Service-based Processing of Gigapixel Images

Florian Fregien
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam,
Germany
florian.fregien@student.hpi.de

Sebastian Pasewaldt
Digital Masterpieces GmbH
Potsdam, Germany
mail@digitalmasterpieces.com

Jürgen Döllner
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam,
Germany
juergen.doellner@hpi.de

Matthias Trapp
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam,
Germany
matthias.trapp@hpi.de

ABSTRACT

With the ongoing improvement of digital cameras and smartphones, more and more people can acquire high-resolution digital images. Due to their size and high performance requirements, such Gigapixel Images (GPIs) are often challenging to process and explore compared to conventional low resolution images. To address this problem, this paper presents a service-based approach for GPI processing in a device-independent way using cloud-based processing. For it, the concept, design, and implementation of GPI processing functionality into service-based architectures is presented and evaluated with respect to advantages, limitations, and runtime performance.

Keywords: Gigapixel Images, Image Processing, Web Technologies, Service-based Processing

1 INTRODUCTION

Nowadays, the acquisition of digital images is an essential part of everyday life. With respect to this, the acquisition technologies have been constantly improving, resulting in an increase of spatial and temporal resolution and precision. Especially the increased (spatial) resolution demands for efficient and scalable approaches for their processing and display. Besides desktop-publishing, advertising and marketing, high-resolution images play an important role in the field of cultural heritage [15], medical analysis [11], and geospatial science [20].

1.1 Problem Statement

One example of such high-resolution images are GPIs, which often comprise several billion pixels. Acquisition of such images is usually more complex and its processing and exploration places high demands on a system [8]. Standard consumer hardware is often not sufficient to process and display GPIs due to the high memory requirements. This is especially true for mobile devices such as smartphones that are limited in processing performance and memory.

To approach this problem, specific software is required for managing and distributing performance-intensive processing tasks. Regarding this, advances in cloud-computing technology and cloud-based microservices represent a suitable alternative to monolithic on device software systems.

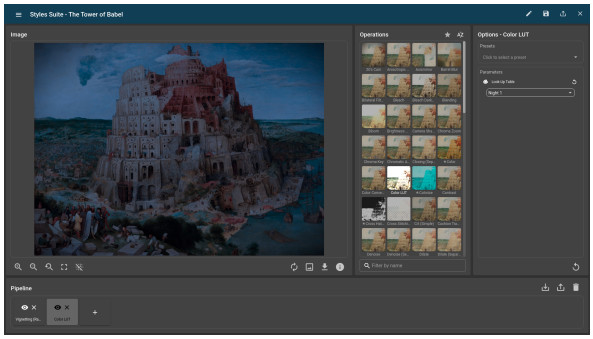


Figure 1: Exemplary screenshot of our service-based application for the exploration and processing of GPIs. It shows the components of our web-based front-end that displays the results of a color grading technique that is applied to a GPI comprising approximately $30\,000 \times 20\,000$ pixels. The provisioning and processing of the GPI was performed using back-end services.

In the context of this work, the technical challenges of integrating a system for processing GPIs into a cloud-based platform is addressed. For this purpose, the representation and storage of GPI have to be addressed first. Further, microservices architectures for image processing are be extended to support GPI, e.g., using a tile-based approach. Furthermore, interfaces are be developed and implemented that enable the communication between front-end and back-end systems.

1.2 Approach and Contributions

This paper describes how the exploration and processing of GPIs can be integrated into to service-based architectures (Figure 1). Therefore, it investigates how such images can be stored and processed in the back-end and accessed by a web-based front-end. It describes the specifics of integrating point-based, neighborhood-based, and global image processing techniques in such a system.

The remainder of this paper is organized as follows. Section 2 describes the concept of GPIs as well as previous and related work on the topic of service-based image processing. Section 3 describes a microservice architecture for processing GPIs. It covers the storage of the images in the back-end, how they can be processed by an image processor, and how the interaction with a front-end is performed in order to display the images efficiently. Section 4 describes the implementation of these concepts and how the processing of the images for point-based, neighborhood-based, and global operations is enabled. Section 5 then evaluates the implementation in terms of runtime and discusses the advantages and disadvantages of the implementation compared to a traditional on-device monolithic approach. Finally, Section 6 summarizes the paper and present future research ideas.

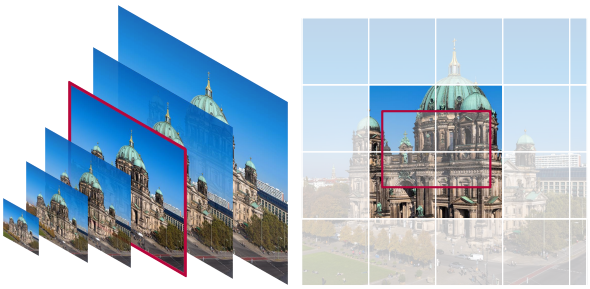
2 RELATED WORK

2.1 Gigapixel Images

While smartphone cameras can capture images with several million pixels, a specific camera setup and software pipeline for automatic composition allows images with several billion pixels to be captured, even with a standard camera and lens [8, 3]. In this work, we refer to the work of Kopf *et al.* [8], who describe how the acquisition and display of GPIs can be implemented. To integrate a GPI-viewer into a service-based architecture, three main requirements are considered: (1) a lossless display to ensure that all information could be extracted, (2) smooth panning & zooming, and (3) responsiveness for an efficient navigation in a GPI [14].

