPU was the fastest, but no significant difference was found when comparing TensorFlow CUDA and TensorRT. A small degradation in runtime was observed when using Windows compared to Ubuntu, but there was no significant difference using GPU. Runtimes on the low-end machine were slower, especially for more demanding models, but if an integrated GPU, such as Intel HD Graphics, is available, inference can be improved using OpenVINO GPU.

In use case 2, OpenVINO outperformed TensorFlow, even with GPU. This is due to TensorFlow having a large initialization overhead, of almost one second. This penalty is lesser in the other use cases, because the network is then run on a large number of patches. In use case 1, using TensorRT, we achieved similar runtime using two GPUs with quite varying memory size, 16 GB vs. 8 GB. This can be explained by both GPUs having similar computational power, and FAST/TensorRT only using the memory required to perform the task at hand. Hence, having a GPU with more memory does not necessarily improve runtime.

A slower runtime on the low-end machine can be explained by a lower frequency and number of cores (2 vs. 6) of the CPU. FAST takes advantage of all cores during inference and visualization. Thus, having a greater number of cores is beneficial, especially when running inference in parallel. Using the high-end machine on Windows, we also saw a small degradation in runtime using CPU. This may be explained by Windows having larger overhead compared to Ubuntu, or differences in hardware components that were not considered in this study, e.g. SSD. However, on GPU using TensorRT, there was a negligible difference between the two high-end machines. The small drop in performance might be due to the Windows machine having a Max-Q GPU design which is known to slightly limit the performance of the GPU, especially with regards to speed.

An interesting observation is that, for several of the use cases, using a GPU for inference does not give a large speedup over CPU inference in the total runtime of processing a WSI. This is mainly because the total runtime was dominated by the slow patch generation and not the neural network inference. The patch generation is slow because the WSI is not present in RAM, and has to be read through a virtual memory system.

In this study, we used two different encoders for use case 1, MobileNetV2 and InceptionV3. Both networks are baseline architectures, commonly used in digital pathology [26], [37], [38]. The latter is more computationally demanding, making inference slower compared to MobileNetV2. MobileNetV2 is useful for real-time deployment, while InceptionV3 is suitable for solving more complex tasks.

We get comparable runtime measurements as in a previous study [15]. This is expected as we use the same InceptionV3 model, applied to the same WSI, on the same use case, both using FAST. Small deviations are probably due to differences in hardware. These are the only measurements that are comparable with the prior study. All additional work is novel.

B. COMPARISON WITH OTHER PLATFORMS

QuPath is known to have a responsive, user-friendly viewer, with a seamless rendering of patches from different magnification levels. An optimized memory management or allocation of large amounts of data in memory is required to provide such a user experience. This could explain why QuPath used the largest amount of memory of all four tested solutions. FastPathology and ASAP provide a similar experience with a considerably smaller memory footprint. Rendering WSIs with Orbit did not work as swiftly, neither on Ubuntu nor Windows. To make FastPathology as efficient and fast as possible, we have in this article described new methods for performing full-resolution image analysis such as segmentation on WSIs, a GPU-based method for visualization of full-resolution segmentations, and a GPU-based method for detection and rendering of millions of bounding boxes on top of a WSI.

There is a wide range of platforms to choose from when working with WSIs. Solutions such as ASAP are made to be lightweight and responsive in order to support visualization and annotation of giga-resolution WSIs. Platforms such as QuPath enable deployment of built-in image analysis methods, either in Groovy, Python, or through ImageJ, as well as the option to implement the user's own methods. Orbit takes it further by making it possible for the user to train and deploy their own deep learning models in Python within the software. FastPathology can deploy CNNs in the same way as Orbit while maintaining comparable memory consumption to ASAP during visualization. It is also simple and user-friendly, requiring no code-interaction to deploy models.

Some models are more computationally demanding and thus naturally require greater memory. To some extent, memory usage can be adjusted through pipeline design, and by choice of model compression and inference engine. Depending on the hardware, FastPathology takes advantage of all available resources to produce a tailored experience when deploying models. Thus, pipeline designs such as batch inference can be done to further improve runtime performance. However, it is also possible to deploy models on low-end machines, even without a GPU. Machines that are using Intel CPUs, typically also include integrated graphics. In this case OpenVINO GPU could be used to improve runtime performance.

