

Technische Hochschule Deggendorf
Fakultät Angewandte Informatik
Studiengang Master Angewandte Informatik

VERGLEICHENDE EVALUIERUNG VON
BILDVERARBEITUNGSBIBLIOTHEKEN FÜR
INDUSTRIELLE ANWENDUNGEN BEI DASSAULT
SYSTEMS

COMPARATIVE EVALUATION OF IMAGE
PROCESSING LIBRARIES FOR INDUSTRIAL
APPLICATIONS AT DASSAULT SYSTEMS

Masterarbeit zur Erlangung des akademischen Grades:

Master of Engineering (M.Eng.)

an der Technischen Hochschule Deggendorf

Vorgelegt von:
Sepehr Fazeli Shahroudi
Matrikelnummer: 12200627

Prüfungsleitung:
Prof. Dr. Schober

Am: 24. Mar 2025

Ergänzende Prüfende:
Martin Steglich

Erklärung

Name des Studierenden: Sepehr Fazeli Shahroudi

Name des Betreuenden: Prof. Dr. Schober

Thema der Abschlussarbeit:

Vergleichende Evaluierung von Bildverarbeitungsbibliotheken für industrielle Anwendungen bei Dassault Systems

.....
.....

1. Ich erkläre hiermit, dass ich die Abschlussarbeit gemäß § 35 Abs. 7 RaPO (Rahmenprüfungsordnung für die Fachhochschulen in Bayern, BayRS 2210-4-1-4-1-WFK) selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Deggendorf,
Datum

.....
Unterschrift des Studierenden

2. Ich bin damit einverstanden, dass die von mir angefertigte Abschlussarbeit über die Bibliothek der Hochschule einer breiteren Öffentlichkeit zugänglich gemacht wird:

- ☐ Nein
☐ Ja, nach Abschluss des Prüfungsverfahrens
☐ Ja, nach Ablauf einer Sperrfrist von ... Jahren.

Deggendorf,
Datum

.....
Unterschrift des Studierenden

Bei Einverständnis des Verfassenden vom Betreuenden auszufüllen:

Eine Aufnahme eines Exemplars der Abschlussarbeit in den Bestand der Bibliothek und die Ausleihe des Exemplars wird:

- ☐ Befürwortet
☐ Nicht befürwortet

Deggendorf,
Datum

.....
Unterschrift des Betreuenden

Contents

Abstract	1
1 Introduction	3
1.1 The Significance of Image Processing in Modern Industry	3
1.1.1 Evolution and Impact of Digital Image Processing	4
1.1.2 Current Applications of Image Processing in Industry	5
1.1.3 The Strategic Importance of Image Processing Libraries	6
1.2 Aim of the Study and Its Implications for Selecting an Image Processing Tool .	8
1.2.1 Research Goals and Objectives	8
1.2.2 Practical Implications for Tool Selection	9
1.3 Related Work	10
2 Methodology	13
2.1 Selection of Libraries for Comparison	13
2.2 Performance Metrics and Criteria for Comparison	14
2.2.1 Defining the Image Conversion Metric	15
2.2.2 Defining the Pixel Iteration Metric	15
2.2.3 Criteria for Library Comparison	16
2.3 Experimental Setup and Environment	17
2.4 Data Collection and Processing	17
2.5 Conclusion	18
3 Implementation	19
3.1 System Architecture and Design Rationale	19
3.2 Benchmarking Implementation	21
3.2.1 Image Conversion Benchmark Implementation	21
3.2.2 Pixel Iteration Benchmark Implementation	22
3.3 Libraries Implementation	25
3.3.1 OpenCvSharp and SkiaSharp Implementation	25
3.3.2 Magick.NET Implementation	29
3.3.3 Emgu CV and Structure.Sketching Implementation	31
3.4 Memory Profiling and Performance Analysis	33
3.5 Result Export and Data Aggregation	35
4 Results	39
4.1 Image Conversion Benchmark Results	39
4.2 Pixel Iteration Benchmark Results	41
4.3 Memory Benchmarking Results	43

4.4	Analysis and Interpretation of Results	44
4.4.1	Comparison of Performance Trends	44
4.4.2	Trade-Offs Between Speed and Memory Usage	45
4.5	Summary	46
5	Discussion	47
5.1	Interpreting the Results: Performance vs. Practicality	47
5.1.1	Performance Trade-offs and Suitability for Real-World Applications . .	47
5.1.2	The Impact of Licensing on Library Selection	47
5.2	Strengths and Weaknesses of the Different Libraries	48
5.3	Considerations for Future Research	49
5.4	Closing Thoughts	50
6	Appendix	51

Abstract

Comparing different image processing libraries is an important job that has to be done to find a solution which is on the one hand cost-effective and on the other hand high-performing that would be appropriate for the industrial and software applications. Automation, quality control, medical imaging, and real-time data analysis are the use cases involving digital image processing, and it requires the libraries that can give both efficiency and elasticity. Whether they are open-source or proprietary, the libraries such as OpenCV, SkiaSharp, Magick.NET, Emgu CV, and OpenCvSharp give a wide spectrum of functionalities, from simple image manipulations to complex computer vision problems.

