

Universiteit Leiden
The Netherlands

Opleiding Informatica

Evaluating parallel Rust by
porting a biomedical pollen imaging pipeline

Marc Hoek

Supervisors:
Rob van Nieuwpoort & Wouter J. van den Berg

BACHELOR THESIS

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

20/07/2026

Abstract

Rust and Rayon promise compile-time memory safety and "fearless parallelism" at a time when governments and industry increasingly advocate the adoption of memory-safe programming languages and multi-core processors have become ubiquitous. To evaluate trade-offs between development effort and performance, we ported a biomedical pollen imaging pipeline to both safe sequential Rust and parallel Rust using Rayon. We performed benchmarks of the resulting implementations on the DAS-6 supercomputer and compared against previously evaluated implementations developed using ImageJ, DIPlib, JAX, Triton, and CUDA. The results show that the Rayon implementation consistently outperforms the evaluated CPU-based implementations across all tested image sizes and achieves better overall performance than the GPU-based JAX implementation. Although highly optimized GPU implementations based on Triton and CUDA generally remain faster, Rayon achieves competitive performance for several pipeline stages and even outperforms Triton for the largest evaluated image sizes. Introducing Rayon required only 11.45% additional lines of implementation code compared to the sequential Rust implementation, with much of the parallelization achieved through high-level parallel iterator abstractions. However, both Rust implementations remain substantially larger in code size compared to most of the evaluated alternatives. These findings suggest that Rust and Rayon provide accessible, competitive CPU-parallel performance and memory safety with a larger implementation complexity cost up-front.

Contents

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Thesis overview . . . 2

2 Related work **2**

3 Background **3**
- 3.1 Pollen classification system . . . 4
- 3.2 The pollen pipeline . . . 5
  - 3.2.1 Minimum projection . . . 5
  - 3.2.2 Thresholding . . . 5
  - 3.2.3 Fill holes . . . 5
  - 3.2.4 Euclidean distance transform . . . 5
  - 3.2.5 Watershed filter . . . 5
  - 3.2.6 Particle filtering . . . 6
  - 3.2.7 Particle projection . . . 6
- 3.3 Rust . . . 6
- 3.4 Rayon . . . 6
- 3.5 Reference technologies . . . 7

4 Methodology **8**
- 4.1 Development methodology . . . 8
  - 4.1.1 Correctness validation and optimization method . . . 9
- 4.2 Sequential Rust implementation . . . 9
- 4.3 Rayon implementation . . . 10
  - 4.3.1 Considerations when porting over pipeline stages . . . 10
- 4.4 Gathering Performance Statistics . . . 12
  - 4.4.1 Input Data . . . 12
  - 4.4.2 Experimental Platform . . . 13
  - 4.4.3 Measurement Procedure . . . 14
  - 4.4.4 Data Processing and Analysis . . . 14

5 Results **14**
- 5.1 Performance scaling over input size . . . 14
- 5.2 Pipeline stage performance comparison . . . 15

6 Discussion **17**
- 6.1 Added development effort and performance of Rayon . . . 17
- 6.2 Technology comparison . . . 18

7 Conclusions and Future Work **19**

References **24**

1 Introduction

Processors with Multi-Core architectures are ubiquitous across modern desktops, laptops, servers, and mobile devices, making parallel computation a fundamental aspect of modern software systems [NNDS22]. While hardware manufacturers have continued to increase the number of available processing cores, many applications still fail to fully utilize the computational resources available to them. Exploiting multiple CPU cores efficiently remains challenging, often requiring developers to concern themselves with topics such as task scheduling, synchronization, memory access patterns, and hardware-specific performance characteristics.

Traditional CPU parallelization frameworks, such as OpenMP [DM98], provide powerful mechanisms for exploiting parallelism but can be difficult to use effectively. Achieving meaningful performance improvements and avoiding parallelism bugs frequently requires substantial expertise in parallel programming and multi-core processor architectures [GAO+16]. As a result, developers without a background in high-performance computing may struggle to obtain speedups or may introduce subtle concurrency bugs. At the same time, concerns regarding software security and reliability have led both industry and governments to advocate for the adoption of memory-safe programming languages [RK24, CISACNN23].

The Rust programming language has emerged as a notable response to these challenges. Rust provides strong memory-safety guarantees without requiring a garbage collector and promotes "fearless parallelism" as a central design goal [KN23]. In addition to the language's safety guarantees, libraries such as Rayon [ea25], considered to be Rust's standard data-parallelism library [PLH+21], aim to make CPU parallelism accessible to non-experts by allowing certain forms of sequential code to be converted into parallel code with minimal modifications. Rayon automatically manages work scheduling and load balancing across available CPU cores, potentially enabling developers to benefit from modern multi-core hardware without requiring extensive knowledge of parallel computing techniques.

The potential benefits of such approaches are particularly relevant in scientific computing domains. Biomedical image analysis and machine learning applications, for example, routinely process large datasets and high-resolution images, making them natural candidates for parallel execution. Nevertheless, biomedical researchers often continue to rely on sequential image-processing tools such as ImageJ [TFea26], partly because of their maturity, extensive feature sets, and ease of use [SYK+22, DSS+23, DSS+24].
A previous study investigated this issue by evaluating the effort and performance trade-offs involved in porting a sequential biomedical pollen imaging pipeline [Pol21] to GPU-based implementations [dBeaed]. The study demonstrated that substantial speedups could be achieved through parallelization, but also found that the porting process was non-trivial even when using higher-level GPU programming frameworks such as JAX and Triton.

This thesis investigates whether Rust and Rayon can provide a more accessible path toward high-performance scientific software. Specifically, we evaluate the effort-performance trade-off of porting the pollen imaging pipeline from [Pol21] to both safe sequential Rust and parallel Rust using Rayon. We developed new implementations of the pipeline and benchmarked them against

the implementations evaluated in [dBeaed]. By comparing development effort and performance outcomes, this work assesses whether safe Rust and high-level parallelism libraries can enable non-expert developers to effectively utilize modern multi-core CPUs while maintaining strong safety guarantees. The results contribute to understanding the feasibility of Rust-based approaches for scientific high-performance computing projects.

1.1 Contributions

The main contributions of this bachelor thesis are:

- safe sequential Rust port of the pollen pipeline described in [Pol21]
- CPU-parallel Rust port using Rayon of the pollen pipeline described in [Pol21]
- performance statistics and comparison of Rayon port against pre-existing implementations of the pollen pipeline defined in [dBeaed]
- an evaluation of the suitability of Rust and Rayon for CPU-parallel programming by analyzing the trade-off between development effort and achieved performance

1.2 Thesis overview

The thesis is structured as follows. In section 2 we discuss previous research that this thesis builds upon. Section 3 provides descriptions of the pollen pipeline and the technologies used to develop the new ports. We describe the general development strategy used to implement both ports and provide a complete description of our benchmarking method in Section 4. We present and analyze performance statistics of all available versions of the pollen pipeline in Section 5. In Section 6 we discuss the trade-off between the reasonably achievable performance and development effort of each Rust port. Lastly, Section 7 concludes the thesis.

2 Related work

Several studies have investigated the performance characteristics of Rust and the Rayon parallelism library in high-performance and parallel computing contexts. This section discusses a variety of these related works, while highlighting the ways this thesis contributes to these findings.