An image pyramid is suitable for storing and loading the image content, whereby the layers are divided into tiles of a smaller number of pixels (Figure 2) [1]. Tile-based approaches enable memory-efficient viewing and processing of GPIs [13]. A GPI viewer loads the tiles closest to the current display first and stores them in a cache. When the user interacts via panning or zooming, the required tiles are loaded from the cache and, if necessary, additional tiles are loaded from the memory. When the cache is full, the tiles furthest away from the current view are removed from it again.

The tile-based approach enables further improvements such as parallel processing of image tiles [4]. Such approaches can also be used to implement image analysis or image segmentation techniques for GPIs [7].



(a) (b)

Figure 2: Concept of image pyramid and tiled image: (a) shows an image pyramid for a GPI. The highlighted layer represents the current zoom level and is used for display; (b) shows the subdivision of the image layer in different tiles. The red bordered rectangle illustrates the viewport and the visible portion of the GPI. Only the tiles contained in the rectangle have to be processed.

2.2 Technologies for GPIs

There are different software libraries available for fetching and displaying GPIs, i.e., for handling such in back-end as well as front-end. The open source project *OpenSeadragon* [2] is based on the Deep Zoom technology, which was originally developed by Seadragon Software and later by Microsoft Live Labs and Google [6]. It is developed in JavaScript (JS) and can be integrated into an existing web application. It supports multiple formats for zooming images and offers various interaction options for users of desktop and mobile devices, including zooming and panning. We forked the project for adjustments required by our microservice platform.

2.3 Microservice Infrastructures

In recent years, microservices have become increasingly popular. Unlike monolithic systems, microservices are autonomous modules that perform various tasks with respect to the business logic. The advantages of using microservices are (1) increased scalability of the components, (2) easy deployment and maintainability as well as, (3) the possibility to introduce various technologies into one system [18].

For our work, we are extending a microservice platform for cloud-based visual analysis and processing that was first presented by Richter *et al.* [16]. Based on that, Wegen *et al.* [19] present an approach for performing service-based image processing using software rendering to balance cost-performance relation.

3 CONCEPT

3.1 GPI Representation

The presented service-based application for image processing is primarily used to stylize photos. We chose the Deep Zoom Image (DZI) format that uses an image pyramid to generate a scale-space of the image. If a user requests to view the entire image for overview, it must be possible to display it on a screen with resolution lower than this of the GPI. A smaller version of the image is better suited for this purpose, as it is also small enough to be loaded into Random Access Memory (RAM). Using the DZI format, the reduction of the resolution per layer is always halved until reaching a 1×1-pixel image. The individual layers are further divided into tiles using a given resolution and overlap (Figure 3). The redundancy introduced by the overlap avoid errors during processing and display.

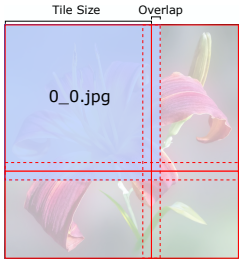


Figure 3: Tiled image as used in DZI format.

Since GPIs have a high spatial resolution, the file size tends to be usually large. With a lossy compression method as used by Joint Picture Experts Group (JPEG) file format, the file size can be kept small. For an image with several Gigapixels (GPs), the file size is still in the Gigabyte range [8]. For this reason, it is feasible not to store these images in a local file system, but to store them in a cloud-based storage solution such as Amazon Web Services (AWS) or Firebase. For our implementation, we decided to use AWS. This allows us to provide a system that is easily scalable and the required storage can be made available as required.

A DZI consists of several files, which are organized and stored in a well-defined structure. For this reason, and to keep communication between different services to a minimum, it makes sense to send these files in one "package" if possible. Of course, additional operations are necessary for a complete DZI, such as requesting the complete DZI for an export functionality or deleting it. Furthermore, we also need to be able to perform operations on individual files of a DZI, such as requesting the definition file or individual tiles for display or image processing. This is a difference to the saving of normal images that we have to consider during the later implementation (Section 4.2).

3.2 Service-based Architecture

To enable processing and exploration of cloud-stored GPIs, a suitable infrastructure is required to enable communication between the following components.

3.2.1 Resource Resolver Service (RRS)

Direct access to a third-party storage management system with a vendor-specific Application Programming Interface (API) is difficult to maintain and extend. To approach this, the RRS service is responsible for managing the files in a storage and abstracts from the vendor specific APIs. It accepts and stores complete DZIs encoded as archives (e.g., ZIP) to preserve the respective folder structure. Thus, a DZI can be transferred with a single message to the service, which can then unpack the archive and store the individual files in the cloud. It also request all files from the storage, compress them into an archive, and deliver them to other services within the system. Furthermore, the RRS can deliver individual files from an archive or accept individual files to add them for an archive export.