In FastPathology, the components used for reading, rendering and processing WSIs, and displaying predictions on top of the image, are made available through FAST. Since Python is one of the most popular languages for data scientists to develop neural network methods, FAST has been made available in Python as an official pip package,[3] and is currently available for Ubuntu (version 18 and 20) and Windows 10. This means that platforms that can use Python (e.g. Orbit and QuPath), could also use our solutions, for instance for enabling or improving deployment of CNNs.

[3]*pip install pyfast* - https://pypi.org/project/pyFAST/

C. STRENGTHS AND WEAKNESSES

The platform has been developed through close collaboration with the pathologists at St. Olavs Hospital, Trondheim, Norway, to ensure user-friendliness and clinical relevance. The memory usage of the platform for reading, visualizing and panning a $\times 400$ WSI has been compared to three existing softwares. As none of the existing platforms have published runtime benchmarks, the present study seeks to bridge the gap by providing a benchmark. The WSIs, models and source code for running these experiments have been made public to facilitate reproducibility and encourage others to run similar benchmarks.

The runtime measurements were only performed using three machines. Runtimes on new machines may vary, depending on hardware, as well as version and configuration of the OS. It is possible to further improve runtime by compressing models (e.g. using half precision), using a different patch size, or running models on lower magnification levels. However, this might degrade the final result. Such a study would require a more in-depth analysis in the trade-off between design and performance. As the models used were only trained to show proof of concept, this was considered outside the scope of this paper.

Regarding memory usage, experiments were only run *once* on *one* machine as these experiments were performed manually and were tedious to repeat. The measurements were performed by one person, and not the most likely end-user of the platform. Thus, in the future, a more in-depth study should be done to verify to what extent runtime and memory consumption differ depending on OS, hardware and user-interaction with the viewer. Including memory usage for the use cases would be interesting. However, FastPathology is the only platform to stream CNN-based predictions as overlays during inference, and thus a fair comparison cannot be made.

In all our experiments, we used a batch size of *one* when running inference. In theory, increasing the batch size should improve runtime. However, batch size is proportional to memory usage. Rendering the WSI and predictions as overlay also require GPU memory. Thus, large batch inference on low-end systems is not realistic. Nonetheless, as seen from all runtime experiments, patch generation is a bottleneck. There is also the dependency on hardware. If a less proficient CPU was available and a strong dedicated GPU, one would likely observe a much larger difference in CPU/GPU runtime, at least regarding inference runtime. Future studies will explore further improvements for the aforementioned trade-offs.

In order to remove areas of the WSI that only contain glass, Otsu's method is commonly used to automatically set the threshold [39]–[41]. However, we observed that when the tissue section was large, covering almost the entire slide, Otsu's method produced thresholds that separated tissue components, rather than background (glass). This phenomenon occurred because the threshold is based solely on the intensity histogram. Therefore, instead of using Otsu's method, we empirically tuned the threshold and set it manually during

deployment. Thus, if suboptimal tissue segmentation on a specific WSI is observed, the threshold can be manually adjusted by the user in FastPathology. Similar behaviour of Otsu's method has been observed in related work by Bandi *et al.* [40]. They proposed a more advanced approach using deep learning. Their proposed design could already be deployed if their trained model was available, as the architecture chosen is already supported in FastPathology.

FastPathology is continuously in development, and thus this paper only presents the first release. Future work includes support for more complex models, support for more WSI and neural network storage formats, and basic annotation and region of interest tools. As this is an open-source project, we encourage the community to contribute through GitHub.

VI. CONCLUSION

In this paper, we presented an open-source deep learning-based platform for digital pathology called FastPathology. It was implemented in C++ using the FAST framework, and was evaluated in terms of runtime on four use cases, and in terms of memory usage while viewing a ×400 WSI. FastPathology had comparable memory usage compared to another C++ platform, outperforming two Java-based platforms. In addition, FastPathology was the only platform that can perform neural network predictions and visualize the results as overlays in real-time, as well as having a user-friendly way of deploying external models, access to a variety of different inference engines, and utilize both CPU and GPU for rendering and processing. Source code, binary releases, video demonstrations and test data can be found online on GitHub at https://github.com/SINTEFMedtek/FAST-Pathology/.

