Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimization of the reconstruction of zebrafish on the LLSC using the VAST microscope

Max Doesburg

Supervisors:
Dr. K. F. D. Rietveld & Prof.dr.ir. F. J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

August 20, 2019

Abstract

A problem every programmer will face is the time any code can be executed in. The VAST microscope produces images of zebrafish. To transfer these images through the code that performs a reconstruction to a volume, a surface area and a 3D model takes about 80 seconds, which can be made faster.
Besides discussing improvements of speed we need to take a look at all the code in the pipeline from the VAST microscope producing images to the creation of a 3D model. This includes using a new method of image segmentation for the microscope images and how this information is transferred and saved between each step in this pipeline. In this thesis we will investigate where we can improve the current speed of the software and how these new steps are to be integrated into a functioning pipeline that produces accurate results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main goal of this thesis is speeding up the code used to process data from the VAST microscope. The original code was created by Guo et al. [6]. This code concerns itself with the automated reconstruction of a 3D zebrafish model from a set of multi axial high-resolution images. The reconstruction is specifically designed to take images from the VAST BioImager. The way the 3D model is constructed also allows us to derive the volume and surface area of the zebrafish. This code can be broadly divided into three different parts: the segmentation, the camera parameter calculations and the actual 3D reconstruction. The segmentation prepares the images made by the VAST BioImager for the reconstruction by turning the full colour images into black and white contours. The camera parameter calculations take these contours to extract several variables. The 3D reconstruction turns the set of segmented images with the help of these variables into a 3D model of a zebrafish and calculates the surface area and volume of this 3D model.

Two parts of the code were recently modified separately, one by Juliëtte Meeuwsen and one by Wilco Verhoef and Wilco de Boer [14]. This means that our the main goal can be divided into two major parts for this thesis.

The first goal is to decrease the runtime of the code provided by Juliëtte Meeuwsen [9]. The second is to integrate this code in the existing pipeline of the VAST BioImager. Juliëtte Meeuwsen converted the original 3D reconstruction code from Matlab to Python. This code was originally made to turn the set of images of a zebrafish made by the VAST BioImager that were processed by the segmentation into a 3D model. This conversion from Matlab to Python caused by a slowdown in performance for individual zebrafish, which the work in this thesis seeks to remedy. The conversion, on the other hand, made it easier to spread a workload of multiple zebrafish around on a cluster. The slowdown in individual performance brings up several questions about how the code will be brought closer to the individual zebrafish runtime on Matlab: how will this be done, where is the performance slowdown located in the code, where can we get a speedup in the code and how big can this speedup be?

The second part of this thesis concerns the process that happens before the code of Juliëtte Meeuwsen and how to recombine the first and the second segment of the code after they were changed. The first segment of the code uses segmentation to create contours of several zebrafish images taken by the microscope. This segmentation code was transformed from a Matlab implementation of several image restoration processing algorithms to a Neural Network. Due to the separate conversion of both the 3D reconstruction and the segmentation there is a need to recombine the two into one functioning pipeline again. This means we need to consider how the output of the new segmentation method flows into the reconstruction, what kind of intermediate storage are we going to use, are we losing accuracy and how much accuracy is being lost if this were the case? We have to take care of all these problems while, at the same time, keeping performance in mind.

This thesis is a bachelor project and is performed at the Leiden Institute of Advanced Computer Science (LIACS). This project is supervised by Kristian Rietveld and Fons Verbeek.

Chapter 2

Materials

In this chapter we will discuss the materials concerning the thesis.

2.1 VAST BioImager

The VAST (Vertebrate Automated Screening Technology) platform is designed for researchers who use zebrafish and need to image large numbers of 2 to 7 day old sedated zebrafish larvae using its high throughput imaging pipeline.
The VAST BioImager is a modular, expandable system that has a microscope that can take 360 degree images. The VAST BioImager can be expanded with a sample introduction option. This sample introduction option can take the form of a bulk sample reservoir, a multiwell plate or a pipettor.

Figure 2.1: The VAST BioImager connected to a microscope and two introduction options.

Source: www.unionbio.com/vast/.

The zebrafish larva gets sucked into the imaging chamber by the sample introduction option of choice, which are seen together with the microscope in Figure 2.1. The zebrafish larva is then positioned for the microscope by adjusting the pressure in the tubes. The VAST internal light source is then turned off, then in addition, multi axial high-resolution images can be taken with the microscope.

After the VAST BioImager makes the images of the zebrafish we cannot just use these images to extract data. To fully use the images made by the VAST BioImager we need to make a 3D model from these individual images.

The images of the zebrafish are transferred to a computer or a cluster where a program uses these images to construct a 3D larvae of the zebrafish with which we can measure certain data.

2.2 Microscopy

We can now describe the working of the types of microscopy used by the VAST BioImager.

2.2.1 High throughput imaging

VAST is made to enable a technique called high throughput imaging to make the images of the zebrafish that are used to construct the 3D model. High throughput imaging is using automated microscopy and image analysis to measure phenotypes which allows us to find disease mechanisms. This has allowed a large amount of automation in the microscopy process.

2.2.2 Bright-field imaging

Bright-field imaging is the standard form of imaging and is the most basic form of optical microscopy illumination techniques. We can not use normal photography because zebrafish are transparent, so taking a picture is harder than in the case of a solid test subject. Bright-field imaging is done by Illuminating the sample zebrafish with a white light source and measuring the refractions that are caused by the body of the transparent zebrafish. From these refractions we can determine where the zebrafish is located [11].

2.2.3 Fluorescent imaging

In 1962 the GFP (green fluorescent protein) was found in jellyfish. This paved the way for fluorescent imaging in several types of animals and the discovery of more fluorescent proteins. Because the zebrafish genome is so extensively documented we can change the zebrafish genome to incorporate the fluorescent proteins, as seen in Figure 2.2. When fluorescent proteins are incorporated in the zebrafish, the microscope can focus on specific parts of the zebrafish like the specific organs and blood vessels. This allows experiments to be run that can collect more data on very specific parts of the zebrafish [12].

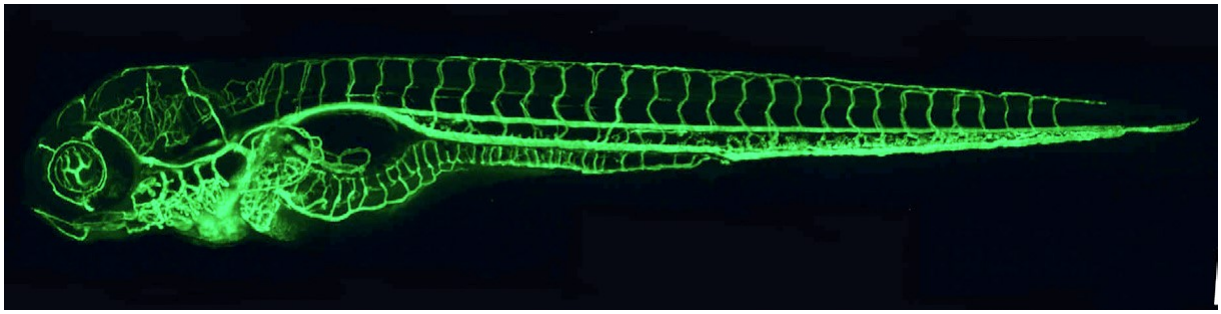

Figure 2.2: A zebrafish picture taken with fluorescent imaging.

Source: https://www.vitascientific.com/image/data/Email/VitalSigns/zebrafish.jpg

2.3 Zebrafish

Zebrafish have many advantages over other lab animals when used to study immunity. Some of the biggest advantages of zebrafish are [4]:

- Zebrafish are cheap.
- There are many more zebrafish than there are mice.
- Zebrafish are vertebrates that are genetically very similar to humans.
- Zebrafish are transparent at an early age so we can see their organs and blood vessels easily.
- Zebrafish are tiny and very robust.
- The genome of zebrafish is very well documented.

These advantages make zebrafish a good candidate for studying the human genome. The transparency combined with the genetic similarity to humans allows us to easily portray what happens inside the zebrafish and allows us to visualise and study the effects of diseases and cures inside the zebrafish.

There are some downsides however [8] [13]:

- Zebrafish are very similar to us genetically, but they are different enough to question human relevance.
- Gender is not genetically determined which causes more divergence from the human genome.
- It can be, like with all lab animals, considered ethically wrong to use zebrafish for these kinds of experiments.

2.4 Zebrafish image datasets

There are two datasets of zebrafish used in the experiments, both are made using the bright-field imaging technique from the VAST BioImager.
The first dataset uses 84 contours and was originally generated by the VAST microscope and the segmentation was generated with Matlab/Python.

A single sample of the zebrafish contains the set of contours of the zebrafish, the X, Y and Z rotations of every contour, the rotation angle of every contour and the focal length of every contour. This dataset, referred to as dataset 1, was made by Guo et al. [6].

The second set of zebrafish consists of 100 contours. This set has not been fully processed as was done for the previous set and therefore only contains the contours. The X, Y and Z rotations of every contour, the rotation angle of every contour and the focal length of every contour have to be calculated separately. This dataset, referred to as dataset 2, was made by the Neural Network of Wilco Verhoef and Wilco de Boer [14].

From the zebrafish in both datasets, a representative set of zebrafish has been selected to serve as example for the results. These zebrafish are denoted with a number, for example Zebrafish 01, Zebrafish 05 and Zebrafish 10. These results were acquired by taking the average of three separate executions of the algorithm on this zebrafish.

2.5 LLSC

The LLSC (LIACS Life Science Cluster) is the target platform for this work and this cluster will be where all the test results will be generated on. The LLSC has multiple types of nodes which will be listed below there are 24 nodes in total. Besides having these nodes the cluster has a 36 Terabyte file storage system.

- cpu-5150, 15 nodes, two Intel Xeon 5150 processors, 2.66 GHz, 2 cores per processor
- cpu-e5430, 6 nodes, two Intel Xeon e5430 processors, 2.66 GHz, 4 cores per processor
- cpu-x5355, 1 node, two Intel Xeon x5355 processors, 2.66 GHz, 4 cores per processor
- cpu-x5450, 2 nodes, two Intel Xeon x5450 processors, 3 GHz, 4 cores per processor

When you have access to the large amount of computing power of a cluster like the LLSC, multiple individuals can use the cluster at the same time and several individuals will queue jobs at the same time. On a cluster like the LLSC the resources that are available then have to be distributed to multiple individuals in the most efficient and fair manner. The code for the VAST BioImager only needs a small part of the computing power of the LLSC to run for one zebrafish, with more resources needed if more zebrafish are added. To regulate the distribution of resources on the LLSC there is a resource manager.