3.2.2 Resource Manager Service (RMS)

The RMS serves as an interface between data storage and other services and manages access via user roles. It acts as a gateway and regulates what can be requested from and sent to the RRS service. Specific to GPIs, this service ensures that only valid files are uploaded, i.e., it supports the concept of multi-resolution images to decide whether a request is valid or not. Further, it ensures that only authorized users have access to individual files or to the complete DZI. Finally, if both conditions are met, the service forwards all requested Create, Read, Update, Delete (CRUD) operations to the RRS.

3.2.3 Image Processor Service (IPS)

This service connects the services above and serves as an interface to a front-end. It forwards data requests to the RMS and sends processing requests to the Graphics Processing Unit (GPU)-Processor. The processed results are stored using the RMS and delivered to the front-end for display. Furthermore, it can perform simple image transformations, such as cropping or resizing. The IPS is also responsible for converting images into the DZI format, compositing of DZI tiles, and exporting a GPI.

3.2.4 GPU-Processor

The GPU-Processor is responsible for processing GPI tiles with hardware-accelerated graphics APIs such as OpenGL or Vulkan. For it, (1) tiles are loaded as textures into the Video Random Access Memory (VRAM), (2) the processing operations defined as Visual Computing Assets (VCAs) are applied as shader programs [5], and (3) the results are read back into RAM and returned as a response.

3.3 Interactive Exploration of GPIs

A front-end for processing GPIs should support at least the following functions: (1) selection and upload to the

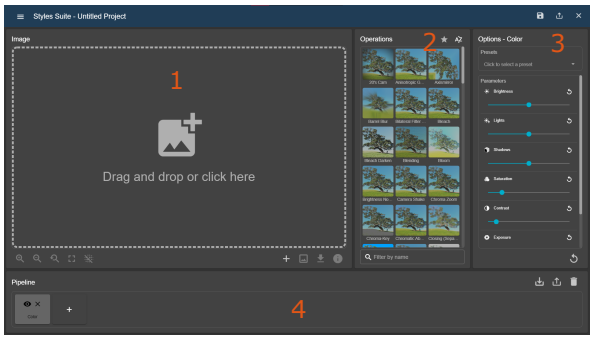


Figure 4: Proposed front-end for displaying and processing GPIs with the following components: (1) Image Canvas, (2) Operations View, (3) Options View, and (4) Pipeline View.

cloud, (2) display and processing, (3) presentation of options for export and download, as well as (4) options to select, compose, and configure different processing operations and their parameters. The following components provide these functionality (Figure 4):

Image Canvas: To address functions 1 to 3, a GPI can be selected and uploaded in a background task. Subsequently, image tiles are requested and displayed accordingly by supporting panning & zooming. For it, the Image Canvas decides which tiles from which image pyramid level should be displayed. Furthermore, actions such as downloading a DZI as an archive, converting it into a normal image, replacing the image, or displaying image metadata can be performed. It also provides explicit control of the zoom levels and full-screen display.

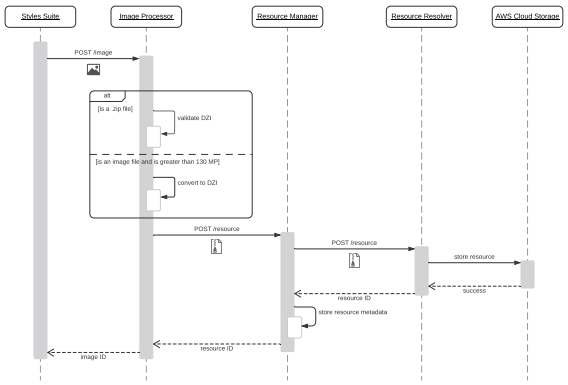
Operation View: To implement function 4, the Operation View depicts a list of processing operations applicable to an GPI. Upon operation selection, it is added to the processing pipeline and the Option View is updated accordingly. Simultaneously, processing is triggered and the result is displayed.

Option View: The Options View displays the various parameters for a selected operation. The user can adjust these as desired and trigger GPI processing subsequently. For easy selection of visually appealing parameter values, presets can be selected.

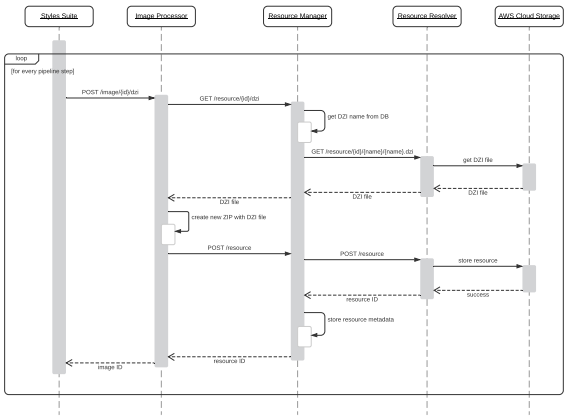
Pipeline View: To allow more complex image transformations, the system allows for combining different operation within a pipeline. For it, the Pipeline View can be used to add and select additional operations from the Operation View. Further actions include clearing the pipeline, and importing or exporting the pipeline.