A work by Michal Sudwoj [Sud20] evaluated Rust on a finite-difference stencil algorithm and compared its performance against equivalent Fortran and C++ implementations. The authors found that Rust achieved performance comparable to these languages. In particular, the Rayon-based implementation was the second-fastest parallel implementation evaluated, ranking only behind a C++ implementation using OpenMP and being compiled using the Intel toolchain and remaining within approximately 20% of its execution time.
The evaluation of Rust and Rayon implementations of the NAS Parallel Benchmarks (NPB) by Martins et al. [MFH+25] yielded similar results. Their study compared sequential implementations written in Rust, C++, and Fortran, as well as parallel implementations using Rayon, C++ with OpenMP, and Fortran with OpenMP. The sequential benchmarks showed Rust to be highly

competitive, achieving performance 5.59% faster than C++ on average while remaining only 1.23% slower than Fortran. However, the Rayon implementations were 16.42% slower than Fortran with OpenMP and 19.05% slower than C++ with OpenMP. The authors further observed that achieving maximum performance often required selective use of Rust's `unsafe` facilities, particularly to eliminate bounds-checking overhead in performance-critical sections of code.
Both results indicate that while Rust itself is suitable for HPC workloads, Rayon may not always match the performance of mature (OpenMP-based) parallel implementations. One factor not considered in these papers is how the compile-time and run-time guarantees provided by safe Rust and Rayon may offer a positive trade-off in terms of development time and effort in non-HPC contexts, such as speeding up scientific computing without the need for expert knowledge. This includes features such as compile-time checks for data-races and Rust's built-in memory safety, among others. Frameworks like OpenMP do not provide these runtime and compile-time guarantees, so these might be less suitable in environments where expert knowledge or experience on parallel programming is uncommon. This thesis compares Rayon with parallelization libraries other than OpenMP, and takes each inspected technology's programming effort trade-offs into account.

The study "When Is Parallelism Fearless and Zero-Cost with Rust?" [APZ+24] examined the broader claim that Rust provides "fearless" and "zero-cost" parallelism. The authors distinguish between regular and irregular parallel workloads. For regular data-parallel applications, where memory accesses are predictable and can be verified statically, Rust and Rayon were found to provide genuinely low-overhead parallelism with strong compile-time safety guarantees. However, for irregular workloads such as graph processing and other algorithms with dynamic memory access patterns, the authors argue that Rust programmers often face trade-offs between performance, safety, and implementation complexity. Their benchmark study found that approximately 29% of parallel memory accesses in the evaluated workloads required mechanisms such as runtime checks, synchronization, or `unsafe` code, suggesting that Rust's "fearless concurrency" model is most effective for regular data-parallel computations.
These findings are in line with the results from Section 5, where algorithms with irregular data access patterns perform worse generally. Proper parallelization of the floodfill and watershed pipeline stages (see Section 4.3) also required the use of a custom data-structure that makes use of atomics, requiring explicit reasoning about concurrent access and synchronization from the programmer. However, the low-level control over implementation details does give Rayon an edge over other parallelization libraries when it comes to peak-achievable performance, even without using the `unsafe` keyword.

3 Background

This chapter introduces the pollen detection and projection pipeline studied in this thesis together with the technologies that were used to implement it. First, the overall pollen classification system is described, followed by an overview of the individual stages of the image-processing pipeline. Finally, the Rust programming language and Rayon library are introduced, together with a brief overview of the reference implementations from [dBeaed] against which the Rust-based implementations are compared.

3.1 Pollen classification system

Polling et al. [Pol21, LPC+23] proposed an automated pollen classification system that uses microscopy and deep learning to identify airborne pollen grains. The goal of the system is to estimate the types and concentrations of pollen present in the air, for example to support allergy forecasting. The pollen pipeline studied in this thesis forms the preprocessing stage of this larger system.

An overview of the complete system is shown in Figure 1. First, a sample of airborne pollen is collected on adhesive microscope slides using volumetric air samplers. The slides are subsequently digitized by acquiring a stack of microscope images at different focal depths, producing a three-dimensional image volume. Because pollen grains are three-dimensional objects, a single focal plane cannot capture every grain in focus. Instead, multiple images are acquired along the optical axis to ensure that each pollen grain is sharply represented in at least one slice of the image stack.

The image stack is then processed by the pollen pipeline, which is the focus of this thesis. The pipeline detects individual pollen grains, separates overlapping particles, removes objects that are unlikely to correspond to pollen, and extracts each detected grain from the original image stack. For every detected pollen grain, three two-dimensional projection images are generated from the original image stack. These projections summarize information from the complete focal stack while suppressing out-of-focus blur, allowing complementary structural information to be preserved in a format suitable for subsequent neural network classification.

Finally, these projection images are used as input to a convolutional neural network, which classifies the pollen grain into one of the supported taxonomic groups. The resulting classifications can subsequently be aggregated to estimate airborne pollen concentrations, enabling automated pollen monitoring with significantly reduced manual effort [Pol21, LPC+23].

Since the preprocessing pipeline is responsible for detecting pollen grains and generating the projection images used for classification, its performance directly influences the throughput of the complete automated pollen classification system. Consequently, improving the efficiency of this pipeline can substantially reduce the time required to process large microscopy datasets.

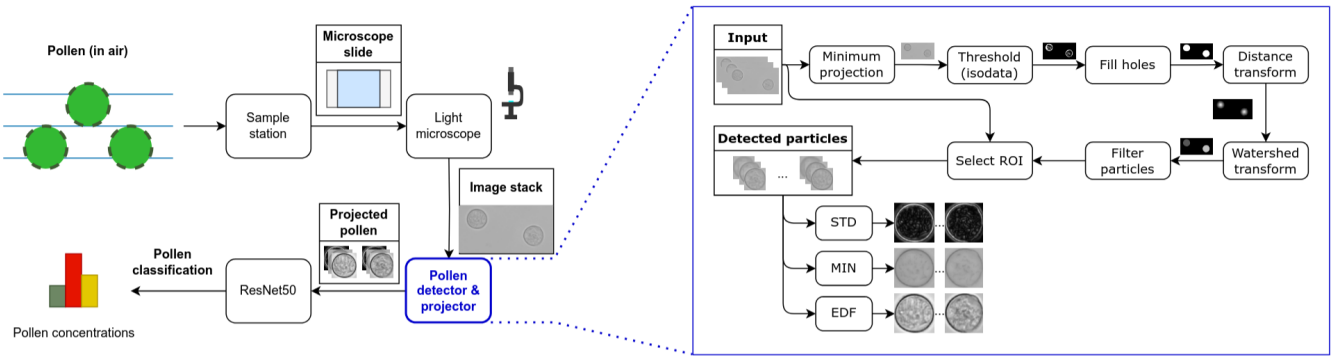


Figure 1: Overview of the full system, with the pipeline studied in this paper highlighted and expanded on the right. Figure originally from [dBeaed] and included pollen images from [LPC+23]

3.2 The pollen pipeline

The pollen pipeline processes a stack of microscope images to detect individual pollen grains and generate projection images that can be used for classification. Figure 1 provides an overview of the complete pipeline. Each stage performs a specific image-processing operation, gradually transforming the raw microscope images into segmented pollen grains suitable for subsequent neural network classification.

3.2.1 Minimum projection

Microscope images are acquired as a stack of grayscale images at different focal depths. Since pollen grains are generally darker than the background, the first stage computes the minimum intensity value for every pixel across all slices of the image stack. The resulting minimum intensity projection combines information from all focal planes into a single two-dimensional image while preserving the dark structures of the pollen grains.

3.2.2 Thresholding

The minimum projection is converted into a binary image by separating foreground pixels, corresponding to pollen grains and other particles, from the background. This is achieved by automatically selecting an intensity threshold using the ISODATA thresholding method [RC+78]. Pixels darker than the threshold are classified as foreground, while the remaining pixels are assigned to the background.

3.2.3 Fill holes

The binary image from the thresholding stage may contain holes inside detected pollen grains because their interiors often appear brighter than their edges. These enclosed background regions are filled to produce solid foreground objects. Filling holes improves the accuracy of later processing stages by ensuring that each pollen grain forms a single connected region.

3.2.4 Euclidean distance transform

The Euclidean distance transform assigns every foreground pixel a value equal to its distance from the nearest background pixel. As a result, pixels near the center of an object obtain larger values than those near its boundary. The resulting distance map highlights the approximate centers of pollen grains and provides the basis for separating touching or overlapping objects needed for the watershed stage.