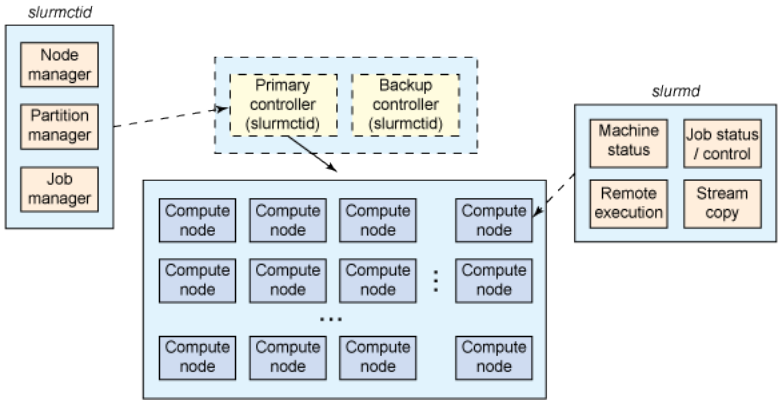

Figure 2.3: Top down view of the SLURM architecture.

Source: https://www.ibm.com/developerworks/library/l-slurm-utility/figure2.gif

Currently the resource manager used for the LLSC is TORQUE [1]. TORQUE is shorthand for Terascale Open-source Resource and QUEue Manager. TORQUE is a distributed resource manager which allows the LLSC to regulate the allocation of resources to batch jobs on the cluster.
For the LLSC the alternative to TORQUE is SLURM [2], shown in Figure 2.3. SLURM is shorthand for Simple Linux Utility for Resource Management, is also being discussed. SLURM, like TORQUE, is a resource manager and has some advantages compared to TORQUE. The most important advantage of SLURM is that it is open source and it is a more modern system. Additionally, it is well maintained in contrast to TORQUE.

Chapter 3

Methodology and Implementation

In this chapter we will discuss how the code for the VAST BioImager is implemented and the methodology behind the optimizations.

3.1 Code structure

The reconstruction code can be divided into three major parts, as seen in Figure 3.1 [6] together with the end result (d). These major parts are microscopy (a), segmentation (b) and the 3D reconstruction (c) which itself can be divided into another five parts as seen in Figure 3.2. The visual hull calculations which will construct a rough point cloud out of the segmented images. The mesh will take the point cloud and make a rough 3D model out of the points with which the volume will be determined. The model will be smoothed using the Poisson method so that the surface area can be accurately determined.

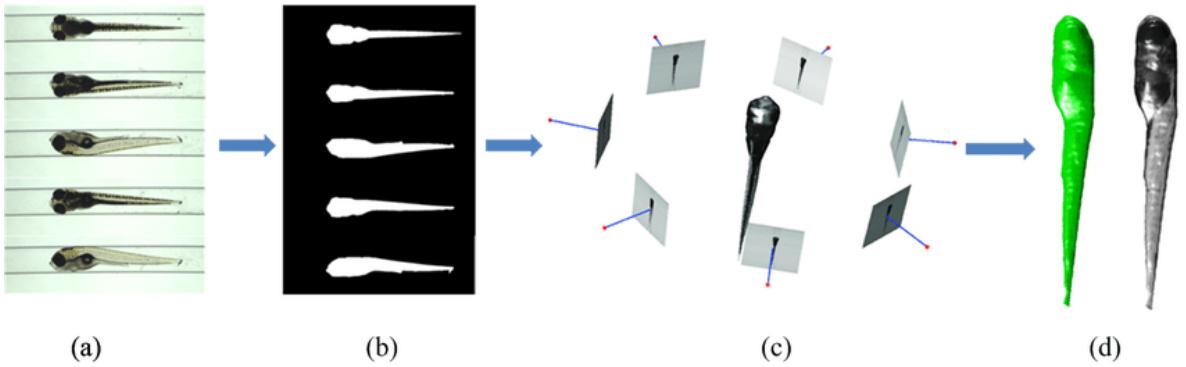

Figure 3.1: A visualisation of every major step of the code. [6]

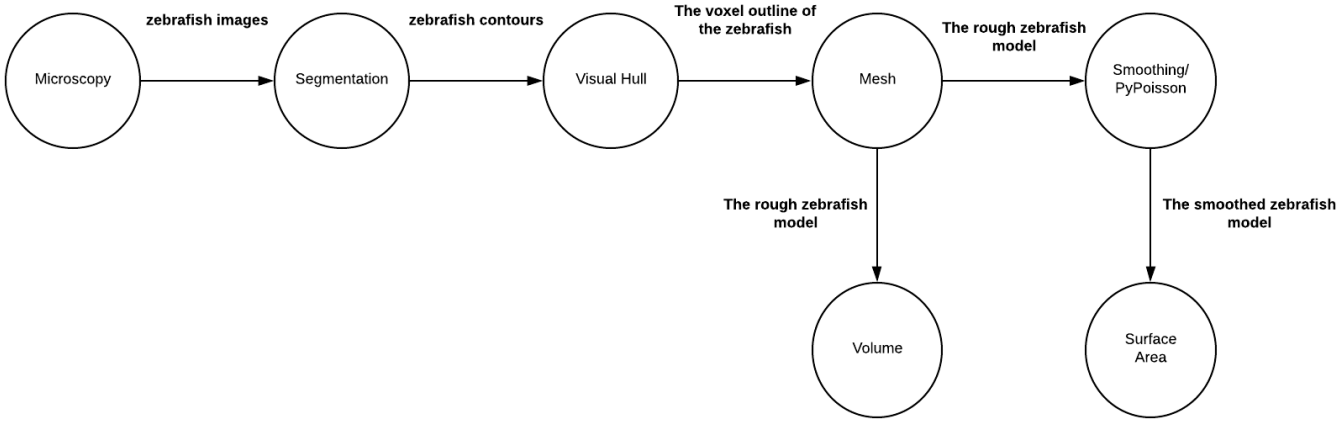

Figure 3.2: A rough flowchart of the order of operation of the code.

3.2 Camera parameter optimization

The old segmentation code was originally made by Guo et al. [6] and was meant to make the already fairly hard to interpret images readable for the computer by turning every zebrafish image made by the microscope, into a black and white contour image. This is also referred to as the Matlab segmentation method. This was originally done by using two sets of segmentation, one for the body and one specifically for the tail, and combining the result. The segmentation that focuses on the tail is to mitigate the extremely transparent nature of the tail, allowing for a better estimation of where the outline of the tail exactly is.
The other segmentation made by Wilco Verhoef and Wilco de Boer is a Neural Network for the segmentation of the zebrafish [14]. This has the advantage of being more flexible compared to the Matlab implementation which requires some amount of preparation of the zebrafish images while the Neural Network can turn any zebrafish image into a contour as long as the network has been trained.

The conversion from Matlab to Python for the 3D reconstruction and the conversion from Matlab to a Neural Network for the segmentation means that there is now no direct link between the Neural Network and the Python code. 3D Reconstruction has a large dependency on segmentation and requires the segmentation to be linked to the 3D reconstruction. This is because the 3D reconstruction requires the contours from the segmentation to create the 3D model, the volume and the surface area. This means we have to reestablish the connection between segmentation and 3D reconstruction, which will be referred to as the pipeline. This connection will be discussed in Section 3.4.

3.3 Reconstruction

After the segmentation is ready and the camera parameters have been calculated and optimized, the 3D model of the zebrafish can be reconstructed. By taking the contours and the parameters, the algorithm can construct a series of voxels in a 3D space. This is done by an algorithm that looks at all the contours and decides where the outline of the zebrafish is in the 3D space. Determining where certain voxels are and how to best make a 3D model out of these voxels is a very resource-intensive process.

The algorithm we use that reconstructs a 3D model from the segmented images we get from the VAST BioImager and the camera parameters as calculated by the camera parameters calculations was converted in the previous work from Matlab to Python by Juliëtte Meeuwsen [9]. We use Python 2.7.12 for the program because Python is free and easy to use, compared to the Matlab programming language, which requires a paid license to be used.
The Python implementation of this algorithm requires several important packages to function. The most important ones are:

- Numpy: Used for a wide variety of mathematical functions.
- Opencv: Used in segmentation for image recognition.
- PyPoisson: Used for the Poisson reconstruction and the smoothing of the 3D zebrafish model.
- SciPy: Contains scientific functions that were native in Matlab but not native in Python.
- Matplotlib: Used to replicate the functions from the Matlab conversion that are not natively in Python, specifically in 2D plotting.

From earlier work we know that there are opportunities for optimization of the reconstruction of the contours into the 3D model. These opportunities will be investigated in this thesis. The current reconstruction method is, as shown Figure 3.1, a multi-step process so there are multiple places that need to be looked at for optimization.

3.3.1 Execution time bottlenecks

The very first action that needs to be done in the optimization process is to establish the amount of time taken by each major function. We ran the code on one of the desktops in LIACS. This desktop had a 4 core Intel I7 processor running with 3.3 GHz. Every individual zebrafish was run on a single thread, where measurements were taken to see how much time certain functions take. The code is divided into five parts as was displayed in Figure 3.1. These parts are visual hull, mesh, volume, smoothing and surface area.

In Figure 3.3 there are three different runtimes for three different zebrafish: Zebrafish 1, Zebrafish 5 and Zebrafish 10. In this figure in particular we are looking for one or several parts of the code that take particularly long compared to the other parts of the code.
Out of the five points there were three parts that took a particularly long time: smoothing, volume calculations and the visual hull calculations, where the visual hull is particularly long as seen in Figure 3.3.