3.4 GPI Processing Workflow Overview

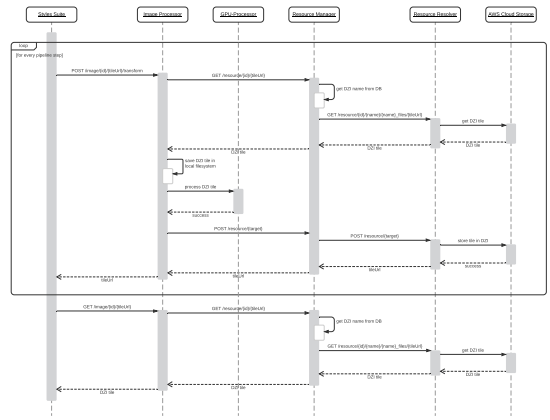
This section describes how the above components interact to process GPIs (cf. Figure 5).



(a) DZI Uploading



(b) GPI Preparation



(c) DZI Processing

Figure 5: Sequence diagrams for the data flow and communication of the respective microservice components for uploading a DZI (a), data preparation for processing (b), and the actual DZI processing (c) (please zoom).

3.4.1 *GPI Upload and Display*

First, the user selects a GPI file via the Image Canvas. The selected file is then sent to the IPS. If the image is available as an archive, it is checked whether it is a valid DZI by validating the definition file and ensuring

that all required tiles are present. If the image has been uploaded as a normal image, the IPS checks if it exceeds a certain number-of-pixels and converts it to the DZI format if necessary. In our implementation, this limit is 130 Megapixels, which is approx. 16K resolution. The valid DZI is then transmitted to the RMS, which forwards it to the RRS. The RRS provides the DZI with an Identifier (ID) and unpacks all files from the archive and stores them under this identifier in the cloud storage.

The ID is sent back to the RMS, which then stores the image in a database with metadata, such as name of the image and ID of the owner. The image ID is then sent back to the IPS, which sends it to the front-end (Figure 5a). Subsequently, the front-end requests the DZI definition file using the ID. The definition file can then be passed to the Seadragon viewer in the Image Canvas. Based on the information from the definition file, the Seadragon viewer computes which image layers and which tiles are available per layer. During interaction, the viewer identifies the tiles to be displayed and request these accordingly. If all required tiles have been requested, they are displayed to the user in the Seadragon viewer as if they were a complete image.

3.4.2 GPI Data-Preparation

A user can create a processing pipeline comprising several operations with different parameters to be applied to the DZI in a non-destructive way. To keep the computation effort low, the individual tiles are only processed on demand. First, a copy of the DZI is created by the IPS for each operation in the pipeline. For it, the IPS requests the respective file from the cloud storage and compresses it into a new archive. This is then stored as a new DZI at the RMS and RRS, the resulting ID is sent to the front-end and stored for the respective pipeline steps. In this way, processed tiles can be written to the new DZIs. Since the original resolution of the image, the tile resolution, and the overlap in the processing steps do not change, the definition file is also the same between all images in the pipeline (Figure 5b).

3.4.3 GPI Processing

Now the definition file of the last pipeline image, which is the result image, can be requested. After reading the definition file, the Seadragon viewer now requests the required tiles that are currently not yet available in the result DZI. For each tile of the requested tiles, the following steps are executed: For each pipeline step, the ID of the previous pipeline image and the tile (level, tile name) is sent to the IPS together with the operation specification (ID, parameters). In addition, the ID of the current pipeline step is sent as the "target" to be able to store the processed tile in the respective pipeline DZI. The IPS now requests the tile from the RMS (and the RMS from the RRS) and sends it to the GPU-Processor together with the operation information. The GPU-Processor applies the operation to the tile. The IPS sends the result back to the RMS to save the tile for the given "target" DZI. When all pipeline steps for a tile have been completed according to this principle, the result tile is finally requested from the IPS via RMS and RRS and subsequently displayed by the Seadragon viewer (Figure 5c).

4 IMPLEMENTATION DETAILS

This section covers development technology (Section 4.1), Representational State Transfer (REST) routes for communication between front- and back-end (Section 4.2), as well as specifics of image processing details (Section 4.3) of our prototypical implementation.

4.1 Back-end & Front-end Technology

For the implementation of the service-based architecture for processing GPIs, the following technologies are used. The respective microservices are developed using JS and executed by the JS runtime environment NodeJS. It offers various modules for handling local file systems or for executing programs as child processes. Further, Express is used to define web service APIs based on REST. The Axios library enables the microservices to communicate over HyperText Transfer Protocol (HTTP). The IPS service communicates with the GPU-Processor via the WebSocket protocol. For high-level image processing, such as reading image metadata or converting images to DZI format, the Sharp library is used [10]. For it, a C++ binding for the VIPS image-processing library [12] provides an API that can be used in NodeJS. As a low-level image processing library, libvips implements more complex operations such as converting a DZI into a normal image by merging the individual tiles. To implement the web application front-end, the Angular framework (based on TypeScript) is used in combination with an adapted fork of the open source library OpenSeadragon.

4.2 Interfacing Front-end & Back-end

This section describes the REST-interface offered by the IPS to handle GPIs using the following routes:

`GET /image/:id` This route returns a DZI addressed by a given ID as a ZIP archive. The IPS service uses the ID to request the image from the RMS. As a parameter, the ID must be specified as a Universally Unique Identifier (UUID).