3.2.5 Watershed filter

Neighboring pollen grains frequently touch or overlap in microscope images, causing them to appear as a single connected object after thresholding. The watershed transform uses the distance map to divide these connected regions into individual particles by identifying the boundaries between neighboring pollen grains. This produces a segmentation in which each detected pollen grain is represented as a separate labelled object.

3.2.6 Particle filtering

Not every segmented object corresponds to a pollen grain. Dust particles, debris, and segmentation artefacts may also be detected. The particle filtering stage removes such objects by evaluating geometric properties including particle size, circularity, and other shape characteristics. Only objects that satisfy the expected characteristics of pollen grains are retained for further processing.

3.2.7 Particle projection

For each remaining pollen grain, the corresponding region is extracted from the original microscope image stack. Three projection images are subsequently generated from the extracted image stack, a minimum projection across slices, a standard deviation projection across slices, and an extended focus projection based on an extended depth of field projection described in [AVDVU08]. Each projection emphasizes different characteristics of the pollen grain while reducing the influence of out-of-focus blur. These projection images form the input to the neural network that performs the final pollen classification, which is out of scope for this thesis.

3.3 Rust

Rust is a systems programming language designed to provide performance comparable to traditional languages such as C and C++ while enforcing strong memory safety guarantees at compile time [KN23]. Unlike other memory-safe languages, Rust does not rely on a garbage collector to enforce memory safety. Instead, it employs an ownership model in which every value has a single owner, while the borrow checker statically verifies that references obey strict aliasing and lifetime rules. This prevents common memory safety errors, including use-after-free, double-free, and dangling pointer bugs, without introducing runtime overhead.

Rust's ownership and type systems also extend these guarantees to concurrent programs. In safe Rust, that is, Rust code that does not use the `unsafe` keyword, the compiler prevents data races by ensuring that mutable data cannot be accessed simultaneously from multiple threads. Consequently, many classes of concurrency bugs are eliminated during compilation rather than at runtime. Although the `unsafe` keyword allows programmers to bypass some of these guarantees when necessary, it is intended for low-level operations where the programmer must manually uphold Rust's safety invariants.

Rust further emphasizes performance through zero-cost abstractions, language features that compile to code with little or no runtime overhead compared to equivalent handwritten low-level implementations [KN23]. Combined with the LLVM compiler backend, this enables Rust programs to achieve performance comparable to optimized C and C++ applications while retaining the language's compile-time safety guarantees. These properties make Rust particularly well suited for high-performance computing applications, where both execution efficiency and program correctness are important.

3.4 Rayon

Rayon is a data-parallel programming library for Rust that provides high-level abstractions for exploiting shared-memory parallelism [ea25]. Rather than explicitly creating and managing threads, programmers express parallelism using constructs such as parallel iterators, allowing existing

sequential iterator-based code to be parallelized with relatively few modifications. Rayon internally employs a fork–join execution model, recursively dividing work into independent tasks that are executed in parallel before their results are combined.
To distribute work efficiently across available processor cores, Rayon uses a work-stealing scheduler. Each worker thread maintains its own local task queue and preferentially executes tasks from this queue. When a worker becomes idle, it attempts to steal work from the queues of other workers, providing dynamic load balancing for irregular workloads [ea25].
Rayon integrates closely with Rust's ownership and type systems. The library only permits data to be shared between threads when it satisfies Rust's thread-safety requirements, allowing the compiler to prevent data races in safe Rust programs. As a result, Rayon enables efficient shared-memory parallelism while preserving Rust's compile-time memory and thread-safety guarantees.

3.5 Reference technologies

To evaluate the Rust implementations, their performance is compared against five previously developed implementations of the pollen pipeline presented by J. van den Berg et al. [dBeaed]. These implementations span a range of programming models, from high-level image-processing frameworks to low-level GPU programming environments, providing different trade-offs between implementation effort and performance.

ImageJ ImageJ [TFea26] is an open-source image-processing framework written in Java that is widely used in scientific imaging [SRE12]. It provides a large collection of built-in image-processing operations together with an extensive plugin ecosystem and macro language, making it well suited for rapidly prototyping image-processing pipelines. Although ImageJ supports parallel execution through plugins, the implementation evaluated in [dBeaed] primarily executes sequentially, serving as a high-level CPU-based baseline with relatively low implementation complexity.

DIPlib DIPlib [con26] is a C++ image-processing library that provides optimized implementations of many common image-processing operations. Several of the pipeline stages automatically exploit shared-memory parallelism through OpenMP [DM98], while stages with more irregular data access patterns remain largely sequential. Specifically, the hole fill and watershed stages are implemented sequentially. Compared to ImageJ, DIPlib provides higher CPU performance while staying at a similar level of abstraction.

JAX JAX [FJL19] is a high-level numerical computing framework for Python that provides a NumPy-like programming interface while targeting hardware accelerators through the XLA compiler. By expressing computations as array operations, JAX can automatically generate optimized GPU kernels without requiring explicit GPU programming. This programming model is particularly well suited to regular, data-parallel computations that can be expressed using vectorized operations. However, implementing algorithms involving irregular memory accesses, dynamic control flow, or branching logic often requires restructuring the algorithm to fit JAX's functional programming model and XLA's compilation constraints. Consequently, while JAX offers relatively low implementation effort for standard numerical workloads, more complex image-processing algorithms can require substantially greater development effort.

Triton Triton [TKC19] is a domain-specific language for GPU programming embedded in Python that provides a lower-level programming model than JAX while remaining more accessible than native CUDA. Rather than relying on automatic kernel generation, programmers explicitly define GPU kernels, which are just-in-time (JIT) compiled before execution. This approach provides significantly greater control over memory accesses, thread organization, and parallel execution. Unlike JAX's array-oriented programming model, Triton supports more dynamic control flow and branching within JIT-compiled kernels, making it better suited to irregular algorithms such as several stages of the pollen pipeline. Consequently, Triton offers a compromise between programmer productivity and GPU performance, trading a higher implementation effort for increased flexibility and optimization opportunities.

CUDA CUDA [K+07] is NVIDIA's native parallel computing platform and programming model for general-purpose GPU computing. It provides direct control over GPU threads, memory hierarchy, and synchronization, enabling highly optimized implementations that fully exploit the available hardware parallelism. This flexibility generally allows the highest attainable performance but also requires substantially greater implementation effort and expertise than the higher-level frameworks considered in this thesis.

4 Methodology

4.1 Development methodology

We developed the Rust implementations incrementally using a combination of test-driven development [Bec03] and performance-driven optimization. The primary objective was to preserve the behavior compared to a reference implementation while attempting to maximize the performance of safe Rust and safe Rust with Rayon.
We chose the pipeline implemented using Triton described in [dBeaed] as the reference implementation. This implementation was selected because it achieved competitive performance while remaining relatively readable compared to the CUDA implementation. Furthermore, the Triton implementation already contained adaptations of several GPU-based algorithms used throughout the pipeline, making it a great reference for proper parallelization techniques for the Rayon port.
The overall development of the ports needed for the experiments proceeded in four phases:

1. We created a functionally correct sequential Rust implementation by porting the Triton implementation.

2. We optimized this implementation while preserving correctness using the validation methodology described in Section 4.1.1

3. We introduced parallelism using the Rayon library, after which the parallel implementation was further optimized while continuing to validate correctness using the same methodology.

4. We ported any optimizations discovered during the implementation of the Rayon pipeline over to the sequential Rust pipeline where applicable. This ensured that any performance difference between each port is the result of the added parallelization through Rayon.

4.1.1 Correctness validation and optimization method

We validated correctness continuously throughout development. For each pipeline stage, the original unit tests from Triton were ported over before the implementation itself. During development, intermediate outputs produced by the Rust implementation were validated by the ported unit tests and subsequently compared against the corresponding outputs generated by the Triton implementation. This stage-by-stage validation approach allowed implementation errors to be isolated and corrected before they were propagated through later pipeline stages.
We performed performance optimization iteratively. After each significant modification, we collected benchmark results using a procedure similar to that described in Section 4.4.3, but on locally available development hardware equipped with a multi-core CPU and using a subset of the pollen scans described in Section 4.4.1. Only the $20 \times 4000 \times 4000$ and $20 \times 8000 \times 8000$ image sets were used during development, as they provided a practical balance between rapid iteration and sufficient sensitivity to detect the effects of fine-grained optimizations. After performing measurements, we compared them against previous benchmarks to detect regressions. This process resembles benchmark-driven development, where optimization decisions are guided by observed measurements rather than assumptions regarding performance characteristics.