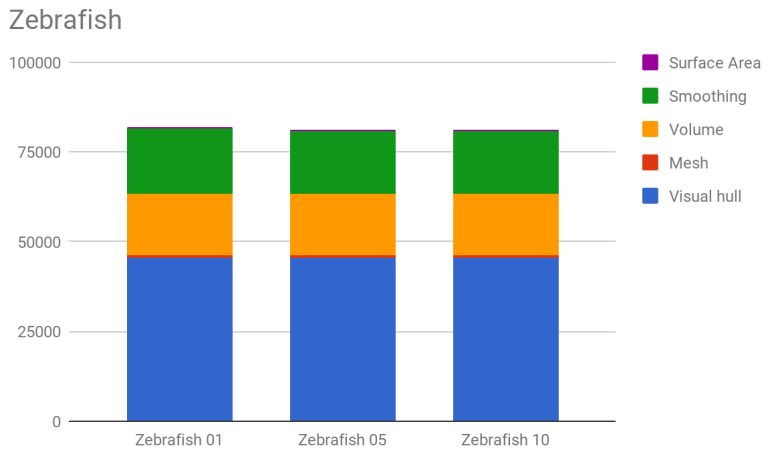

Figure 3.3: The time it takes to process three zebrafish in milliseconds.

We can now examine the different aspects of the code.

Mesh and surface area

The code concerning the mesh is a simple function that takes the outline of voxels created by the visual hull and creates a rough 3D model in preparation for the smoothing function and the volume calculations. The surface area is calculated by taking the faces and vertices that are calculated by the PyPoisson library [10] with which we can calculate the surface area.
These computations cost only around 600 milliseconds for the mesh and about 200 to 300 milliseconds for the surface area, as seen in Figure 3.3. This means that early on the decision was made that it was not worth looking at them for optimization.

Smoothing

Smoothing is the part of the code that takes the 3D model created in the mesh function and, like the function name suggests, smooths the model. Smoothing a model is used to increase the accuracy of the model and make a more real to life model. For the smoothing we use a library called PyPoisson [10]. This library is made up of a small Python wrapper around a library of C++ code [7] that implements the Poisson reconstruction called PoissonRecon.
Poisson reconstruction is a mathematical set of functions for surface reconstruction. Poisson reconstruction uses a set of points to recreate a smooth surface. In the case of this library this set of points is in a 3D space.
The decision was made not to optimize this part because the runtime is dictated by the underlying C++library, PoissonRecon [7], and trying to optimize the library for this code was outside of the intended scope.

Volume calculations

The volume calculations are done by adding all the voxels together. Over 99% of the time spent in volume calculations was done by combining the values of the voxel chunks. The combining of the values was done inefficiently because of the use of the summation function. Using the Numpy library variant of this function reduced the runtime dramatically as seen in the results. These improvements will be discussed further in Chapter 4.

Visual hull calculation

The visual hull calculation is the most time consuming process in the entire program. An average program run before the optimization was between half and two thirds of the program runtime as we will explore in the results. Most of the runtime in calculating the visual hull is done by extracting an index from a large array of voxels and finding the value of 1 or 255 in the array amongst those indexes which will be discussed further in Chapter 4. The voxels with an index value of 1 and 255 are voxels that are part of the zebrafish, the other voxels which have a value of 0 are disregarded as background. All these voxels form a cloud together in the rough shape of a zebrafish.

3.3.2 Method

The main way the performance was improved was through the removal of dead code and applying the Numpy library.
For the volume calculations there was a single use of the standard sum function of Python that took around 17 seconds to complete. Using the sum function from the Numpy library made that number much lower, with the volume calculations taking around 300 milliseconds instead of the previous 17000 to 18000 milliseconds.
In the case of the visual hull calculations there were several functions where parts did functionally nothing but still took runtime to execute. These were removed and the runtime was lowered while the output stayed exactly the same. Further in the visual hull code the index needed to be determined. The way the index was calculated was not optimal for the Python implementation. There were several extra steps that were taken that could be done in one line of code in Python. The results explained in Chapter 3 will be further discussed in Chapter 4.

3.4 Implementation

In this section we will discuss how the 3D reconstruction and the camera parameter calculations are implemented in the code.

3.4.1 Camera parameter calculations

For the implementation of the pipeline after segmentation it is necessary to calculate the camera parameters for every contour from the segmentation. This is done so the 3D reconstruction is able to properly read and process the contours from the segmentation into a 3D model. The camera parameters let the algorithm know from which angle the contour was taken.
This is necessary for the 3D reconstruction because several parts in the algorithm need to know from where every contour was taken relative to every other contour. This is done by calculating the offsets of the x axis, y axis and z axis with which we can calculate the necessary variables for the 3D reconstruction. In the process of converting the reconstruction code from Matlab, the functions that calculate these parameters were not converted to Python. In this case there are several of these parameters that we need to calculate for the 3D reconstruction.

We need to calculate several variables using the focal length and three variables named α, γ and ϕ. They are calculated in the first part of the camera parameter calculations. To calculate α, γ and ϕ there is need for an optimization algorithm. The optimization method that was used is named the Nelder-Mead method [5].

The variables that are needed for the 3D reconstruction are called, K, R, P and T. These variables are calculated using a set of matrix calculations as defined by Guo et al. [6] using α, γ, ϕ and the focal length.

In the camera parameter calculations the focal length is represented as the variable K which has a value of 65000 nanometer. α, γ and ϕ are used to create the rotations of the y and z axis in the form of a three by three array as seen in the graphs below. The rotation of the x axis is calculated differently for each zebrafish contour. Instead of taking the α or γ value like we do for the Y and Z rotation we take the angle compared to the other fish. The first angle is zero, the last angle is 2π, and every angle in between has a value in between, for example if there are 100 views for a fish the fiftieth view has a value of π.

The calculations for the Z rotation can be seen in Equation 3.1, the calculation for the Y rotation in Equation 3.2 and the calculation for the X rotation in Equation 3.3. These three values are multiplied by each other to form $R = Z * Y * X$.

$$Z = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Equation 3.1: Z rotation calculation.

$$Y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$

Equation 3.2: Y rotation calculation.

$$X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(angle) & -\sin(angle) \\ 0 & \sin(angle) & \cos(angle) \end{pmatrix}$$

Equation 3.3: X rotation calculation.

The last two values P and T are made with the variable R, which we just calculated, and a variable which we name t. The variable t is calculated as seen in Equation 3.4. To make P we insert t behind the matrix R which ends in a 3 by 4 matrix. The calculation for T is $T = t * R$.

$$t = \begin{pmatrix} 0 \\ \sin(\phi) \\ \cos(\phi) \end{pmatrix}$$

Equation 3.4: Calculation of t.

In the end of the calculation we pass along the four variables together with the contours to the 3D reconstruction where they are used to calculate the volume and surface area.

The parameter calculations were originally made in Matlab by Guo et al. [6] so it was an easy task to convert the matrix calculations from Matlab to Python. The biggest hurdle was the part that preceded it, the Nelder-Mead algorithm [5] for the optimization of α, γ and ϕ.
The second big hurdle is that the format the segmented images are saved in, is different than in the previous Matlab implementation. This means the 3D reconstruction code, where the segmented image is needed, needs to be transformed to accommodate the new format of the segmented image.

3.4.2 The JSON structure

In this thesis we have, on multiple occasions, described processes earlier in the pipeline and the information, like the variables from the camera parameter calculations, we need from these processes for the 3D reconstruction. Each section in the pipeline needs something from one of the previous sections and also has to store multiple parameters for later use.

We need to make sure each section of the pipeline can obtain the information said section needs in a dynamic and efficient manner. We also need to make sure that each section can store information for later use. To implement this we have decided to use a JSON structure to use as a storage solution.

There are three steps that add information to the JSON structure that we will discuss. These steps will be centered on what the JSON structure brings to the 3D reconstruction. The 3D reconstruction can also add information to the JSON structure after its execution, but this will not be discussed.

We will also have to discuss which information enters the JSON structure when and why. This can change in the future according to the needs of the user. We will also provide an example of the JSON structure in every point of the pipeline.
In Figure 3.4 the information that will be put into the JSON structure is shown.

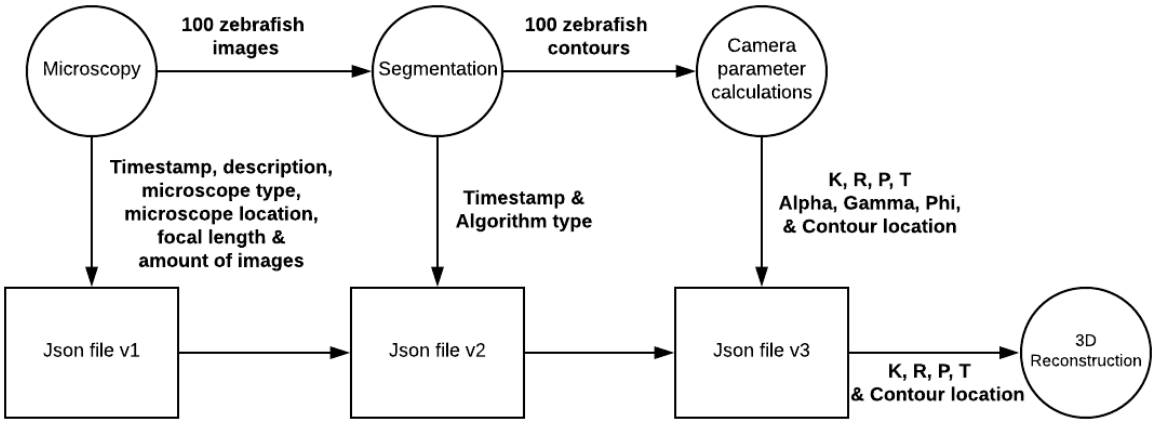

Figure 3.4: Flowchart detailing the information flow of the JSON structure.

We will now discuss an example of the JSON structure with specified parameters from Section 3.4.1.

Microscopy

The first step which puts information into the JSON structure is the microscopy.
For the microscopy most of the information that will be stored into the JSON structure is for tracking what the microscope has done, when the microscope was used and information about the microscope itself.

As can be seen in Figure 3.4, several identifiers about the microscope will be listed, such as the microscope type and microscope location to link image sets to a particular microscope, and any parameters the microscope used. This information is stored in the JSON structure, so any errors in any part of the pipeline can be tracked all the way to the microscope if necessary. The timestamp of when the microscope was used will also help with this. Lastly we will record the amount of contours made by the microscope. If we want to change the amount of contours used in the pipeline we do not need to change a number in every step of the pipeline.