`POST /image` This route takes as parameter a file which is either a normal image or an DZI archive. In the case of a normal image, the size of the image

is checked. If it exceeds a limit, the image is converted to DZI format using the Sharp library, which provides the `tile()` function for conversion. Then the DZI is transferred to the RMS for cloud storage. The resource ID, obtained by the RMS response, is forwarded to the IPS as a request response.

`GET /image/:id/dzi` This route returns the definition file of a DZI using a given ID. This request is forwarded directly to the RMS.

`GET /image/:id/:level/:tile` This route returns a specific tile of a DZI using a given ID. The name of the DZI is also added here by the RMS. Beside the ID of the DZI the `tileUrl`, which consists of the level of the tile in the image pyramid and the tile coordinates in the form `x_y.format` (where `format` is element of {`png`, `jpeg`, ...}), has to be specified.

`POST /image/:id/dzi` This route copies a DZI of a given ID and removes all files except the definition file from the copy. For it, the IPS service requests the definition file of the DZI with the ID specified in the path, compresses this file into a new archive, and uploads it back to the RMS for storage. The resulting ID is forwarded as response.

`PUT /image/:id/dzi` Similar to the POST route, all files except the definition file (i.e., all tiles) are deleted from the DZI under the ID given in the path.

`GET /image/:id/export` This route exports the DZI using the given ID to a normal image and delivers it as response. It is assumed that at least all tiles of the maximum level of the image pyramid are present. How the tiles are joined to form an image is discussed in Section 4.3.3.

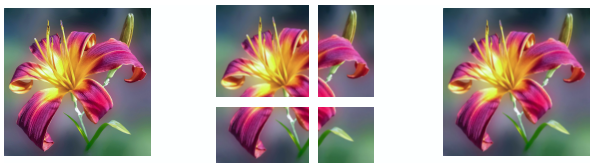
`POST /image/:id/:level/:tile/transform` This route is used to process a tile in the GPU-Processor. The tile with the given `tileUrl` is requested from the DZI with the given ID at the RMS. Further parameters are sent as `FormData` object in the body of the request.

4.3 GPI Processing Operations

4.3.1 Operation Types

There are different types of operations that the GPU-Processor can apply: point-based, neighborhood-based, and global operations.

Point-based Operations: For point-based operations, a pixel is only computed based on the original pixel. An example of a point-based operation is "Grayscale". In a simple implementation, the average of the Red, Green, and Blue (RGB) values of the pixel is computed and assigned to all three channels. Since point-based operations do only operate on one pixel at a time, they can directly be applied to tile-based processing without adjusting the implementation.



(a) Mean Blur operation (kernel of 4 pixels) applied to a DZI level.

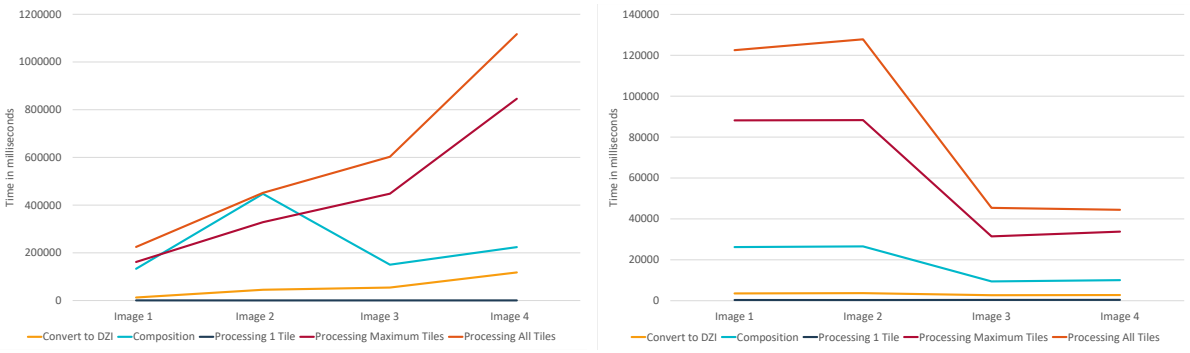
(b) Vignette operation (with a radial size of 0.58 and a radial smoothness of 0.21) applied to one level of a DZI without UV correction.

(c) Vignette operation (with a radial size of 0.58 and a radial smoothness of 0.21) applied to one level of a DZI with UV correction.

Figure 6: Left column shows the original image, middle column shows the operations applied to the single tiles, and right column shows the results after compositing.

Neighborhood-based Operations: These operations use not only the values of a pixel, but also the values of pixels that are in the neighborhood of that pixel. The computation is based on a kernel that can be of any size and weight to determine how the pixels in the neighborhood are contributing. As an example operation, we consider the "Mean Blur" filter. In this operation, the average of all RGB values of the pixels affected by the kernel is computed and set as RGB value for the output pixel. For a larger kernel, the effect is usually stronger, but the computation also takes longer, because more pixels have to be requested.