4.2 Sequential Rust implementation

The first implementation target was a version of the pollen pipeline written entirely in safe sequential Rust. This implementation served both as a correctness reference for the later parallel implementation and as a baseline for evaluating the performance impact of Rayon.
We implemented the command-line user interface, input and output TIFF image handling, benchmarking logic and output structure of the program to be behaviorally identical to the pipeline implementations from [dBeaed]. Reusing the same external behavior simplified comparison with the previous implementations by making it compatible with the same utility scripts used for testing and generating results from [dBeaed].
One practical challenge encountered during the porting process was TIFF image handling. The Triton implementation relies on higher-level image processing libraries that provide convenient abstractions for reading and writing multi-page TIFF files for the purpose of reading and writing ImageJ compatible microscope-scans. Equivalent functionality was not directly available in the Rust ecosystem. Consequently, substantial portions of the TIFF reading infrastructure were implemented manually using the lower-level APIs provided by the `tiff` crate [tif26], while the TIFF writing infrastructure required minor modifications to the `tiffwrite` crate [wim26]
During evaluation, the version of the `tiff` crate used in this work rejected images of size $20 \times 20000 \times 20000$ due to a built-in maximum image size limit. As a result, benchmarks for this image size could not be performed and are therefore omitted from Figure 2. This limitation is expected to be resolved by updating to a future version of the crate or by applying a local patch that removes the image size restriction.
After we implemented the main program interface, we ported each of the pipeline stages individually. For each stage, we translated the associated tests first, followed by the implementation itself. This ensured that functionality could be validated immediately after porting. Once all tests passed and the runtime behavior matched that of the reference implementation, we optimized the code where appropriate.

Optimization efforts focused primarily on reducing memory allocations, improving cache locality, and eliminating unnecessary data movement. Since the sequential implementation also served as the basis for the parallel implementation, preference was generally given to optimizations that would remain applicable after introducing Rayon-based parallelism. However, several optimizations were found that were only applicable in a sequential context and were therefore not retained in the parallel version. Examples include not needing to use atomic data types in used data structures or using less parallelizable, but more cache-efficient data access patterns. This thesis decided to keep these optimizations in the sequential port and their exclusion from the Rayon port is treated as an inherent trade-off of introducing parallelism.

4.3 Rayon implementation

After obtaining a functionally correct and sufficiently efficient sequential implementation, we introduced parallelism using the Rayon library. The goal was not merely to parallelize the pipeline, but also to evaluate how effectively Rayon enables non-expert developers to exploit multi-core processors while maintaining memory safety guarantees and mitigating data races.
Our parallelization process began with straightforward substitutions of sequential iterators with Rayon parallel iterators. This provided an initial parallel baseline and helped identify which pipeline stages could benefit from embarrassingly parallel execution.
Subsequently, we analyzed each pipeline stage individually. After performance bottlenecks were identified through benchmarking, we applied stage-specific optimizations to remove or mitigate them. We validated correctness after every modification using the existing test suite and by comparing intermediate outputs against those of the sequential Rust implementation and the Triton implementation from [dBeaed].
The optimization process followed an iterative process, similar to the correctness validation procedure. Following each optimization, we compared benchmark results against previous measurements. If a modification failed to produce a measurable improvement or negatively affected scalability, we discarded it. This approach ensured that we introduced additional implementation complexity only when justified by measurable performance gains.
Several parallelization patterns emerged throughout the implementation. Operations involving independent pixels or image rows were typically parallelized using data-parallel iterators. Aggregation tasks, such as histogram construction and particle statistics collection, were implemented using parallel map-reduce patterns. More complex stages, including connected-component labeling and watershed segmentation, required custom lock-free data structures based on atomic operations. Implementation and optimization considerations are discussed in more detail in Section 4.3.1.

4.3.1 Considerations when porting over pipeline stages

This subsection discusses the most important considerations and implementation decisions for each pipeline stage.

Minimum projection This pipeline stage is trivially parallelizable over each output pixel. We optimized performance here by parallelizing over chunks of output pixels. The width of the input-image, `width` was used for the chunk-size. This made the chunk size scale appropriately for each

square input image size, had a side-effect of reducing false sharing and ensured cache-performance was predictable per chunk.

Thresholding To compute the histogram needed for thresholding efficiently, we used a parallel map-reduce pattern[DG08] over a chunk of pixels of a significant size for the image. For the chunk-size `img_slice.len() / rayon::current_num_threads()` in order to divide the work evenly across every available thread and to reduce the amount of needed allocations and reduction steps. Each parallel task collects it's own histogram over it's assigned chunk. Each local histogram is then summed up in parallel to create the final histogram. This is the recommended way to compute multi-element associative operations, like sums, in parallel. We kept all iterations over the histogram sequential, as attempts to parallelize those led to performance-degradation related to Rayon's task-scheduling overhead. The last thing that was worth parallelizing in this stage, was the production of the boolean mask output, which we accomplished using a simple parallel `for_each` pattern [par26b].

Fill holes To parallelize the hole fill algorithm in an efficient way, we adapted the component labeling algorithm from the Triton implementation from [dBeaed], which is an adaptation of an algorithm from [RT06] using an implementation of union-find methods for the GPU from [OL10]. Implementing this in parallel Rust involved implementing a lock-free data structure that implements common union-find operations, including the `find` and `union` operations. This means that properly parallelizing this stage required knowledge on synchronization primitives, atomics in this case, and lock-free programming principles. However, this was worth it as it led to performance competitive with Triton for this stage (see Figure 3).

Euclidean distance transform To efficiently compute the Euclidean distance transform (EDT) on the CPU, we adapted the algorithm by Felzenszwalb and Huttenlocher [FH12] instead of the more GPU-oriented Jump Flooding-based approach used in the Triton implementation [RT06]. The Felzenszwalb–Huttenlocher method computes the exact EDT in $\mathcal{O}(H \cdot W)$ time and is separable across dimensions, making it well-suited for parallel execution over independent rows and columns. Comparatively, the Jump Flooding-based approach has a higher time complexity ($\mathcal{O}(H \cdot W \cdot lg(max(H, W))))$ and has more irregular data access patterns.
The implementation follows the described two-pass decomposition described in [FH12]. A 1D squared distance transform is first computed along each independent image column in parallel, followed by a second parallel pass along each independent row which is combined with a conversion of squared distances to euclidean distances. Each 1D transform is implemented using the lower-envelope construction of parabolas as described in [FH12].
We introduced parallelism at the level of independent 1D transforms. Independent columns and rows were distributed across worker threads using a parallel `for_each` pattern [par26b]. To reduce allocation overhead and improve cache locality, each thread maintains a reusable scratch buffer containing the parabola indices and boundary values required by the envelope construction.

Watershed We implemented the watershed stage by adapting an approximation algorithm present in the Triton implementation, that was based on an approach from [YV18]. Like the Hole Fill stage, this algorithm also made use of parallel union-find operations. Therefore, we parallelized the union

find operations using the existing union-find data structure implemented for the Hole Fill stage. The H-maxima initialization and final label generation were implemented using simple parallel iterators over output pixels. During the initialization phase, each pixel independently determines its local state and label pointer by inspecting its neighborhood, making this step embarrassingly parallel. The plateau resolution step was more challenging due to dependencies between neighboring pixels. Instead of updating pixels directly in parallel, we used a two-phase approach: candidate updates were first collected in parallel using a filter-map pattern [par26a] over candidate pixels in a frontier, which were then applied while building the next frontier. This avoided data races and unnecessary work while retaining most of the available parallelism. Finally, plateau merging was parallelized over image rows, where neighboring plateau pixels were connected using the `union` operation of the union-find data structure. After all unions were performed, A final parallel pass resolves union-find roots and generates both the labeled output image and the set of unique region identifiers.