An example of a JSON structure at the end of the microscopy will look like Figure 3.5.
The microscopy timeMin and timeDMY show the exact time and date the microscopy was executed. The variable timeMin is notated using the notation for the 24-hour clock while the timeDMY is written in the DMY format and is written down as ”DD-MM-YYYY”.

The microType is the type of microscope used by the user so we know what microscope we use and the microLocation is the name of the institute or the city the microscope was in at the time of the creation of this file. The description indicates what the zebrafish is to be used for. The parameterFocal is an example of a parameter, in this case it is the focal length of the microscope. The focal length is in this case 65000 nanometer but can be different. The imageCount is how many images were generated by the microscope. In this case it is 100 images, and is left to the user who determines the best amount of images for the experiments they are running.

```
{

    "microscope": {

        "timeMin":"15:49",

        "timeDMY": "10-05-2019",

        "microType": "Upright Compound Microscope",

        "microLocation": "LIACS",

        "description": "Zebrafish STEM research",

        "parameterFocal": "65000",

        "imageCount": "100"

    }

}
```

Figure 3.5: JSON structure example after the microscopy was executed.

Segmentation

In the segmentation part of the pipeline the timestamp will also be stored in the JSON structure to track when the segmentation was executed. We also need to store the type of algorithm that was used in the segmentation step. At the moment of this thesis we are using the Neural Network implementation of Wilco Verhoef and Wilco de Boer [14]. Other implementations can also be developed for segmentation in the future. If we store the algorithm type in the JSON structure it is easier to track at the end of the pipeline which algorithm was used. An example of a JSON structure at the end of the segmentation will look like Figure 3.6.
The segmentation timeMin and timeDMY show the exact time and date the segmentation was executed, just like with the microscopy. The algorithmType displays what algorithm was used to generate the contours; in this case the Neural Network was used.

```
{
    "microscope": {
        "timeMin":"15:49",
        "timeDMY": "10-05-2019",
        "microType": "Upright Compound Microscope",
        "microLocation": "LIACS",
        "description": "Zebrafish STEM research",
        "parameterFocal": "65000",
        "imageCount": "100"
    },
    "segmentation":{
        "timeMin":"17:24",
        "timeDMY": "12-05-2019",
        "algorithmType": "Neural Network"
    }
}
```


Figure 3.6: JSON structure example after the segmentation was executed.

Camera parameter calculations

In the last step of the pipeline before we go to the 3D reconstruction more information needs to be stored before the 3D reconstruction can be executed. The 3D reconstruction needs the variables α γ ϕ. The 3D reconstruction also needs the location of the contours so it can use the contours to reconstruct the 3D model. The advantage of storing the location of the contours in the JSON structure means that several individuals can run the 3D reconstruction algorithm at the same time and so we know which contours the parameters calculated by the camera parameter calculations relate to. There is also no need to adapt the location every execution of the 3D reconstruction algorithm.

An example of a JSON structure at the end of the camera parameter calculations will look like Figure 3.7. For the camera parameter calculations the variables calculated by said function are put into the JSON structure. The parameter K is 65000 nanometer, just like the parameterFocal defined by the microscope. The variables α, γ and ϕ are optimized using the Nelder-Mead algorithm during the camera parameter calculations. The contourLocation is where the contours are at the end of the segmentation; in this case this is in the folder "/home/s1518445/npytemp".

```
{
    "microscope": {
        "timeMin":"15:49",
        "timeDMY": "10-05-2019",
        "microType": "Upright Compound Microscope",
        "microLocation": "LIACS",
        "description": "Zebrafish STEM research",
        "parameterFocal": "65000",
        "imageCount": "100"
    },
    "segmentation":{
        "timeMin":"17:24",
        "timeDMY": "12-05-2019",
        "algorithmType": "Neural Network"
    },
    "camParamCalc": {
        "K": "65000",
        "alpha": 1.7e-11,
        "gamma": 0.5e-6,
        "phi": 0.1e-14,
        "contourLocation": "/home/s1518445/npytemp"
    }
}
```


Figure 3.7: JSON structure example after the camera parameter calculations were executed.

As an extra clarification Table 3.1 is provided to clarify which of the variables in the example are required for the execution of the algorithms and functions in the pipeline and which variables just provide extra information. It also provides which unit the variable is supposed to be in. The examples in Figure 3.5, 3.6 and 3.7 contain values for the variables which are accurate representations of a real life example.

Variable	Unit of measurement	Required
timeMin	24-hour clock format string	No
timeDMY	string in "DD-MM-YYYY" format	No
microType	string with microscope type	No
microLocation	string with the location of the microscope	No
description	string with additional information	No
parameterFocal	focal length in nanometer	Yes
imageCount	integer containing the image count	Yes
algorithmType	string containing which algorithm was used	No
K	focal length in nanometer	Yes
alpha	the rotation of the y axis in radian	Yes
gamma	the rotation of the z axis in radian	Yes
phi	the translation angle in radian	Yes
contourLocation	string containing the location of the contours	Yes

Table 3.1: The units and formats of every variable described in the JSON structure.

3.4.3 Implementation of the 3D reconstruction

Out of the five sections in the program there are only three bottlenecks that are relevant to the optimization of the reconstruction code. These three sections are the visual hull calculation, volume calculation and smoothing. The other two sections are mesh and surface area. These sections are not bottlenecks but necessary for the understanding of the other three sections so those sections will be briefly explained for context.

Visual hull calculation

In Figure 3.8 a visual hull calculation of two contours of a man is depicted. One contour is projected from the point at the bottom of the picture and one contour is projected from the point on the left of the picture. Both contours are projected into a 3D space. The visual hull calculation for the zebrafish uses the camera parameters to determine where these points are and how they project the contours into the 3D space. The intersection of the projections creates a rough shape of the 3D model. Figure 3.9 shows the intersection for the example of the contours of the man from Figure 3.8. The execution of the visual hull calculation for the zebrafish looks similar to Figure 3.8 and Figure 3.9. Instead of the man-shaped contours and instead of two points from which we project these contours we use the zebrafish contours and we project with 84 contours or 100 contours from an equal amount of points. These contours make the zebrafish model a lot more accurate when compared to the 2 contour man-shaped model.

The visual hull calculation of the zebrafish in our thesis takes the segmented images and the variables from the camera parameter calculations and creates the visual hull out of these images and variables. The creation of the visual hull is essential for the creation of a 3D model. A visual hull is created by taking the set of contours that was generated by the segmentation. These contours are taken from the multiple points of view. These points of view are calculated by the camera parameter calculations for every contour, so that the visual hull calculation has access to which contour was taken from which point in a 3D space. If we take every contour from every point of view and project the contours, the visual hull calculation can create a solid voxel model in a 3D space, it can distinguish which of the voxels are part of the zebrafish and which voxels are part of the background and uses the outer voxels in the point cloud. This means a solid voxel model in a 3D space is created that resembles a zebrafish.

Mesh

The mesh uses the cloud of points generated by the visual hull calculations to create a rough mesh. The initial mesh is generated using the Marching Cubes Lewiner algorithm [3] on the points that were generated by the visual hull calculations. From this we can generate a set of vertices, faces, normals and values. These four variables allow us to define the mesh of the zebrafish shape.

Volume calculation

In the mesh and visual hull calculation we determine which voxels belong to the zebrafish and which voxels do not. Every voxel that belongs to the zebrafish has a certain value, every voxel that does not belong to the zebrafish has no value. By adding all the voxels together we can determine the volume.

Smoothing

The smoothing, performed by PyPoisson and PoissonRecon earlier in the thesis, takes the rough 3D model generated by the mesh function and runs the rough model through the PyPoisson library. PyPoisson uses the C++ library PoissonRecon which implements the Poisson reconstruction, this library was written by Michael Khazdan [7]. PoissonRecon has been wrapped in Python by Miguel Molero-Armenta [10] to make the PyPoisson library.
Poisson reconstruction uses the Poisson equation formulation of the surface reconstruction. The reconstruction algorithm requires the vertices and the normals of the zebrafish as generated by the mesh function to return a set of new faces and new vertices to represent a new surface for the new smoothed model.

Surface area

The surface area takes the new vertices and new faces as generated by the Poisson reconstruction. With these new vertices and new faces we can easily generate the surface area of the zebrafish by integrating over the faces.

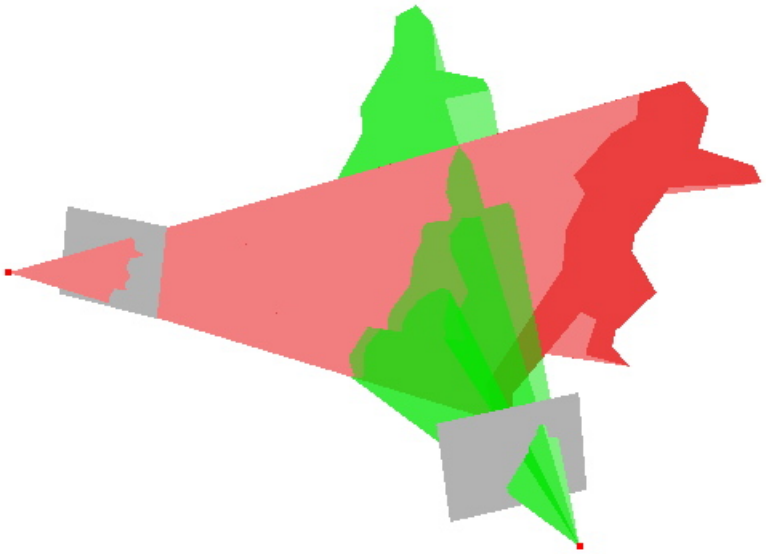

Figure 3.8: Pictures taken from two perspectives of a model.

Source: https://upload.wikimedia.org/wikipedia/en/2/2f/SilhouetteCones.jpg.

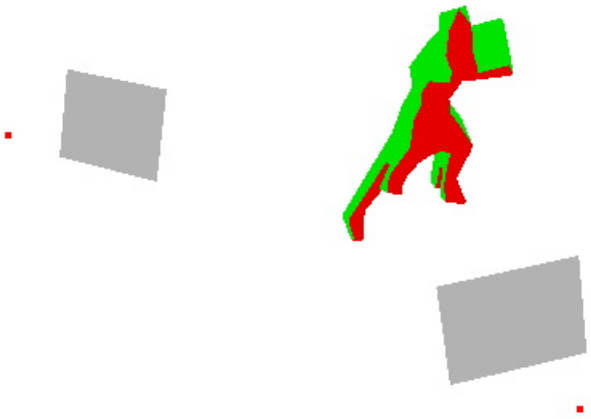

Figure 3.9: The visual hull created from the intersection of the two perspectives.