REFERENCES

[1] P. Bandi *et al.*, "From detection of individual metastases to classification of lymph node status at the patient level: The camelyon17 challenge," *IEEE Trans. Med. Imag.*, vol. 38, no. 2, pp. 550–560, Feb. 2019.

[2] A. Baidoshvili, A. Bucur, J. van Leeuwen, J. van der Laak, P. Kluin, and P. J. van Diest, "Evaluating the benefits of digital pathology implementation: Time savings in laboratory logistics," *Histopathology*, vol. 73, no. 5, pp. 784–794, Nov. 2018.

[3] J. A. Retamero, J. Aneiros-Fernandez, and R. G. del Moral, "Complete digital pathology for routine histopathology diagnosis in a multicenter hospital network," *Arch. Pathol. Lab. Med.*, vol. 144, no. 2, pp. 221–228, Feb. 2020.

[4] X. Liu, L. Faes, A. U. Kale, S. K. Wagner, D. J. Fu, A. Bruynseels, T. Mahendiran, G. Moraes, M. Shamdas, C. Kern, J. R. Ledsam, M. K. Schmid, K. Balaskas, E. J. Topol, L. M. Bachmann, P. A. Keane, and A. K. Denniston, "A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: A systematic review and meta-analysis," *Lancet Digit. Health*, vol. 1, no. 6, pp. e271–e297, Oct. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2589750019301232

[5] J. Ker, Y. Bai, H. Y. Lee, J. Rao, and L. Wang, "Automated brain histology classification using machine learning," *J. Clin. Neurosci.*, vol. 66, pp. 239–245, Aug. 2019.

[6] J. Ker, L. Wang, J. Rao, and T. Lim, "Deep learning applications in medical image analysis," *IEEE Access*, vol. 6, pp. 9375–9389, 2018.

[7] P. Bankhead, M. B. Loughrey, J. A. Fernández, Y. Dombrowski, D. G. McArt, P. D. Dunne, S. McQuaid, R. T. Gray, L. J. Murray, H. G. Coleman, J. A. James, M. Salto-Tellez, and P. W. Hamilton, "QuPath: Open source software for digital pathology image analysis," *Sci. Rep.*, vol. 7, no. 1, Dec. 2017, Art. no. 16878.

[8] M. Stritt, A. K. Stalder, and E. Vezzali, "Orbit image analysis: An open-source whole slide image analysis tool," *PLOS Comput. Biol.*, vol. 16, no. 2, Feb. 2020, Art. no. e1007313.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, and S. Ghemawat. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[10] F. Chollet. (2015). *Keras*. [Online]. Available: https://github.com/fchollet/keras

[11] R. Hundt, "Loop recognition in C++/Java/Go/Scala," in *Proc. Scala Days*, 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[12] L. Gherardi, D. Brugali, and D. Comotti, "A java vs. C++ performance evaluation: A 3D modeling benchmark," in *Proc. Int. Conf. Simulation, Modeling, Program. Auton. Robots*, vol. 7628, Nov. 2012, pp. 161–172.

[13] Eclipse Deeplearning4j Development Team. *Deeplearning4j: Open-Source Distributed Deep Learning for the JVM*. Accessed: Apr. 15, 2020. [Online]. Available: http://deeplearning4j.org

[14] E. Smistad, M. Bozorgi, and F. Lindseth, "FAST: Framework for heterogeneous medical image computing and visualization," *Int. J. Comput. Assist. Radiol. Surgery*, vol. 10, no. 11, pp. 1811–1822, Nov. 2015.

[15] E. Smistad, A. Ostvik, and A. Pedersen, "High performance neural network inference, streaming, and visualization of medical images using FAST," *IEEE Access*, vol. 7, pp. 136310–136321, 2019.