With tile-based processing, such operations cannot be used as straightforward as for the point-based operations. A pixel at the edge of a tile needs information from the neighboring tile for correct computation because the kernel overlaps. To solve this problem, we can take advantage of the overlap option in the DZI format. The overlap, which is set when an image is converted to a DZI, appends a certain number of overlap-pixels of the neighboring tile (cf. Figure 3). This allows the kernel to access the neighboring pixels for the edge of the tile if the overlap is



(a) Test Case 1

(b) Test Case 2

Figure 7: Duration of operations in milliseconds.

greater than or equal to the kernel radius. Figure 6a shows how to apply the Mean Blur filter to an image consisting of four tiles.

Global Operations: A global operation considers the entire image, e.g., tone-mapping or compositing/blending. Operations that render a texture over the input image use UV coordinates to determine the size and position of the texture. Since the UV coordinates must be in the interval $[0,1]$, the image size is needed to scale the absolute pixel coordinates. Here the coordinate origin is in the lower left corner of the image and in the upper right corner is the point $(1,1)$. An exemplary operation for this category is the "Vignette".

4.3.2 Tile-based Processing

If all tiles are processed individually and such an operation is applied, it will only be computed in relation to these tiles, because the UV coordinates will be determined relative to the size of the tile. This leads to unwanted results, as shown in Figure 6b. To approach the problem, the UV coordinates are recomputed relative to the entire image prior to performing the operation. For it, we pass the original size of the image and the absolute position of the tile to the GPU-Processor.

To compute the correct UV coordinates, two steps are required: (1) the normalized coordinates in $[0,1]^2$ must be scaled relative to the tile resolution by the original size and (2) the coordinates must be translated so that they are relative to the tile in the complete image. For efficiency, these computations are performed on the GPU using vertex shader as follows. First, two quantities that do not depend on vertex coordinates are computed and and represented as uniform variables: (1) the *relative tile resolution* is obtained by dividing the absolute tile resolution in pixels by the size of the original image; (2) similar, absolute coordinates of the tile are converted into relative coordinates. In the vertex shader, these are used by uniform variables for texture sampling, e.g., `u_TileSize` and `u_TilePosition`. For it, first the tile position is flipped, because the origin of the tiles is in the upper left corner, but the origin of the UV coordinates is in the lower left corner. Subsequently, the UV coordinates are scaled to the tile using `u_TileSize` and translated by a tile-offset. During rasterization, the coordinates are passed to the fragment shader(s) that perform the processing. Figure 6c shows the result for the image of Figure 6b using the converted UV coordinates.

4.3.3 Tile Compositing

The export route converts a DZI into a normal image by using the NodeJS library Sharp in combination with libvips. For it, (1) the number of tiles exist at the maximum level of the image pyramid is computed, (2) all respective tiles are requested from the RMS, and (3) stored in the local file system. Subsequently, a child process is spawned for each tile, using libvips to remove the overlap from the tiles. Finally, all tiles are joined to a single output image.

5 RESULTS & DISCUSSION

5.1 Runtime Performance Evaluation

In this section, the runtime performance of our implementation is evaluated regarding two aspects: (1) the time consumed to process GPIs and (2) how the timings depends on different operations regarding the tile resolution and overlap. For both test cases, we use a dedicated GPU-server equipped with a Xeon E5-2637 v4, 3.50 GHz processor (8 cores), 64 GB RAM, and a NVIDIA Quadro M6000 24 GB VRAM.

5.1.1 Test Case 1

Setup: For this test, we measure the runtime for the following operations: (1) image conversion to DZI format, (2) processing of a single tile, (3) processing of all tiles at the maximum level, (4) processing of all tiles of the DZI, and (5) compositing and exporting the tiles to a complete image. For processing, we use a point-based operation, i.e., color adjustments via color

Table 1: Input data for Test Case 1

	Image 1	Image 2	Image 3	Image 4
Resolution (Pixels)	34 561 × 15 620	35 690 × 28 030	34 861 × 44 360	64 172 × 45 559
Gigapixel	0.54 GP	1.00 GP	1.55 GP	2.92 GP
Size (MB)	129.4	894.9	939.6	1601.6

look-up tables [17]. Table 1 shows the four test images and their metadata. The tile resolution is 512 pixels and the overlap is 2 pixels. The timings were determined using the functions `console.time()` and `console.timeEnd()`. Three consecutive measurements were obtained for each operation and the average was calculated subsequently.

Results: Figure 7a shows the duration of the operations in milliseconds. The operations "Convert to DZI", "Processing Maximum Tiles", and "Processing All Tiles" are linear with respect to the input size of the image. The operation "Processing Single Tile" is constant at about 500 ms. This also meets the expectation, since the tiles are all the same size and should therefore be processed at about the same speed. The "Composition" operation also appears to be linear in the size of the images, but Image 2 is an outlier in this measurement. The reason for this must be the file format, which impacts gathering and transmission times. Even if all images are in JPEG format, they can have different compression rates, which can increase the file size of the image and its complexity. The file size (Table 1) also shows this: Although Image 2 has just under twice as many pixels as Image 1, the file size is almost seven times higher. Furthermore, Image 3 has about 50 % more pixels than Image 2 and yet their file sizes are almost the same. However, according to the measurements, this difference does not play a role when converting to the DZI format. This is probably due to the fact that only small parts are removed from the image and stored in a tile. Composition, on the other hand, is the process of merging a large image from all the tiles, which means that the actual saving of the image is more time-consuming.