Particle filtering This stage first collects the total area, perimeter and bounding box information of each particle found in the watershed stage and then filters out particles that are considered misclassified using a heuristic. To efficiently collect each label information in parallel, we pass over the image once to collect all three needed statistics simultaneously. This is accomplished using a parallel map-reduce pattern over each output row. To improve memory efficiency for each row, label data is collected using a dynamically scaling hash map, since not all labels will be present in each row. The final filtering step of this stage was trivially parallelizable over each label using a parallel filter-map operation [par26a].

Particle projection The particle projection stage is embarrassingly parallel over each separate particle via a parallel `map` operation and each separate projection via rayon's built in `join` function [joi26]. The standard deviation projection, like the earlier discussed minimum projection, was trivially parallelizable over the 2D output pixels. The Extended Focus (EDF) projection was more complex, containing more steps that depended on each other's output. We parallelized all of these steps separately with the use of nested parallel iterators over slices and rows to provide enough tasks for each core to work on.

4.4 Gathering Performance Statistics

After the safe sequential Rust and Rayon ports had been implemented and iteratively optimized, the final steps of this research focused on their evaluation. Specifically, we evaluated the performance of both pipelines against the performance of the previously published pipelines, and compared the two implementations in terms of implementation complexity and performance cost.
To enable a direct comparison with the results reported by J. van den Berg et al. [dBeaed], our experimental setup and measurement methodology follow those of the original study as closely as possible.

4.4.1 Input Data

We performed the evaluation using the same microscope image stacks as those used in the experiments of [dBeaed]. Each input consisted of a multi-page TIFF image containing 20 grayscale slices

acquired at different focal depths, together forming a three-dimensional image stack.
To evaluate the scalability of the implementations, six image sizes were considered:

- $20 \times 2000 \times 2000$ pixels (80 MB)
- $20 \times 4000 \times 4000$ pixels (320 MB)
- $20 \times 8000 \times 8000$ pixels (1280 MB)
- $20 \times 12000 \times 12000$ pixels (2880 MB)
- $20 \times 16000 \times 16000$ pixels (5120 MB)
- $20 \times 20000 \times 20000$ pixels (8000 MB)

For each image size, we benchmarked the implementations using a dataset of ten distinct microscope image stacks. As mentioned in Section 4.2, benchmarks on datasets containing images of size $20 \times 20000 \times 20000$ could not be performed for the Rust-based implementations due to limitations in the used version of the `tiff` crate. Nevertheless, the remaining images span a wide range of input sizes, allowing the scalability of the different pipeline implementations to be evaluated under increasingly demanding workloads.

4.4.2 Experimental Platform

We executed all experiments on nodes of the Distributed ASCI Supercomputer 6 (DAS-6) [BEdL+16]. Each node was equipped with the following hardware:

- CPU: $2 \times$ AMD EPYC 7282 ($2 \times 16 = 32$ cores, $2 \times 32 = 64$ threads, 2.8 GHz)
- Main memory: 128 GB RAM
- CPU peak memory bandwidth: 2×85.3 GB/s $= 170.6$ GB/s [CPU]
- CPU estimated peak floating point performance: $2\times \sim 725$ GFLOPS ≈ 1.45 TFLOPS [Esh19]
- GPU: NVIDIA RTX 4000 Ada (48 Streaming Multiprocessors, 192 Tensor Cores, 1.5 GHz)
- GPU memory: 21.5 GB VRAM
- GPU peak memory bandwidth: 360GB/s
- GPU peak floating point performance: 26.7 TFLOPS [NVI23]

The Rayon implementation spawned one operating-system thread per logical hardware thread, ensuring that all 64 logical hardware threads available on the node were utilized during the benchmarks. Although the Rust implementations did not use the GPU, we include its specifications for completeness because they are relevant when comparing against the GPU-based implementations discussed in Section 5.

4.4.3 Measurement Procedure

For each image size, both Rust pipelines were executed ten times. During each run, we recorded both the total pipeline runtime and the runtime of each individual pipeline stage.
We used the collected measurements to compute the mean execution time and standard deviation for each image size and pipeline component. These statistics provide insight into the implementation's average performance, average standard deviation and performance variation among test images.

4.4.4 Data Processing and Analysis

We stored performance measurements in NumPy archive (`.npz`) files, generating one file for each image size. These files contained the timing information required for subsequent analysis and visualization.
The resulting performance data were incorporated into the existing plots reported by J. van den Berg et al. [dBeaed], enabling a direct comparison between the Rayon implementation and the previously evaluated ImageJ, DIPlib, JAX, Triton, and CUDA implementations. This comparison allows the effectiveness of thread-level parallelism using Rayon to be assessed relative to both CPU-based and GPU-based parallelization framework/library alternatives.

5 Results

5.1 Performance scaling over input size

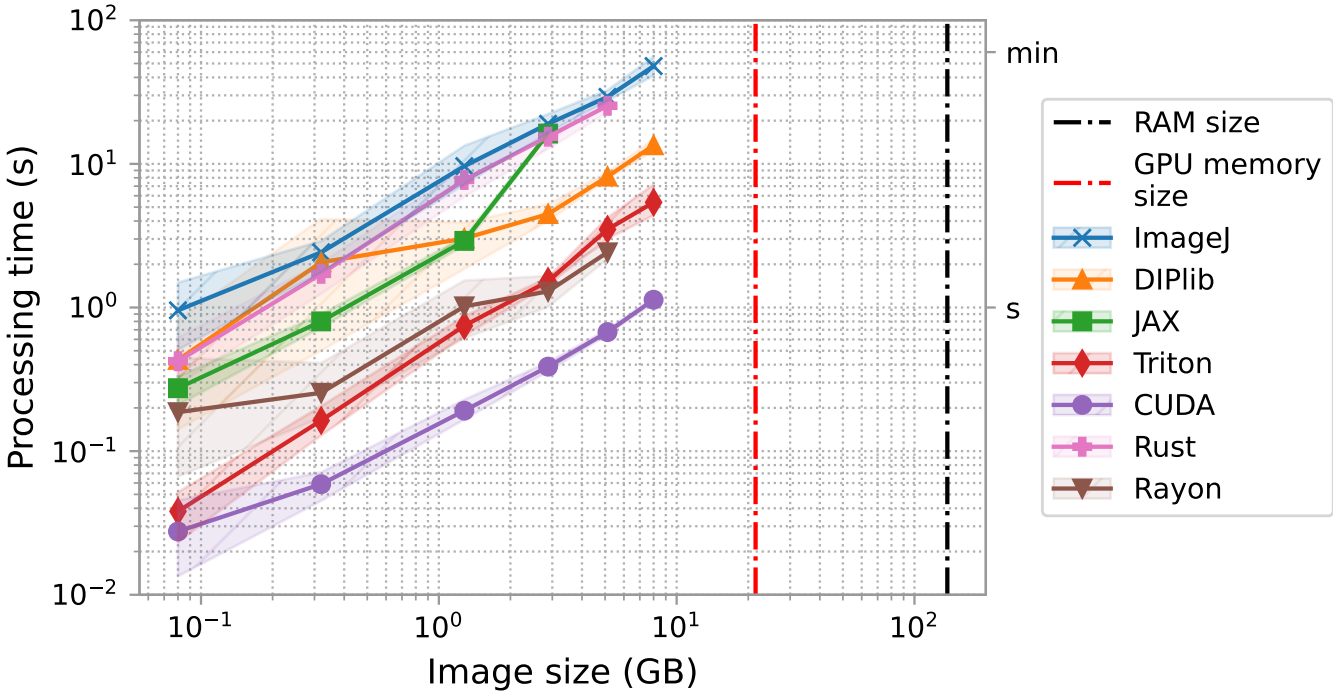


Figure 2: Mean processing time for different image sizes on an Ada 4000 GPU and AMD EPYC. Colored area around each line shows average performance difference between images.