Source: https://upload.wikimedia.org/wikipedia/en/2/2f/SilhouetteCones.jpg.

Chapter 4

Experiments and Results

In this chapter we investigate the effectiveness of the implemented optimizations when compared to the older runtimes. To confirm the effectiveness, several experiments have been executed on the LLSC cluster to compare the results of the new optimized code to that of the older code. All the experiments list the time it takes to execute 5 areas: visual hull, mesh, volume, smoothing and surface area. All tests list these 5 values in the preoptimization and postoptimization which represents the runtime of the code before and after the optimization respectively.

The first of these experiments is to look at the effectiveness of the improvements. To measure this there will be a set of experiments run on the LLSC. The experiments were run on the LLSC on the following node type: cpu-e5430, two Intel Xeon e5430 processors, 2.66 GHz, 4 cores per processor.
All the data in every table and figure has been run three times to check for outliers.

There are broadly speaking two sets of experiments, the experiment on the dataset of 84 contours (dataset 1) and the experiment on the dataset of 100 contours (dataset 2).
Table 4.1 contains which figure uses which dataset and Table 4.2 contains which table uses which dataset.

Figure	Fig.3.3	Fig.4.1	Fig.4.2	Fig.4.3	Fig.4.4	Fig.4.5	Fig.4.6	Fig.4.9	Fig.4.10
Dataset	1	1	1	1	1	1	1	2	2

Table 4.1: The notation of which figure uses which dataset during the results.

Table	Table 4.3	Table 4.4	Table 4.5	Table 4.6	Table 4.7	Table 4.8	Table 4.9	Table 4.10
Dataset	1	1	1	1	1	1	2	2

Table 4.2: The notation of which table uses which dataset during the results.

There will also be reference to “old” and “new” code. The old code is a reference to the code as delivered by Juliëtte Meeuwsen [9] while the new code is the code optimized in this thesis.

For the 84 contours experiments, the first experiment is to check if the new, optimized code has changed the volume and surface area of the zebrafish. This is done to test the correctness of the new, optimized code. After this test we will run a single zebrafish on a single node on both the old and the new code. These experiments will be done on one, two and four cores to show the difference between the old and new code, and shows how well both the old and new code scale. The second set of experiments will be done on a dataset of 20 zebrafish on four cores to check how well the performance is on larger datasets. We will also test how well the 84 contours dataset performs with the new code while using multiple nodes, with multiple zebrafish executing parallel to eachother.

For the 100 contours experiments there will be two experiments, one to test the correctness of the 100 contours results and one that tests the time it takes for a zebrafish to complete on four cores. The performance tests for the 100 contours will split the test into the code for the 3D reconstruction and the camera parameter calculations to see the extra runtime added by the camera parameter calculations and to properly measure the impact of the extra 16 contours.

4.1 Zebrafish with 84 contours

In this section we will discuss the correctness and the runtime of the dataset of zebrafish that consists of 84 contours.

4.1.1 Correctness of the results

Evidence has to be delivered that the optimizations in the new code have not influenced the results. To test if the new code gives a similar and correct result, the surface area and the volume of the zebrafish have to be compared to the results of the old version of the code. To confirm this, we will take several zebrafish that were measured in the old code and compare these to the results generated by the new code.

The first test we run is to see if the new code delivers the same results from the same zebrafish as the old code, as shown in Table 4.3 and Table 4.4. This test is done on 4 zebrafish selected from a dataset of 20 zebrafish.

Zebrafish	84 contours old	84 contours new
1	253863933	253863933
5	237735689	237735689
10	274142488	274142488
20	308396300	308396300

Table 4.3: Volume comparison of the zebrafish in cubic nanometer.

Zebrafish	84 contours old	84 contours new
1	3473968	3473968
5	3202020	3202020
10	3653512	3653512
20	4001473	4001473

Table 4.4: Surface area comparison of the zebrafish in square nanometer.

Table 4.1 shows that the volume of Zebrafish 1, Zebrafish 5, Zebrafish 10 and Zebrafish 20 are exactly the same. In Table 4.2 we can see likewise that the surface area of Zebrafish 1, Zebrafish 5, Zebrafish 10 and Zebrafish 20 are also exactly the same.

4.1.2 Runtime improvements of the results

After establishing that the results are exactly the same we can now discuss the runtime improvements that were made by the new code.

Small-scale testing on one node

To test the improvements of the runtime in the results we shall run several tests. The first tests are the execution of a single zebrafish, Zebrafish 1, on one core, two cores and four cores. This is done to see how the code scales. The figures in this chapter contain the terms pre and post. The term pre is to denote the runtime of the code before the code was optimized, the term post is to signify the runtime of the code after the optimization.

The execution of Zebrafish 1 on one core is seen in Figure 4.1, the execution of Zebrafish 1 on two cores is seen in Figure 4.2 and the execution of Zebrafish 1 on four cores is seen in Figure 4.3. These figures give an overview of the impact of the optimizations done in this thesis.

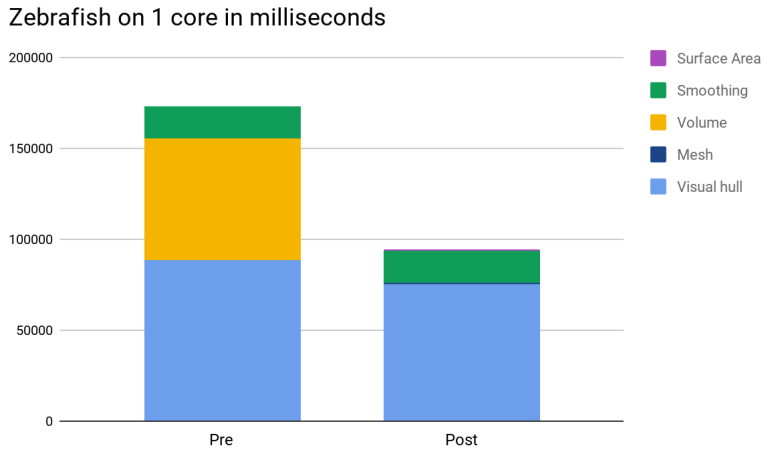

Figure 4.1: Single core performance of Zebrafish 1 in milliseconds.

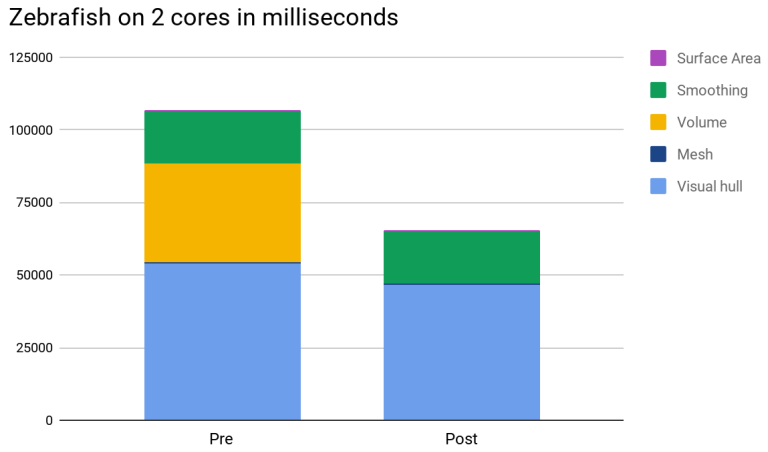

Figure 4.2: Two core performance of Zebrafish 1 in milliseconds.

Table 4.5 is the numerical representation of Figures 4.1, 4.2 and 4.3. In this table the exact numbers of the performance and runtime for Zebrafish 1 over one, two and four cores is represented. The tables are there to document the exact numbers used in the figures. The figures are there to visualise the comparison of the different runtimes. The total runtime displayed at the end is divided into 5 categories. These categories are visual hull, mesh, volume, smoothing and surface area.

	Visual hull	Mesh	Volume	Smoothing	Surface area	Total runtime
1 core preoptimization	88274	605	66419	17837	284	173419
1 core postoptimization	75295	603	145	17988	284	94315
2 core preoptimization	53912	604	34113	17921	284	106834
2 core postoptimization	46397	604	169	17926	297	65393
4 core preoptimization	46198	605	16946	17899	283	81931
4 core postoptimization	41217	606	235	18597	285	60940

Table 4.5: Runtime of Zebrafish 1 in milliseconds over multiple sets of cores.

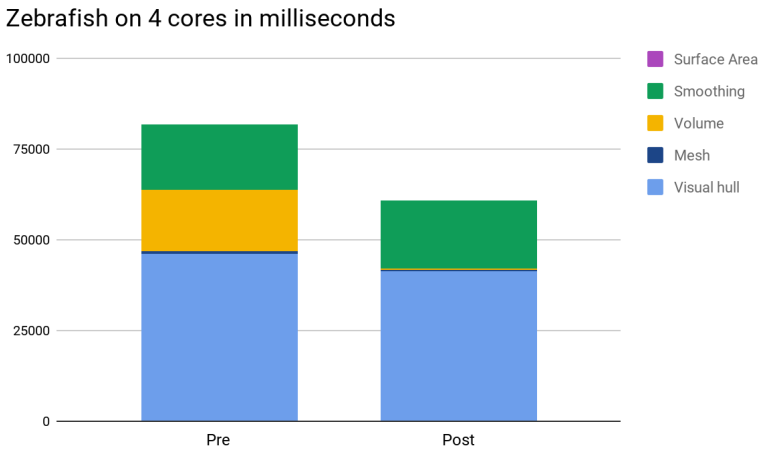

Figure 4.3: Four core performance of Zebrafish 1 in milliseconds.

We can see a significant increase in performance from the tests with a lower amount of cores. In Figure 4.1 we see an 83% increase in performance. In Table 4.5 as well as in Figure 4.2 and Figure 4.3 we can see that the relative gain goes down the more cores are added to the process. In Figure 4.2 there is a 63% increase in performance while in Figure 4.3 there is only a 34% increase.