5.1.2 Test Case 2

Setup: For the second test case we used a GPI of 13 206 × 6676 pixels (0.088 GP, 53.3 MB). For test purposes, we decided to use an image with a smaller resolution. We converted this image to the DZI format with different parameters for tile resolution and overlap. The test sizes are shown in Table 2. For these four variants, we tested the same operations as in Test Case 1 and computed the average from three subsequent measurements.

Table 2: Input resolution in pixels for Test Case 2.

	Image 1	Image 2	Image 3	Image 4
Tile Resolution	256	256	512	512
Overlap	1	16	1	16

Results: Figure 7b shows the absolute duration of the operations in ms, while Figure 8 shows the file sizes of the four DZIs. The measurement results show that the processing of a tile takes between approx. 300 ms to 400 ms. A greater overlap has the consequence that both the file size and the duration of the operations increase. This is because a higher overlap increases the redundancy of the pixels and therefore more has to be stored and calculated. It is noticeable that converting to a DZI was about one second faster with the 512 pixels tiles than with the 256 pixels tiles. The other operations – "Processing Maximum Tiles", "Processing All Tiles", and "Composition" – were even over 60 % faster for the 512 pixels tiles than for the 256 pixels tiles. Because the duration of the operations is directly proportional to the number of tiles, the operations for Image 3 and 4 are faster because there are fewer tiles due to the larger tile resolution.

5.2 Discussion

5.2.1 Service-based vs. Monolithic Approach

In the following, we will briefly discuss the advantages and disadvantages of our service-based implementation compared to a classic approach with only one service. The described microservice infrastructure enabled the development of a modular system following the Separation-of-Concerns pattern [9]. This allows to multiply different parts of the system as required, thus enabling parallel processing that increases performance compared to a monolithic system. Through the connection to the cloud storage service AWS, additional storage is available that can be expanded on demand. A

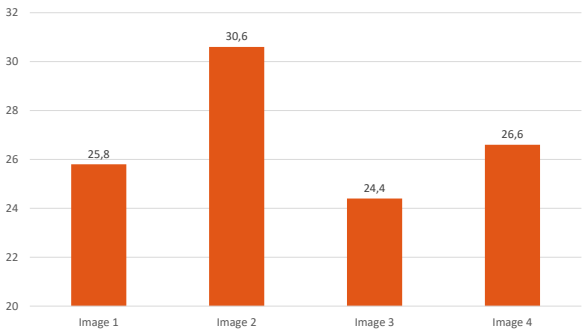


Figure 8: Test Case 2 – File size of converted images in DZI format in Megabytes.

disadvantage of the proposed approach, however, is that GPIs are required to be sent through several services, which increases the overall traffic and increases runtime. In our test system all services run on one server only, i.e., that this server is no less busy than a server on which everything would run as one service anyway. In other words, the potential of our implementation in terms of performance only becomes apparent when the services are used as a distributed system with several servers.

5.2.2 Tile Resolution and Overlap

In our implementation of DZI format conversion, constant values for tile resolution and overlap are used. Changing these parameters later is time-consuming, since the DZI would first have to be reassembled and then converted back into the DZI format. This requires to find suitable values for both parameters to process the tiles as efficiently as possible and to avoid artifacts, e.g., due to kernel sizes of neighborhood operations larger then the actual overlap. The tile resolution should not be chosen too small, because this increases the number of tiles and thus the effort when converting from and to DZI format. However, the overlap should be as small as possible because it only contains additional (redundant) data.

The results of Test Case 2 show that choosing a higher tile resolution can have a positive effect on runtime and memory consumption. This is because the higher tile resolution means that fewer tiles are created during the conversion, and therefore fewer steps per operation have to be performed. However, with a difference of less than 100 ms, the runtime for processing a 512 pixels tile is remarkably close to the runtime for a 256 pixels tile. This means that if the tile resolution is not too large, the runtime can be reduced without having to make major losses when processing a tile.

A higher value for the overlap increases runtime and file size, since more pixels are stored redundantly. A key factor in the choice of the overlap is the kernel size of neighborhood-based operations. If a kernel requires n neighboring pixels in each direction, the overlap should be at least n, to process the edge pixels of the tile correctly.

5.2.3 Context-sensitive Global Operations

Using tile-based processing, our implementation allows for the application of global operations to a DZI. The approach converts the UV coordinates transparent to the processing shaders. However, this approach does not work for global operations that require random access to all parts of an GPI, e.g., "Mirroring". To approach this problem, one could give the GPU-Processor more image context about a DZI, so that it can request additional tiles if required.

6 CONCLUSIONS & FUTURE WORK

This paper describes how service-based architectures for image processing can be extended to allow the processing of GPI. It shows how the storage of GPIs in the DZI format can be implemented with a cloud-based approach and microservices for transfer, preparation, and tile-based processing. For it, the differences between specific features of point-based, neighborhood-based, and global operations are discussed. A runtime analysis shows that efficient processing and display of GPIs is possible in a service-based infrastructure. This work can serve as a basis for further research and development in this area.