Figure 2 shows a graph plotting the mean processing time over different image sizes for each mentioned pollen pipeline implementation from Section 3 along with the new Rust-based pipelines introduced in this paper. This figure is a modified version of the same figure found in [dBeaed], just with the new Rust-based pipeline implementations added. Each pipeline is represented as a line with a unique color and each measurement point on that line is marked with that pipeline's unique symbol. Lightly colored areas around a line indicate the 10th-90th percentile of the mean processing times across all used testing-images of that size. Both Rust-based implementations do not include data points for images of size $20 \times 20000 \times 20000$ due to limitations in the TIFF decoding library used, as described in Section 4.2.

The sequential Rust pipeline implementation is shown to perform similarly to the ImageJ implementation, being slightly faster overall. This shows that the ImageJ implementation implements no or a negligible amount of parallelism.
Furthermore, performance of Rust is roughly equal to JAX at images of $20 \times 12000 \times 12000$. Given that the growth of JAX' processing time is significantly larger compared to the growth of Rust from $20 \times 8000 \times 8000$ images to $20 \times 12000 \times 12000$, it is likely that Rust and probably even ImageJ will perform significantly better than JAX on larger images sizes.

The graph shows that the Rayon pipeline consistently performs better than the other CPU-based pipelines: ImageJ, Rust and DIPlib. Comparatively, the speedup of Rayon compared to ImageJ and Rust stays significant and grows with greater image sizes, while the speedup of Rayon compared to DIPlib starts very significant and becomes relatively less significant and stable around larger image sizes, showing a difference of around 5.5 seconds at $20 \times 16000 \times 16000$ images.
Furthermore, it consistently shows better performance than the JAX pipeline, which is primarily GPU-based. However, DIPlib also significantly outperforms JAX for images larger than $20 \times 8000 \times 8000$, suggesting that JAX is not well suited to the workloads required by the pollen pipeline. This is likely due to its high level of abstraction and its primary focus on machine learning applications rather than large-scale image processing.
The pipelines with GPU-based technologies that offer more low-level implementation control, Triton and CUDA, outperform the Rayon pipeline on lower image sizes. This shows the inevitable achievable gap that massively parallel GPUs have over CPUs in terms of total parallelization capability. While this performance difference stays relatively consistent for the CUDA pipeline, this is not the case for the Triton pipeline. On greater image sizes, Rayon is shown to outperform Triton slightly, with a difference of about 1 second at $20 \times 16000 \times 16000$ images. This suggests that languages with more control over memory management, CUDA C++ and Rust in this case, tend to scale better at workloads that use larger amounts of memory.

5.2 Pipeline stage performance comparison

Figure 3 shows the mean processing time of all $20 \times 8000 \times 8000$ images per pipeline stage for each pipeline implementation, including the total time. The means are shown along with their 10th–90th percentile range, labeled *measurement variation*, and their standard deviations, labeled *measurement + image variation*. Again, this is a modified version of the same figure found in [dBeaed] with the new Rust-based pipelines added. To ease performance comparison of Rayon with Rust and other pipelines, the mean gained execution time speedup of Rayon compared to every

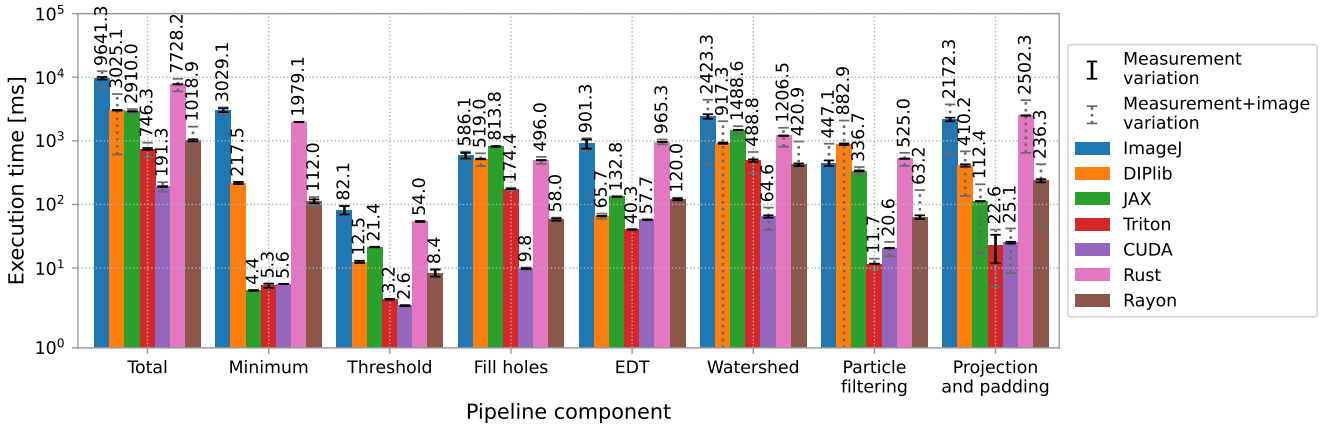


Figure 3: Performance per pipeline stage, averaged over all measured $20 \times 8000 \times 8000$ images. Solid error bars correspond to the 10—90th percentile range. Dotted error bars correspond to the standard deviation. CUDA resource allocation is measured separately and included in *Total* execution time

Pipeline	**Minimum**	**Threshold**	**Fill holes**	**EDT**	**Watershed**	**Filtering**	**Projection**	**Total**
ImageJ	27.05×	9.77×	10.10×	7.51×	5.76×	7.07×	9.19×	**9.46**×
DIPlib	1.94×	1.49×	8.95×	0.55×	2.18×	13.96×	1.74×	**2.97**×
JAX	0.04×	2.54×	14.03×	1.11×	3.54×	5.32×	0.48×	**2.86**×
Triton	0.05×	0.38×	3.01×	0.34×	1.16×	0.18×	0.10×	**0.73**×
CUDA	0.05×	0.30×	0.17×	0.48×	0.15×	0.33×	0.11×	**0.19**×
Rust	17.68×	6.42×	8.55×	8.55×	2.87×	8.30×	10.59×	**7.58**×

Table 1: Rayon's mean speedup over every other pipeline implementation discussed in this thesis when run on the $20 \times 8000 \times 8000$ image dataset. We compare speedup of each separate pipeline stage and the total pipeline execution

other pipeline in Figure 3 is shown in Table 1.
The main observations per pipeline stage, focusing on Rayon's performance, are as follows:

Total A more detailed version of the dynamic shown in Figure 2.

Minimum Shows significant performance improvement over the other CPU-based pipelines, while remaining at least 20× slower than each GPU-based implementation. This demonstrates that Rayon is highly effective at parallelizing embarrassingly parallel workloads, but not as effective as pipelines utilizing the GPU. As discussed in Section 4.4.2, the GPU used for evaluation provides substantially greater achievable computational throughput and more than twice the peak memory bandwidth of the dual-socket CPU. Since the minimum projection is one of the most memory-bound pipeline stages (reading $depth \times height \times width$ pixels and writing $height \times width$ pixels) and consists primarily of independent per-pixel reductions, the GPU is able to effectively exploit its parallelism and memory bandwidth advantages over the CPU.

Threshold An improvement compared to other CPU pipelines and significantly worse compared to GPU pipelines with lower level implementation control. Furthermore, the performance improvement over JAX is significant and the performance gaps between CPU and GPU pipelines are relatively smaller compared to the minimum projection stage. This indicates that the thresholding stage offers less exploitable parallel work per unit of memory accessed. Consequently, CPU implementations remain comparatively competitive for this stage when considering that the thresholding operation is embarrassingly parallel.

Fill holes Very competitive performance, even better compared to Triton at this image size. This shows that the extra effort needed to implement a lock-free algorithm can sometimes make a very large difference in total execution time. This is also a side-effect of Rayon offering enough low-level control over the parallelization method to make this possible.