[16] Intel. (2019). *OpenVINO Toolkit*. Accessed: Jun. 10, 2019. [Online]. Available: https://software.intel.com/openvino-toolkit

[17] NVIDIA. (2019). *TensorRT*. Accessed: Jun. 10, 2019. [Online]. Available: https://developer.nvidia.com/tensorrt

[18] M. Linkert, C. Rueden, C. Allan, J.-M. Burel, W. Moore, A. Patterson, B. Loranger, J. Moore, C. Neves, D. Macdonald, A. Tarkowska, C. Sticco, E. Ganley, M. Rossner, K. Eliceiri, and J. Swedlow, "Metadata matters: Access to image data in the real world," *J. Cell Biol.*, vol. 189, pp. 777–782, May 2010.

[19] M. Satyanarayanan, A. Goode, B. Gilbert, J. Harkes, and D. Jukic, "OpenSlide: A vendor-neutral software foundation for digital pathology," *J. Pathol. Informat.*, vol. 4, no. 1, p. 27, 2013. [Online]. Available: http://www.jpathinformatics.org/article.asp?issn=2153-3539;year=2013;volume=4;issue=1;spage=27;epage=27;aulast=Goode;t=6

[20] U. Schmidt, M. Weigert, C. Broaddus, and G. Myers, "Cell detection with star-convex polygons," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assist. Intervent.*, Jun. 2018, pp. 265–273.

[21] G. Litjens. (2017). *ASAP*. [Online]. Available: https://github.com/geertlitjens/ASAP

[22] R. Marée, L. Rollus, B. Stevens, R. Hoyoux, G. Louppe, R. Vandaele, J.-M. Begon, P. Kainz, P. Geurts, and L. Wehenkel, "Cytomine: An open-source software for collaborative analysis of whole-slide images," *Diagnostic Pathol.*, vol. 1, no. 8, p. 13, 2016.

[23] The Qt Company. *Qt 5*. Accessed: Oct. 1, 2020. [Online]. Available: http://www.qt.io

[24] A. Howard, A. Zhmoginov, L.-C. Chen, M. Sandler, and M. Zhu, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," in *Proc. CVPR*, 2018.

[25] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[26] G. Aresta *et al.*, "BACH: Grand challenge on breast cancer histology images," *Med. Image Anal.*, vol. 56, pp. 122–139, Aug. 2019.

[27] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assist. Intervent.* Springer, 2015, pp. 234–241.

[28] M. J. Engstrøm, S. Opdahl, A. I. Hagen, P. R. Romundstad, L. A. Akslen, O. A. Haugen, L. J. Vatten, and A. M. Bofin, "Molecular subtypes, histopathological grade and survival in a historic cohort of breast cancer patients," *Breast Cancer Res. Treatment*, vol. 140, no. 3, pp. 463–473, Aug. 2013.

[29] J. Gamper, N. A. Koohbanani, K. Benet, A. Khuram, and N. Rajpoot, "PanNuke: An open pan-cancer histology dataset for nuclei instance segmentation and classification," in *Proc. Eur. Congr. Digit. Pathol.* Cham, Switzerland: Springer, 2019, pp. 11–19.

[30] J. Gamper, N. A. Koohbanani, K. Benes, S. Graham, M. Jahanifar, S. A. Khurram, A. Azam, K. Hewitt, and N. Rajpoot, "PanNuke dataset extension, insights and baselines," 2020, *arXiv:2003.10778*. [Online]. Available: http://arxiv.org/abs/2003.10778