This means that the single core performance has increased more in comparison to the multicore performance. In the old code going from a single core to two cores, the performance increased by 62% while in the new code the performance increase is only 44%. Going from two cores to four cores, the performance has gone from a 30% increase in the old code to only a 7% performance increase in the new code.

The only area where there is a performance increase in the new code due to adding more cores is in the visual hull calculation. In the old code there is also the volume calculation which gets substantially faster, doubling the amount of cores halves the runtime.

Large-scale testing on one node

The second set of tests is the execution of a larger set of zebrafish on four cores. This is done to see how the code works on large scale tests.

The terms Pre1, Post1, Pre5, Post5, Pre10 and Post10 signify two things for each term. Pre means that the zebrafish is run on the unoptimized code, post means that the zebrafish is run on the optimized code. The numbers after Pre and Post, 1, 5 and 10, signify which zebrafish were selected for the test. These three zebrafish were picked out of a set of 20 zebrafish that was run as one batch.
When comparing the results on four cores there is a difference in two main areas as can be seen in Figure 4.4. The time it takes to do the volume calculations has been reduced drastically. Where previously it took around 17000 milliseconds to complete the volume calculations, it currently takes between 200 and 400 milliseconds to complete the volume calculations. The other improvement is on the visual hull calculations where there is a decreased runtime of around 45000 to between 38000 and 40000 milliseconds, as can be seen in Table 4.6.

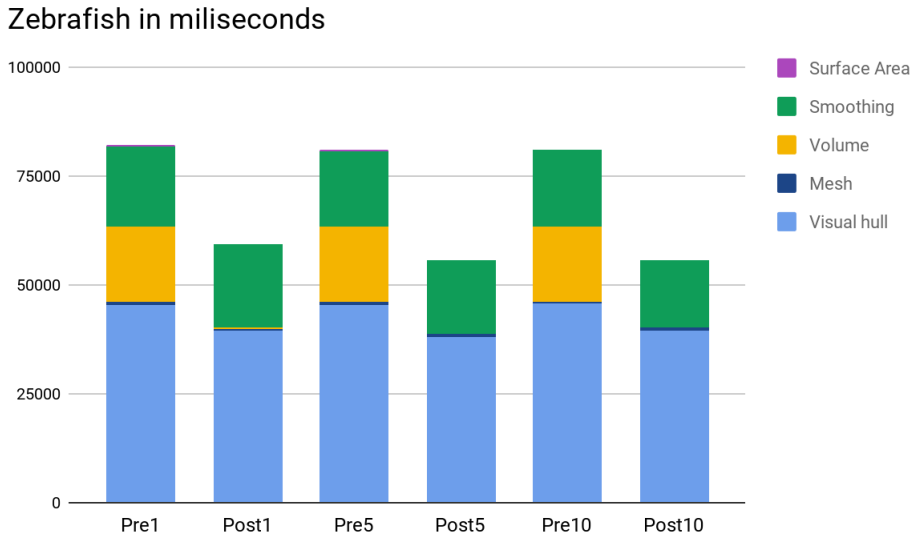

Figure 4.4: Comparison of post- and pre optimization runtimes of zebrafish in milliseconds on four cores.

	Visual hull	Mesh	Volume	Smoothing	Surface area	Total runtime
Zebrafish 1 preoptimization	45524	593	17149	18461	304	82031
Zebrafish 1 postoptimization	39509	514	255	18969	259	59506
Zebrafish 5 preoptimization	45515	589	17210	18461	255	81058
Zebrafish 5 postoptimization	38108	523	287	18969	101	55592
Zebrafish 10 preoptimization	45610	570	17210	17489	288	81167
Zebrafish 10 postoptimization	39530	531	328	15117	278	55784

Table 4.6: Runtime of Zebrafish 1, 5 and 10 in milliseconds on four cores.

Large-scale testing on multiple nodes

To see how the improvements made scale up to a higher node and core count, several tests were performed. The tests are a single run of 20 zebrafish. Each individual zebrafish is run on four cores. This means that for the tests with eight cores and two nodes, two fish are executed in parallel. For 16 cores and four nodes, four zebrafish are run in parallel.
The following results from those tests are a sample of four zebrafish with their runtimes broken up into five parts. The tests were also run in the previous unmodified code to compare and see how large the decrease in total runtime is.

	Visual hull	Mesh	Volume	Smoothing	Surface area	Total runtime
Zebrafish 1 preoptimization	42340	596	16646	16993	259	76834
Zebrafish 1 postoptimization	37666	569	239	16433	260	55167
Zebrafish 5 preoptimization	42899	567	16820	16204	285	76755
Zebrafish 5 postoptimization	41425	590	311	17780	263	60369
Zebrafish 10 preoptimization	42687	543	16849	16820	266	77165
Zebrafish 10 postoptimization	41174	629	259	16989	282	59333
Zebrafish 20 preoptimization	45060	590	17006	16838	251	79494
Zebrafish 20 postoptimization	41011	612	228	18110	280	60241

Table 4.7: Runtimes of zebrafish on 8 cores and 2 nodes in milliseconds.

	Visual hull	Mesh	Volume	Smoothing	Surface area	Total runtime
Zebrafish 1 preoptimization	45822	602	16796	17765	301	81286
Zebrafish 1 postoptimization	41105	606	224	17994	278	60207
Zebrafish 5 preoptimization	46647	588	16608	16146	260	80249
Zebrafish 5 postoptimization	37617	522	264	14602	276	53281
Zebrafish 10 preoptimization	46366	599	16598	17805	306	81674
Zebrafish 10 postoptimization	37906	571	235	16545	252	55509
Zebrafish 20 preoptimization	46573	612	17149	18116	281	82731
Zebrafish 20 postoptimization	41105	606	224	17994	278	60207

Table 4.8: Runtimes of zebrafish on 16 cores and 4 nodes in milliseconds.

In the tests there are two noteworthy things. The first noticeable thing is that the total runtime of individual zebrafish does not vary greatly from fish to fish with an increase in nodes. The total runtime fluctuates between 55000 and 61000 milliseconds in the new code and between 75000 and 85000 milliseconds in the old code, as seen in Table 4.7 and Table 4.8. Figure 4.5 visualises the total runtime of four individual sample zebrafish from Table 4.6 (as seen measured in the previous section), Table 4.7 and Table 4.8.

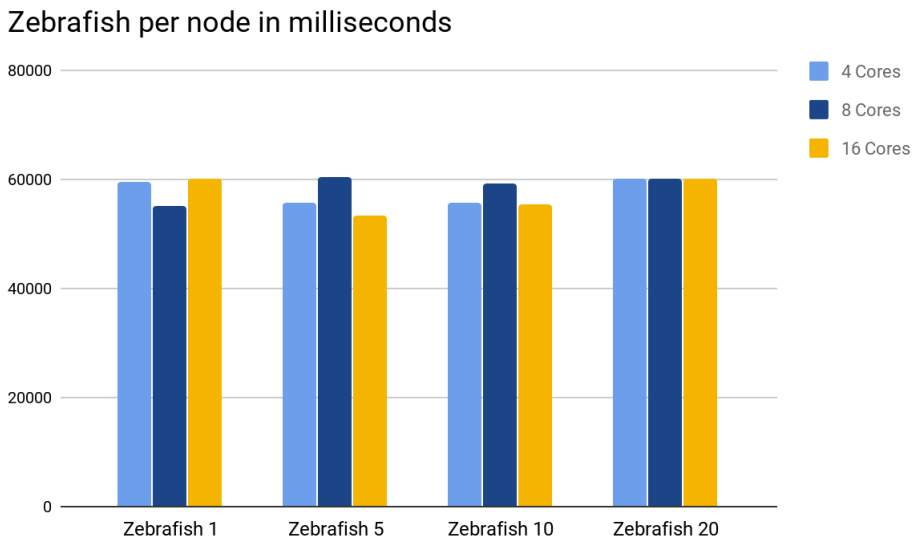

Figure 4.5: Comparison of the runtime for 4 fishes on different core counts in milliseconds.

The second noticeable thing is that the parallelisation does greatly reduce the total runtime of all zebrafish combined. To complete a set of 20 zebrafish takes much less time the more nodes are added. On four cores and one node it takes 532 seconds to complete the set of 20 zebrafish while in the old code it took 722 seconds. On eight cores and two nodes it takes 274 seconds to complete the set of 20 zebrafish while in the old code it took 368 seconds. On 16 cores and four nodes it takes 146 seconds to complete the set of 20 zebrafish while in the old code it took 186 seconds as depicted in Figure 4.6.

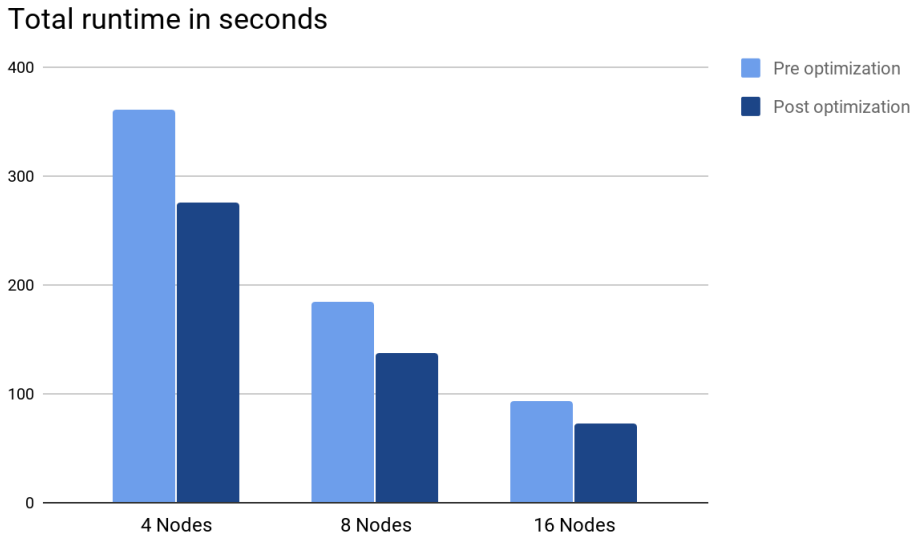

Figure 4.6: Comparison of runtimes of sets of 20 zebrafish on different amounts of cores in seconds.

4.2 Zebrafish with 100 contours

In this section we will discuss the correctness and the runtime of the dataset of zebrafish that consists of 100 contours.