EDT Rayon is approximately twice as slow as the DIPlib implementation for this stage. Although both implementations have the same computational complexity ($\mathcal{O}(H \cdot W)$), the Rust implementation performs significantly more column-wise memory accesses. Since image pixels are stored contiguously in row-major order, these accesses occur over much larger memory strides, reducing spatial locality [Sta16] and increasing the number of cache misses, ultimately resulting in poorer cache utilization. Future work includes adapting the Rayon EDT implementation to more closely mirror DIPlib's EDT implementation, with the goal of improving performance for this stage.

Watershed A significant improvement over the CPU pipelines and JAX, while also being slightly faster than Triton. This is likely due to the efficiency of the integrated Union-Find implementation as well as the effectiveness of the parallel filters mentioned in Section 4.3.1.

Particle filtering Rayon achieves a considerable speedup over the CPU implementations and JAX, while remaining only approximately 3× slower than CUDA. This suggests that Rayon's parallel implementations of the map-reduce and filter-map patterns are efficient. Additionally, the optimizations that reduced memory usage and the number of passes over the input image (see Section 4.3.1) likely also contributed to this performance.

Projection and padding Rayon outperforms DIPlib but remains substantially slower than all GPU implementations. Although Rayon exploits most of the available CPU parallelism during this stage, it cannot match the degree of parallelism provided by the GPU.

6 Discussion

6.1 Added development effort and performance of Rayon

Table 2 shows that introducing parallelism via Rayon required relatively little additional code. Excluding tests and the manually adapted `tiffwrite` crate, the sequential implementation consists of 1371 lines of code, while the Rayon implementation consists of 1528 lines, corresponding to an increase of only 11.45%.

Implementation	Tests?	`tiffwrite`?	Files	Lines of code
Rayon	yes	yes	16	2908
Rayon	no	yes	15	2225
Rayon	no	no	14	1528
Rust	yes	yes	15	2714
Rust	no	yes	14	2067
Rust	no	no	13	1371

Table 2: Total files and lines of source code per implementation written in Rust, counted using the `cloc` utility [ea]. For each implementation, multiple counts are shown that include and exclude testing code and code of the manually imported `tiffwrite` crate [wim26]. All counts shown exclude comments, empty lines and code of used external crates other than `tiffwrite`.

Much of the parallelization was achieved by replacing sequential iterator operations with their parallel equivalents and introducing a limited number of Rayon-specific constructs. Consequently, the overall program structure remained largely unchanged, allowing the sequential implementation to serve as a direct basis for the parallel version.
In contrast to the modest increase in implementation size, the achievable performance improvement is substantial. The Rayon implementation is 7.58× faster on average compared to the sequential implementation for $20 \times 8000 \times 8000$ images and approximately 10× faster for $20 \times 16000 \times 16000$ images. These results indicate that Rayon provides a favorable trade-off between implementation effort and performance for this workload.

6.2 Technology comparison

Implementation	Order developed	Files	Lines of code
Original ImageJ	0 (initial)	1	109
JAX	1	15	997
ImageJ + plugin	2	2	630
CUDA	3	27	2502
DIPlib	4	5	538
Triton	5	14	1315
Rust	6	13	1371
Rayon	7	14	1528

Table 3: Table 4 from [dBeaed] with added details of the Rust-based implementations. Files and lines of code are counted using `cloc`, and does not include code of tests, standard plugins and used external libraries.

Table 3 places the Rust implementations in the context of the previously published implementations. The sequential Rust implementation contains 1371 lines of implementation code, making it comparable in size to the Triton implementation (1315 LOC) while remaining considerably larger than the DIPlib implementation (538 LOC). Adding Rayon increases the implementation size to 1528 LOC, representing only a modest increase over the sequential version.

Although the Rayon implementation requires substantially more implementation code than DIPlib, it consistently achieves lower execution times across the evaluated datasets. Conversely, CUDA remains the fastest implementation overall, although this performance comes at the cost of a considerably larger implementation consisting of 2502 lines of code.
Triton achieves competitive performance while maintaining an implementation size comparable to the sequential Rust implementation. However, for the largest evaluated image sizes, the Rayon implementation slightly outperforms Triton despite requiring only a modest increase in implementation size over the sequential Rust version.
Although GPU-based implementations achieve the highest performance, they require suitable hardware and GPU-specific implementations. For systems without a dedicated GPU, or where portability across CPU architectures is preferred, the Rayon implementation provides an attractive compromise between implementation complexity and execution time.

7 Conclusions and Future Work

This thesis investigated the trade-off between development effort and performance gain of porting the pollen imaging pipeline described in [Pol21] to safe Rust and parallel Rust using the Rayon library. The goal was to evaluate whether Rust's safety guarantees and high-level parallelism ecosystem can provide an accessible path towards high-performance scientific software on modern multi-core CPUs.
To answer this question, two new implementations of the pipeline were developed: a fully safe sequential Rust implementation and a parallel implementation based on Rayon. Subsequently, the performance of the Rayon implementation was evaluated on DAS-6[BEdL+16] and compared against the ImageJ, DIPlib, JAX, Triton, and CUDA implementations previously studied in [dBeaed].
The results demonstrate that Rayon is capable of delivering significant performance improvements over existing CPU-based implementations. Across all tested image sizes, the Rayon implementation consistently outperformed both ImageJ and DIPlib. Furthermore, Rayon achieved better overall performance than the GPU-based JAX implementation despite running entirely on the CPU. Although CUDA remained significantly faster overall, Rayon became competitive with Triton for larger image sizes and slightly outperformed Triton on datasets with the largest images. This suggests that Rust's lower-level control over memory management allows it to scale well for memory-intensive workloads.
At the pipeline-stage level, the performance gains varied across the different stages. The embarrassingly parallel stages, such as Minimum Projection and Thresholding, benefited substantially from Rayon's data-parallel programming model, yielding significant improvements over the other CPU-based implementations. Nevertheless, these stages remained considerably slower than the GPU-based implementations, highlighting the much higher degree of parallelism available on modern GPUs. In contrast, the fill holes and watershed stages achieved performance competitive with, and in some cases exceeding, the Triton implementation. These results demonstrate that efficient algorithmic design and low-level implementation optimizations can allow a CPU implementation to compete with GPU solutions for certain workloads. The Euclidean Distance Transform, however, remained the primary bottleneck of the pipeline, suggesting that improving the underlying algorithm and memory access patterns is likely to provide larger performance gains than additional parallelization alone.

From a software engineering perspective, the added development effort required to introduce Rayon to the existing Rust code-base was relatively small. The parallel implementation required only a modest increase in code size compared to the sequential Rust implementation, while yielding substantial performance improvements. Much of the parallelization could be expressed through Rayon's parallel iterator abstractions, allowing large portions of the sequential implementation to be converted into parallel code with limited modifications. More advanced optimizations were only necessary for a small number of performance-critical stages.
These findings suggest that Rust and Rayon provide competitive CPU-parallel performance, at the cost of greater initial implementation complexity. While Rust and Rayon cannot match the raw throughput of highly optimized and memory-efficient GPU implementations, they consistently outperform the evaluated CPU-based implementations and even exceed the performance of JAX and, for the largest evaluated image sizes, Triton, while preserving memory safety guarantees and requiring substantially less expertise in low-level parallel programming techniques compared to traditional low-level CPU and GPU programming frameworks. Consequently, Rust and Rayon appear to be viable technologies for scientific and high-performance computing projects targeting multi-core CPUs, particularly where development effort, portability, and correctness are important considerations.

Several opportunities for future work remain, such as implementing possible leftover optimizations to improve the Rayon port. The most immediate optimization opportunity is replacing the existing Euclidean Distance Transform algorithm adapted from [FH12] with the more efficient algorithm used for the DIPlib implementation.
Future work could also investigate other Rust parallelism frameworks, such as Velvet [ea26], to determine whether the gained performance using Rayon can be eclipsed by leveraging the additional performance gains that are possible for divide-and-conquer workloads. Finally, it would be interesting to evaluate Rust-based GPU programming frameworks such as `rust-cuda` [rus] and `rust-gpu` [tea] in order to determine whether the benefits of Rust's safety guarantees can be extended to high-performance GPU implementations as well.