[31] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767
[32] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham, Switzerland: Springer, 2014, pp. 740–755.
[33] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, nos. 3–4, pp. 591–611, Dec. 1965, doi: 10.1093/biomet/52.3-4.591.
[34] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Roy. Stat. Soc., B, Methodol.*, vol. 57, no. 1, pp. 289–300, Jan. 1995.
[35] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, vol. 5, no. 2, pp. 99–114, 1949.
[36] M. Veta *et al.*, "Predicting breast tumor proliferation from whole-slide images: The TUPAC16 challenge," *Med. Image Anal.*, vol. 54, pp. 111–121, May 2019.
[37] S. H. Kassani, P. H. Kassani, M. J. Wesolowski, K. A. Schneider, and R. Deters, "Classification of histopathological biopsy images using ensemble of deep learning networks," Sep. 2019, *arXiv:1909.11870*. [Online]. Available: https://arxiv.org/abs/1909.11870
[38] O.-J. Skrede, S. De Raedt, A. Kleppe, T. S. Hveem, K. Liestøl, J. Maddison, H. A. Askautrud, M. Pradhan, J. A. Nesheim, F. Albregtsen, I. N. Farstad, E. Domingo, D. N. Church, A. Nesbakken, N. A. Shepherd, I. Tomlinson, R. Kerr, M. Novelli, D. J. Kerr, and H. E. Danielsen, "Deep learning for prediction of colorectal cancer outcome: A discovery and validation study," *Lancet*, vol. 395, no. 10221, pp. 350–360, Feb. 2020.
[39] K. R. J. Oskal, M. Risdal, E. A. M. Janssen, E. S. Undersrud, and T. O. Gulsrud, "A U-Net based approach to epidermal tissue segmentation in whole slide histopathological images," *Social Netw. Appl. Sci.*, vol. 1, no. 7, p. 672, Jul. 2019.
[40] P. Bándi, M. Balkenhol, B. van Ginneken, J. van der Laak, and G. Litjens, "Resolution-agnostic tissue segmentation in whole-slide histopathology images with convolutional neural networks," *PeerJ*, vol. 7, p. e8242, Dec. 2019.
[41] Z. Guo, H. Liu, H. Ni, X. Wang, M. Su, W. Guo, K. Wang, T. Jiang, and Y. Qian, "A fast and refined cancer regions segmentation framework in whole-slide breast pathological images," *Sci. Rep.*, vol. 9, no. 1, p. 882, Dec. 2019.

ANDRÉ PEDERSEN received his civil engineering degree in Applied Physics and Mathematics with specialization in machine learning and statistics in 2019, at the Arctic University of Norway (UiT) in Troms121, Norway. Currently, he is pursuing a PhD in medical technology with focus on artificial intelligence for improved breast cancer prognostication at the Norwegian University of Science and Technology (NTNU). He also works part-time at SINTEF Medical Technology. His main research interests include statistics, medical image analysis, deep learning and computational pathology and radiology.

MARIT VALLA is a specialist in pathologist from 2013, with a PhD in medicine from 2017. Her main research interests include digital pathology and the use of artificial intelligence-based methods, molecular pathology, and translational research. She works as an associate professor at NTNU and as a consultant pathologist at St. Olavs Hospital, Trondheim University Hospital.

ANNA M. BOFIN is Professor of Medicine (Pathology) and Academic leader of the Medical Student Research Programme at NTNU. She is PI of the Breast Cancer Subtypes research project. She graduated from the Royal College of Surgeons in Ireland, University of Medicine and Health Sciences, in 1979, became a specialist in Pathology in 1992, and Doctor Medicinae (D.M.Sc.), NTNU in 2004. Her main research interests include tissuebased studies of breast cancer, molecular pathology and translational research.

JAVIER PÉREZ DE FRUTOS received his industrial engineer degree in industrial electronics and automation in 2014, and his MSc degree in automation and robotics in 2016, both at the Universidad Politecnica de Madrid (UPM) in Madrid, Spain. Currently, he is a PhD candidate working on machine learning methods for image to image registration at SINTEF Medical Technology and the Norwegian University of Science and Technology (NTNU), and a researcher at SINTEF Medical Technology. His main research interests include image processing, use of robotics in medical applications and deep learning.

INGERID REINERTSEN received her civil engineering degree in physics in 1999 from Institut National des Sciences Appliquees (INSA), Toulose, France and her MSc in medical physics from McGill University, Montreal, Quebec, Canada in 2002. She completed her PhD in Biomedical engineering at McGill University in 2007. She is currently working as a Senior Reseach Scientist at rSINTEF Digital, Medical Technology and holds a position as Adjunct Associate Professor at the Norwegian University of Science and Technology (NTNU). Her research interests include medical image processing in radiology and pathology, intraoperative imaging and ultrasound.

ERIK SMISTAD received his civil engineering degree in computer science in 2012, and his PhD in medical image processing in 2015, both at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. Currently, he his working as a research scientist at SINTEF Medical Technology, an independent not-for-profit research organization, and as a post doctoral researcher at NTNU. His main research interests include medical image processing, deep learning, ultrasound and GPU and parallel computing. His personal webpage can be found at https://www.eriksrnistad.no.