While the presented approach and prototypical implementation is sufficient for processing and exploration of GPIs, the system can be extended further. To enable the processing of high-resolution panoramic images, the support of additional projection methods (e.g., equirectangular or stereographic) are required. A service-based approach has the advantage that even less powerful devices can process images using potentially complex operations. However, this has the disadvantage that additional network traffic is generated by the communication between the individual microservices, which can result in waiting times for the user. Further, the service-based architecture allows the various microservices to be multiplied as required, thus enabling parallel processing of requests. This can also be exploited for processing GPIs, e.g., by processing different tiles in parallel. This however results in further challenges, e.g., how the tiles are transferred or how tile-access is managed between different processors.

ACKNOWLEDGMENTS

We thank Josafat-Mattias Burmeister for his support. This work has been funded by the German Federal Ministry of Education and Research (BMBF) through grants 01IS18092 ("mdViPro") and 01IS19006 ("KI-Labor ITSE").

7 REFERENCES

[1] Edward Adelson, Charles Anderson, James Bergen, Peter Burt, and Joan Ogden. Pyramid methods in image processing. *RCA Eng.*, 29, 11 1983.

[2] OpenSeadragon Contributors. Openseadragon 2.4.2, 2021. last visited: 03/30/2021.

[3] O. S. Cossairt, D. Miau, and S. K. Nayar. Gigapixel computational imaging. In *2011 IEEE International Conference on Computational Photography (ICCP)*, pages 1–8, 2011.

[4] Zhenlong Du, Xiaoli Li, Xiaojian Yang, and Kangkang Shen. A parallel multigrid poisson pde solver for gigapixel image editing. In Yunquan

Zhang, Kenli Li, and Zheng Xiao, editors, *High Performance Computing*, pages 89–98, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Tobias Dürschmid, Maximilian Söchting, Amir Semmo, Matthias Trapp, and Jürgen Döllner. ProsumerFX: Mobile Design of Image Stylization Components. In *Proceedings SIGGRAPH ASIA Mobile Graphics and Interactive Applications (MGIA)*, pages 1:1–1:8, New York, 2017. ACM.

[6] Paul Khouri-Saba, Antoine Vandecreme, Mary Brady, Kiran Bhadriraju, and Peter Bajcsy. Deep zoom tool for advanced interactivity with high-resolution images. *SPIE Newsroom*, page 1, 05 2013.

[7] Enrico Kienel and Guido Brunnett. Tile-based Image Forces for Active Contours on GPU. In P. Alliez and M. Magnor, editors, *Eurographics 2009 - Short Papers*. The Eurographics Association, 2009.

[8] Johannes Kopf, Matt Uyttendaele, Oliver Deussen, and Michael Cohen. Capturing and viewing gigapixel images. *ACM Transactions on Graphics (TOG)*, 26:93, 08 2007.

[9] Phillip Laplante. *What Every Engineer Should Know About Software Engineering*. CRC Press, 2007.

[10] libvips Contributors. libvips, 2021. last visited: 03/30/2021.

[11] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, George E. Dahl, Timo Kohlberger, Aleksey Boyko, Subhashini Venugopalan, Aleksei Timofeev, Philip Q. Nelson, Gregory S. Corrado, Jason D. Hipp, Lily Peng, and Martin C. Stumpe. Detecting cancer metastases on gigapixel pathology images. *CoRR*, abs/1703.02442, 2017.

[12] K Martinez and J Cupitt. Vips ? a highly tuned image processing software architecture. In *IEEE International Conference on Image Processing (01/09/05)*, pages 574–577, 2005. Event Dates: Sept. 2005.

[13] Mayur Patel. Memory-constrained image-processing architecture. *Dr. Dobb's Journal (DDJ)*, 22:24, 26–29, 07 1997.

[14] Dominik Perpeet and Jan Wassenberg. Engineering the ideal gigapixel image viewer. In *Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS 2011)*, Maui, Hawaii, 12 2011.

[15] Nancy Proctor. The google art project: A new generation of museums on the web? *Curator: The Museum Journal*, 54(2):215–221, 2011.

[16] Marvin Richter, Maximilian Söchting, Amir Semmo, Jürgen Döllner, and Matthias Trapp. Service-based Processing and Provisioning of Image-Abstraction Techniques. In *Proceedings International Conference on Computer Graphics, Visualization and Computer Vision (WSCG)*, pages 97–106, Plzen, Czech Republic, 2018. Computer Science Research Notes (CSRN).

[17] Jeremy Selan. Using Lookup Tables to Accelerate Color Transformations. In *GPU Gems*, pages 381–392. Addison-Wesley, 2004.

[18] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. *CoRR*, abs/1808.04836, 2018.

[19] Ole Wegen, Matthias Trapp, Jürgen Döllner, and Sebastian Pasewaldt. Performance Evaluation and Comparison of Service-based Image Processing based on Software Rendering. In *Proceedings International Conference on Computer Graphics, Visualization and Computer Vision (WSCG)*, pages 127–136, Plzen, Czech Republic, 2019. Computer Science Research Notes (CSRN).

[20] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Geosparkviz: A scalable geospatial data visualization framework in the apache spark ecosystem. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, SSDBM '18, New York, NY, USA, 2018. Association for Computing Machinery.