References

[APZ+24] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C Jeffrey. When is parallelism fearless and zero-cost with rust? In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 27–40, 2024.

[AVDVU08] François Aguet, Dimitri Van De Ville, and Michael Unser. Model-based 2.5-d deconvolution for extended depth of field in brightfield microscopy. *IEEE Transactions on Image Processing*, 17(7):1144–1153, 2008.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BEdL+16] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(05):54–63, may 2016.

[CISACNN23] Cybersecurity, Federal Bureau of Investigation (FBI) Australian Cyber Security Centre (ACSC) Canadian Centre for Cyber Security (CCCS) National Cyber Security Centre–United Kingdom (NCSC-UK) Infrastructure Security Agency (CISA), National Security Agency (NSA), and New Zealand National Cyber Security Centre (NCSC-NZ). The case for memory safe roadmaps: Why both c-suite executives and technical experts need to take memory safe coding seriously. 2023.

[con26] DIPlib contributors. diplib.org. https://diplib.org/, June 2026. [Accessed 17-07-2026].

[CPU] AMD EPYC 7282 Specs — techpowerup.com. https://www.techpowerup.com/cpu-specs/epyc-7282.c2255. [Accessed 21-07-2026].

[dBeaed] Wouter J. Van den Berg et al. Comparing GPU abstractions by Investigating the Performance-Effort Trade-off in Imaging Pipelines. *LIACS, Leiden University*, Unreleased (In the process of being Peer reviewed).

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[DSS+23] Constanze Depp, Ting Sun, Andrew Octavian Sasmita, Lena Spieth, Stefan A Berghoff, Taisiia Nazarenko, Katharina Overhoff, Agnes A Steixner-Kumar, Swati Subramanian, Sahab Arinrad, et al. Myelin dysfunction drives amyloid-β deposition in models of alzheimer's disease. *Nature*, 618(7964):349–357, 2023.

[DSS+24] Manyu Du, Simon Hendrik Stitzinger, Jan-Hendrik Spille, Won-Ki Cho, Choongman Lee, Mohammed Hijaz, Andrea Quintana, and Ibrahim I Cissé. Direct observation of a condensate effect on super-enhancer controlled gene bursting. *Cell*, 187(2):331–344, 2024.

[ea] AlDanial et al. GitHub - AlDanial/cloc: cloc – count lines of code — github.com. https://github.com/AlDanial/cloc. [Accessed 17-07-2026].

[ea25] J. Stone et al. Rayon. https://github.com/rayon-rs/rayon, 2025. [Accessed 22-09-2025].

[ea26] B. Liokouras et al. Velvet: Parallel Divide-and-Conquer in Safe Rust. *accepted for publication in Europar'26*, 2026.

[Esh19] Eliot Eshelman. Detailed Specifications of the AMD EPYC "Rome" CPUs - Microway — microway.com. https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-amd-epyc-rome-cpus/, 2019. [Accessed 21-07-2026].

[FH12] Pedro F Felzenszwalb and Daniel P Huttenlocher. Distance transforms of sampled functions. *Theory of computing*, 8(1):415–428, 2012.

[FJL19] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *SysML conference 2018*, 2019.

[GAO+16] Rogério Gonçalves, Marcos Amaris, Thiago Okada, Pedro Bruel, and Alfredo Goldman. Openmp is not as easy as it appears. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5742–5751. IEEE, 2016.

[joi26] Rayon join function documentation — docs.rs. https://docs.rs/rayon/latest/rayon/fn.join.html, 2026. [Accessed 01-07-2026].

[K+07] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[KN23] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[LPC+23] Chen Li, Marcel Polling, Lu Cao, Barbara Gravendeel, and Fons J Verbeek. Analysis of automatic image classification methods for Urticaceae pollen classification. *Neurocomputing*, 522:181–193, 2023.

[MFH+25] Eduardo M Martins, Leonardo G Faé, Renato B Hoffmann, Lucas S Bianchessi, and Dalvan Griebler. Npb-rust: Nas parallel benchmarks in rust. *arXiv preprint arXiv:2502.15536*, 2025.

[NNDS22] Tatjana R Nikolić, Goran S Nikolić, Bojan R Dimitrijević, and Mile K Stojcev. From single cpu to multicore systems. In *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 1–8. IEEE, 2022.

[NVI23] NVIDIA. Nvidia rtx 4000 ada generation datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/products/workstations/nvidia-rtx-4000-datasheet.pdf, 2023. [Accessed 21-07-2026].

[OL10] Victor MA Oliveira and Roberto A Lotufo. A study on connected components labeling algorithms using gpus. In *Workshop of Undergraduate Works, XXIII Sibgrapi, Conference on Graphics, Patterns and Images*, volume 9, 2010.

[par26a] Rayon ParallelIterator `filter_map` method documentation — docs.rs. https://docs.rs/rayon/latest/rayon/iter/trait.ParallelIterator.html#method.filter_map, 2026. [Accessed 01-07-2026].

[par26b] Rayon ParallelIterator `for_each` method documentation — docs.rs. https://docs.rs/rayon/latest/rayon/iter/trait.ParallelIterator.html#method.for_each, 2026. [Accessed 01-07-2026].

[PLH+21] Ricardo Pieper, Júnior Löff, Renato B Hoffmann, Dalvan Griebler, and Luiz G Fernandes. High-level and efficient structured stream parallelism for rust on multi-cores. *Journal of Computer Languages*, 65:101054, 2021.

[Pol21] M. et al. Polling. Neural networks for increased accuracy of allergenic pollen monitoring. *Sci Rep 11, 11357 (2021). https://doi.org/10.1038/s41598-021-90433-x*, 2021.

[RC+78] Thomas Wilhelm Ridler, S Calvard, et al. Picture thresholding using an iterative selection method. *IEEE Trans. Syst. Man Cybern*, 8(8):630–632, 1978.

[RK24] Alex Rebert and Christoph Kern. Secure by design: Google's perspective on memory safety. *Google Security Engineering Technical Report. Google*, 2024.

[RT06] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, 2006.

[rus] GitHub - Rust-GPU/rust-cuda: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust. — github.com. https://github.com/Rust-GPU/rust-cuda. [Accessed 17-07-2026].

[SRE12] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. Nih image to imagej: 25 years of image analysis. *Nature methods*, 9(7):671–675, 2012.

[Sta16] William Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 2016.

[Sud20] Michal Sudwoj. Rust programming language in the high-performance computing environment. 2020.

[SYK+22] Rigel Suzuki, Daichi Yamasoba, Izumi Kimura, Lei Wang, Mai Kishimoto, Jumpei Ito, Yuhei Morioka, Naganori Nao, Hesham Nasser, Keiya Uriu, et al. Attenuated fusogenicity and pathogenicity of sars-cov-2 omicron variant. *Nature*, 603(7902):700–705, 2022.

[tea] Rust GPU team. Rust GPU — rust-gpu.github.io. https://rust-gpu.github.io/. [Accessed 17-07-2026].

[TFea26] Tiago Ferreira et al. ImageJ2 — imagej.net. https://imagej.net/software/imagej2/, 2026. [Accessed 02-07-2026].

[tif26] GitHub - image-rs/image-tiff: TIFF decoding and encoding library in pure Rust — github.com. https://github.com/image-rs/image-tiff, 2026. [Accessed 02-07-2026].

[TKC19] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[wim26] wim. tiffwrite — git.pomppervova.nl. https://git.pomppervova.nl/wim/tiffwrite, 2026. [Accessed 02-07-2026].

[YV18] Varduhi Yeghiazaryan and Irina Voiculescu. Path reducing watershed for the gpu. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 577–585. IEEE, 2018.