4.2.1 Correctness of the results

All the previous zebrafish in this thesis used 84 contours to create the 3D model. The next set of tests is with 100 contours instead of 84. The next few tests are to measure if the 100 contours sets produce correct results from the camera parameter calculations and the 3D reconstruction code. The tests in this section, unlike in previous sections, are all exclusively executed on the new, optimized code.
We decided to use 100 contours instead of 84 contours because of increases in accuracy of the surface area calculations and volume calculations.

We will do this by running and measuring the surface area and volume of the 100 contours zebrafish in the new code and comparing the surface area and volume of these of 100 contour zebrafish to the probabilistic results generated by Guo et al. [6], as seen in Figure 4.7 and Figure 4.8. This is to determine if the results fall within acceptable parameters.
In Figure 4.7 and Figure 4.8 there are 3 lines: 3dpf, 4dpf and 5dpf. Each line represents a set of zebrafish belonging to a certain age group, where 3dpf represents the youngest zebrafish and 5dpf represents the oldest zebrafish.
We compare the set of 100 contour zebrafish to this table instead of using the previous 84 contour zebrafish because there are no zebrafish with 84 contours to directly compare the set of 100 contour zebrafish to. There will also be a measurement of the runtime of these images and they will be compared to the runtime of the 84 contour zebrafish. This is to see if there is any impact on runtime because of the 16 extra contours that need to be processed.

In Table 4.9 the measurements of six zebrafish of the set with 100 contours are displayed. Zebrafish 1, 3, 5, 8, 15 and 25 are 100 contour zebrafish in the 3dpf set. The 3dpf set is a set of zebrafish that is used for this particular comparison. These numbers are the average of 3 runs of the zebrafish.

Zebrafish	Volume µm^3	Surface area mm^2
1	231514390	2941485
3	227409467	2953582
5	226581084	2912703
8	235087370	3137147
15	251783611	3212453
25	259021905	3321305

Table 4.9: Volume in cubic nanometer and surface area in square nanometer of the 100 contour zebrafish.

When comparing the results of Table 4.9 to the red 3dpf zebrafish line in Figure 4.7 and Figure 4.8 all of the results fall under the 3dpf curve. Zebrafish 3 and 5 are on the small side with a volume of 227409467 cubic nanometer and 226581084 cubic nanometer respectively of the volume curve, which is possibly due to their younger age, but all of the other results fall well within the curve of 3dpf for volume.
All of the zebrafish, even 3 and 5, are well within the curve of 3dpf for the surface area.

From this we can see that the results are correct enough to continue with the performance tests.

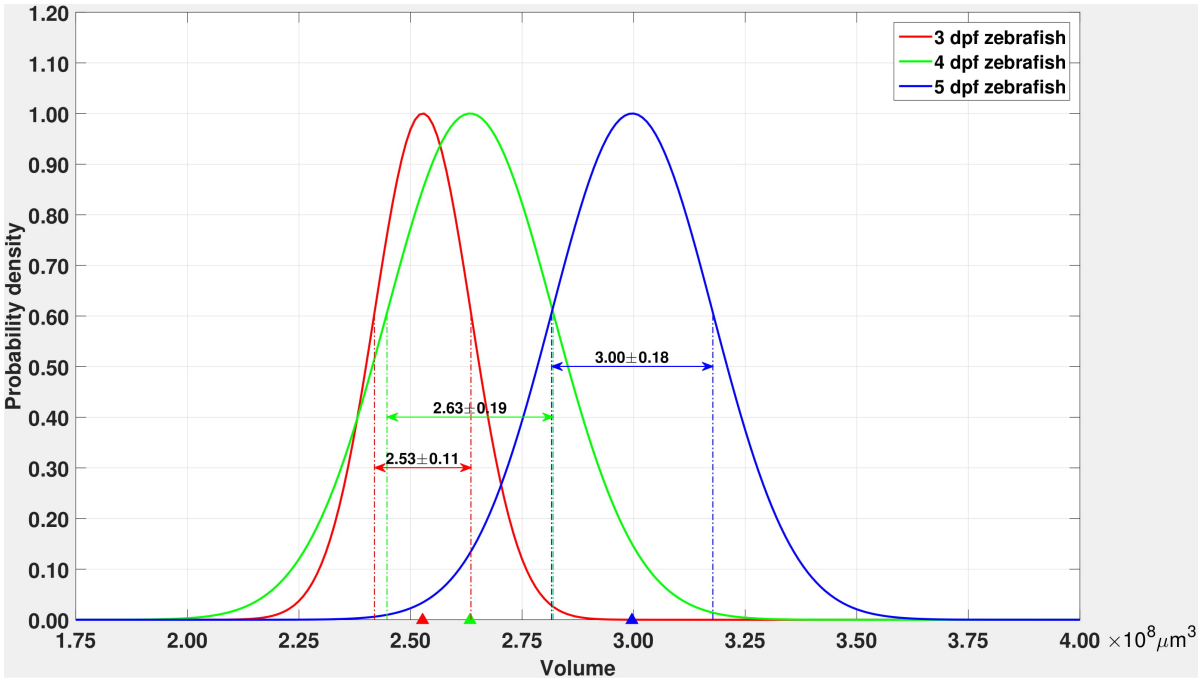

Figure 4.7: Graph on the probability distribution of volume. Source: [6]

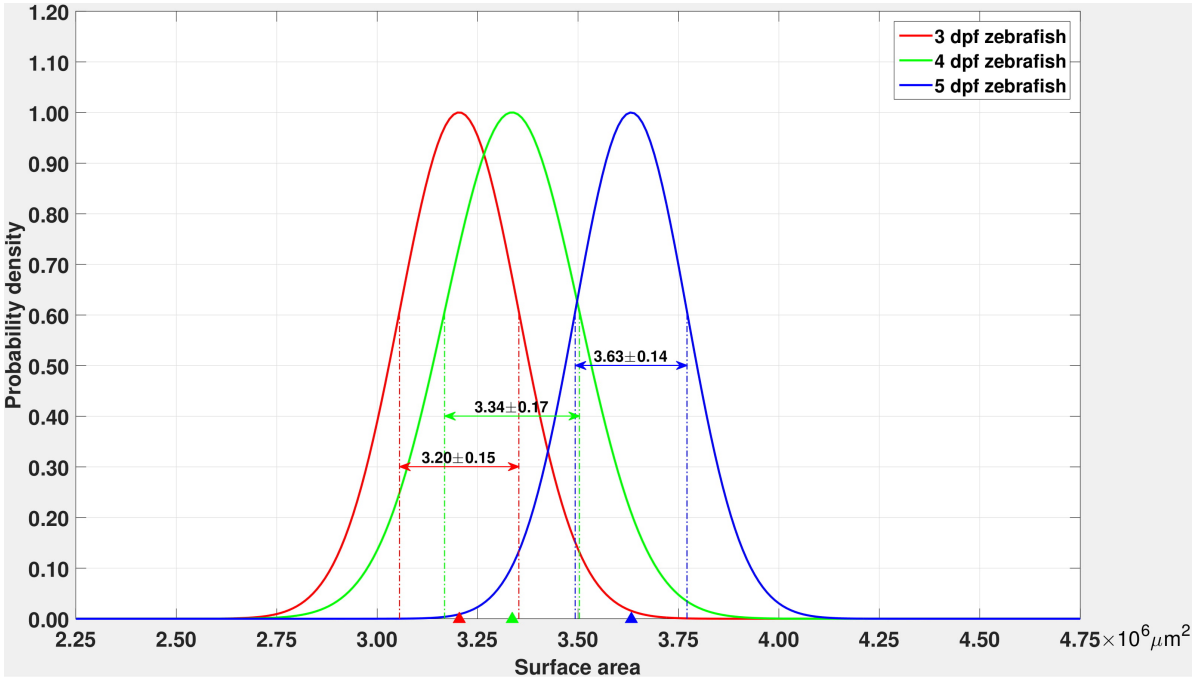

Figure 4.8: Graph on the probability distribution of surface area. Source: [6]

4.2.2 Runtime improvements of the results

To discuss the runtime impact of increasing the number of contours from 84 to 100, the camera parameter calculations have to be discussed first. The new 100 contour zebrafish code can only run with the new camera parameters. These new camera parameters have to be calculated separately using the new camera parameter calculations. The 84 contour zebrafish did not use the new camera parameter calculations to generate the variables. The 84 contour zebrafish used the older Matlab implementation which already included the variables with the contours and so does not need to calculate the variables separately using the new camera parameter optimization. So the difference has to be measured between the 84 contour implementation and the 100 contour implementation and the impact of the new camera parameter calculations on the runtime of the 100 contour implementation.

When looking at the new camera parameter calculations there are two parts that need to be looked at. The first part is to look at the new K, R, P and T variables and their potential impact on the runtime. In Section 4.2.1 we determined the validity of the variables K, R, P and T as generated by the new camera parameter calculations. This means these variables will not have an impact on the runtime of the 3D reconstruction.

The second part that we need to discuss are the camera parameter calculations that optimize α, γ and ϕ and calculate the variables K, R, P and T used in the reconstruction. These optimizations and calculations add much to the runtime. The increase in runtime is between 51 and 55 minutes for these camera parameter calculations, as seen in Figure 4.9.

For Zebrafish 1 the runtime for the variable calculation is just over 53 minutes, for Zebrafish 5 the runtime for the variable calculation is 52 minutes and 30 seconds, for Zebrafish 10 the runtime for the variable calculation is 54 minutes and 30 seconds and for Zebrafish 20 the runtime for the variable calculation is just over 52 minutes.

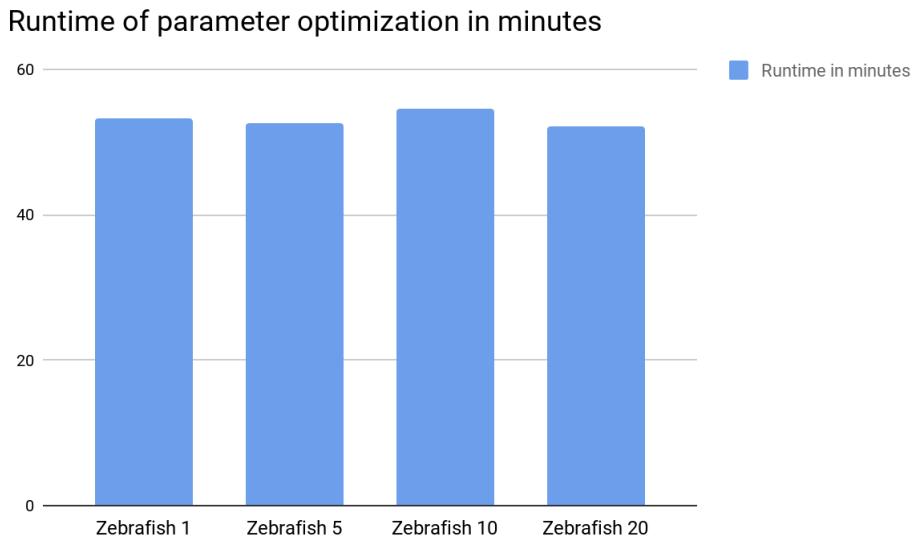

Figure 4.9: The runtime of the camera parameter optimization calculation in minutes.

Now that we have looked at the camera parameter calculations and determined the impact of those calculations we can now look at the runtime of the 100 contour zebrafish using the new variables. In Figure 4.10 we can see four zebrafish named Zebrafish 1, Zebrafish 5, Zebrafish 10 and Zebrafish 20. These are four zebrafish for which each reconstruction runs on on four cores.

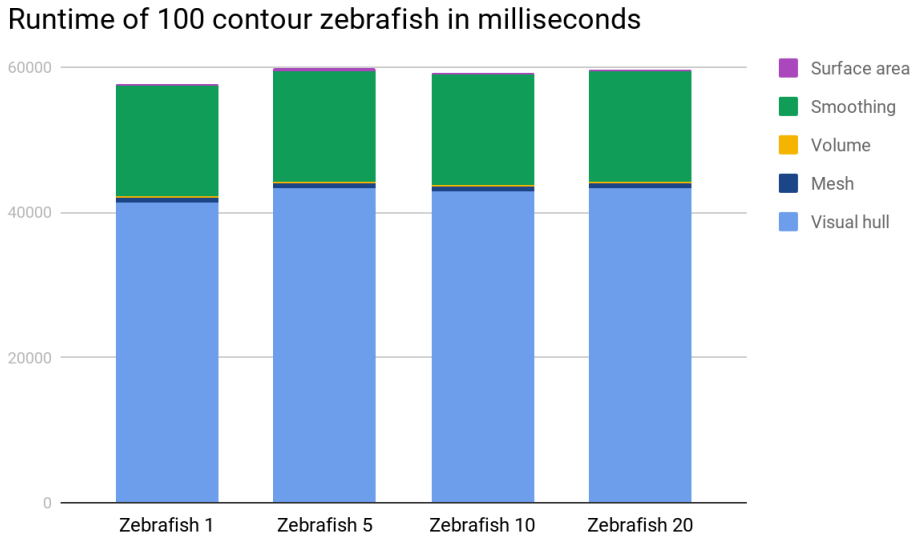

Figure 4.10: Runtime of the reconstruction of 100 contour zebrafish in milliseconds on four cores.

When looking at Figure 4.10 we can see that there is fairly little variance between each zebrafish and that the runtime is around 1 minute or 60000 milliseconds per zebrafish, as can be seen in Table 4.10. The 84 contour, post-optimization zebrafish on the other hand have a similar runtime of minute or 60000 milliseconds per zebrafish as can be seen in Table 4.6 and Figure 4.4.
When comparing the results of the 100 contour zebrafish in Table 4.10 and Figure 4.10 to the tests of the 84 contour post-optimization zebrafish as seen in Table 4.6 and Figure 4.4, it can be observed that the results are on average about as fast. The only noticeable thing is that the best-case scenario zebrafish of the 84 contour variant run faster by about 2000 milliseconds than the best-case scenario of the 100 contour zebrafish.

This means that the impact of the extra 16 contours is a slightly worse best-case scenario. This is surprising considering that the program theoretically has to do almost 20% more work while the runtime barely increases.

	Visual hull	Mesh	Volume	Smoothing	Surface area	Total runtime
Zebrafish 1	41387	597	207	15218	239	57648
Zebrafish 5	43361	614	202	15356	248	59781
Zebrafish 10	42861	594	238	15315	242	59250
Zebrafish 20	43372	592	208	15243	241	59656

Table 4.10: Runtime of Zebrafish 1, 5, 10 and 20 in milliseconds on four cores.

Chapter 5

Conclusions and Future work

The goals of this thesis were to look at the code that handled the images of the zebrafish taken by the VAST BioImager, optimize this code and link the separate functions of the code together. In this chapter we will discuss the completion of these goals and what can be further achieved on this subject.

5.1 Conclusions

In this thesis three goals were achieved. The first was the optimization of the 3D reconstruction code. The second was to recreate the camera parameter calculations and connect these calculations to the 3D reconstruction code and the segmentation. The last thing that was achieved was to make all of this code work for the new 100 contour zebrafish datasets as well as for the older 84 contour zebrafish dataset.

With the optimization of the code that covers the creation of a 3D reconstruction from the contours of the segmentation we reached a speedup of around 34% per zebrafish on four cores and an 83% increase in performance on a single core. This increase in performance is found in the speedup of individual zebrafish as well as in the decrease of total runtime of larger sets of zebrafish. This combined with the full computing power of the 24 nodes containing a total of 66 cores of the LIACS Leiden Science Cluster means that we can run up to 16 zebrafish at the same time using four cores per zebrafish. This allows us to utilise the full potential of the high-throughput imaging of the VAST BioImager.

In a move away from a Matlab implementation to a full Python implementation, the camera parameter calculations had to be recoded in Python. The new camera parameter calculations, which were converted to Python by Kristian Rietveld, can take the images of the contours from the new segmentation and convert them to the format used for the 3D reconstruction. This conversion to Python, combined with the JSON structure automating the flow of information between all the different parts of the pipeline, has automated most of this pipeline. This means that users that want to use the code for VAST BioImager need to get less involved with the process of creating the 3D model.

The last achievement was a conversion from 84 to 100 contours. This helped update the code for the creation of 3D models from the images taken by the VAST BioImager to an arguably more accurate model of the zebrafish. This increase in accuracy has come with an almost negligible increase in runtime, 3% in the most extreme case. This means that 100 contours as the new standard for measuring zebrafish in the near future is a good choice.

5.2 Future work

The results of the improvement in performance are substantial but there is room for further improvement. The visual hull calculations have some room for improvement and the use of the PyPoisson library [10] in the smoothing can potentially be improved. But PyPoisson, which uses PoissonRecon [7] which is written in C++, is already fairly fast and efficient at doing computations when compared to Python.

Using the full potential of the LLSC or another similarly large cluster with a larger set of the new 100 contour zebrafish to fully utilise the cluster. This could be done by running 16 zebrafish on four cores per zebrafish on the LLSC parallel to each other for multiple runs to see how the runtime would be impacted. Such a large test would shed light on how the code and cluster would behave under such demanding circumstances.

Another component that can be optimized further is the code that calculates the camera parameters for the 3D reconstruction. The current runtime for this computation is currently over 50 minutes compared to the 1 minute of the zebrafish itself. This means that there is a large incentive to look at this part of the code for optimization first.

As part of the code, volume and surface area are calculated. There are however more possible variables that can be calculated and that are needed for the research of the users of the VAST BioImager. An example of a variable that we would want to be able to calculate, would be the exact age of the zebrafish used in the VAST.

The segmentation component of the code is a project where there is an ongoing discussion about what the optimal number of contours is to make a 3D model with. There could be a better more efficient way of dealing with the segmentation or an optimal number of contours that has not been found yet.

The JSON structure has not been implemented yet over the entire pipeline. For the JSON structure to reach its full potential there needs to be tighter integration between the segmentation, the camera parameter calculations and the 3D reconstruction so that each part can deliver its information to the JSON structure for the user.

Chapter 6

Acknowledgements

I would like to thank Fons Verbeek and Kristian Rietveld for guiding my project, they were extremely helpful and enthousiastic in their guidance on the project. I would like to thank Yuanhao Guo for making the original Matlab implementation and Juliëtte Meeuwsen for converting that code to Python. At last I would like to thank Wilco Verhoef and Wilco de Boer for the segmentation code.

Bibliography

[1] https://www.adaptivecomputing.com/products/torque/, Last accessed 12-7-2019.

[2] https://slurm.schedmd.com/overview.html, Last accessed 12-7-2019.

[3] P. Borke. Polygonising a scalar field, May 1994. http://paulbourke.net/geometry/polygonise/.

[4] E. Burke. Why use zebrafish to study human diseases?, Oct 2016. U.S. Department of Health and Human Services.

[5] F. Gao and L. Han. Implementing the Nelder-Mead simplex algorithm with adaptive parameters, May 2010. Published by Computational Optimization and Applications.

[6] Y. Guo, J. W. Veneman, P. H. Spaink, and F. J. Verbeek. Three-dimensional reconstruction and measurements of zebrafish larvae from high-throughput axial-view in vivo imaging, Apr 2017. The Royal Society.

[7] M. Khazdan. Poisson surface reconstruction C++, 2015. https://github.com/mkazhdan/PoissonRecon/.

[8] M. Lardelli. Using zebrafish in human disease research: some advantages, disadvantages and ethical considerations, Jun 2017.

[9] J. Meeuwsen. Design and implementation of 3d reconstruction from axial views on the LIACS life sciences cluster, 2017.

[10] M. Molero. Poisson surface reconstruction Python binding, 2015. https://github.com/mmolero/pypoisson.

[11] M. Pluta. Advanced light microscopy vol. 1 principles and basic properties, 1988. Elsevier.

[12] O Shimomura, F.H. Johnson, and Y Saiga. Extraction, purification and properties of aequorin, a bioluminescent protein from the luminous hydromedusan, aequorea, June 1962. Published by Journal of Cellular and Comparative Physiology.

[13] S. N. Trede, M. D. Langenay, D. Traver, T. A. Look, and I. L. Zon. The use of zebrafish to understand immunity, Nov 2012. Palgrave Macmillan UK.

[14] W. Verhoef. Convolutional neural networks for automatic classification of radar signals in time domain: Learning the micro-doppler signature of human gait, 2019.