The choice of a graphics programming library that is applicable to a given task is primarily influenced by several factors (the first of which is) the computational efficiency used in the algorithms, as well as licensing, and integration concepts as well. Some libraries take into account the speed and flexibility and thus cater to performance-critical applications whereas others are more centered on the ease of use and support for different platforms. To test the performance of libraries, you will be needed to measure their run-time, memory usage, and requirements for different environments, including embedded systems, desktop applications, and cloud-based platforms.

A key consideration in the industrial context is the trade-off between the processing power and the stability in the setting of the operation. On the one hand, with the help of the high-performance computers, the image transformation can be done in a flash and the real-time analysis can be done but, on the other hand, embedded systems and industrial controllers always set limits that make the execution slow. It is crucial to make the necessary trade-offs between these three in the case of such companies, which approach the issue of image processing optimization by factoring in reliability and cost efficiency.

The research report is a comprehensive analysis of the image processing libraries. During the course of the project, the main strengths, weaknesses, as well as the best-use scenarios are seen. The results compile efficiency measurements, integration difficulties, and cost-related issues, offering a practical guide not only for software developers but also for hardware engineers and decision-makers who would like to principally survive the expanding market of image processing solutions in any type of industry across the world.

1 Introduction

1.1 The Significance of Image Processing in Modern Industry

Digital image processing has emerged as a cornerstone of modern industrial applications, revolutionizing the way industries operate and innovate. From quality control in manufacturing to advanced simulations in aerospace, the ability to process and analyze images digitally has unlocked unprecedented efficiencies and capabilities. This field, which involves the manipulation and analysis of images using algorithms, has evolved significantly over the past few decades, driven by advancements in computing power, algorithm development, and the proliferation of digital imaging devices [1, 2].

The significance of digital image processing in industrial applications cannot be overstated. In manufacturing, for instance, image processing is integral to quality assurance processes, where it is used to detect defects, measure product dimensions, and ensure compliance with stringent standards. This capability not only enhances product quality but also reduces waste and operational costs. In the automotive industry, image processing is pivotal in the development of autonomous vehicles, where it aids in object detection, lane departure warnings, and pedestrian recognition. Similarly, in the healthcare sector, digital image processing is used in medical imaging technologies such as MRI and CT scans, enabling more accurate diagnoses and treatment planning [3, 4].

The evolution of digital image processing has been marked by several key developments. Initially, the field was limited by the computational resources available, with early applications focusing on basic image enhancement and restoration. However, the advent of powerful processors and the development of sophisticated algorithms have expanded the scope of image processing to include complex tasks such as pattern recognition, 3D reconstruction, and real-time image analysis. The integration of artificial intelligence and machine learning has further propelled the field, allowing for the development of intelligent systems capable of learning from data and improving over time [1, 4, 5].

For industries like Dassault Systems, which operates at the forefront of aerospace, defense, and industrial engineering, a comparative study of image processing libraries is crucial. These libraries, which provide pre-built functions and tools for image analysis, vary significantly in terms of performance, ease of use, and functionality. Selecting the right library can have a profound impact on the efficiency and effectiveness of image processing tasks. For instance, libraries such as OpenCV, ImageMagick and ImageSharp offer different strengths and weaknesses, and understanding these differences is essential for optimizing industrial applications [6].

A comparative study of these libraries not only aids in selecting the most suitable tools for specific tasks but also highlights areas for potential improvement and innovation. By analyzing the performance of different libraries in various scenarios, industries can identify gaps in

current technologies and drive the development of new solutions that better meet their needs. Moreover, such studies contribute to the broader field of digital image processing by providing insights into best practices and emerging trends.

1.1.1 Evolution and Impact of Digital Image Processing

Digital image processing has evolved significantly since its inception, transforming from a niche scientific endeavor into a cornerstone of modern technology with applications spanning numerous industries. This essay outlines the historical development of digital image processing, highlighting key advancements and their impact on industrial innovation.

Early Beginnings

The origins of digital image processing can be traced back to the 1920s and 1930s with the development of television technology, which laid the groundwork for electronic image capture and transmission. However, it wasn't until the 1960s that digital image processing began to take shape as a distinct field. The launch of the first digital computers provided the necessary computational power to process images digitally. During this period, NASA played a pivotal role by using digital image processing to enhance images of the moon's surface captured by the Ranger 7 spacecraft in 1964. This marked one of the first significant applications of digital image processing, demonstrating its potential for scientific and exploratory purposes [7].

The 1970s and 1980s: Theoretical Foundations and Practical Applications

The 1970s saw the establishment of theoretical foundations for digital image processing. Researchers developed algorithms for image enhancement, restoration, and compression. The Fast Fourier Transform (FFT), introduced by Cooley and Tukey in 1965, became a fundamental tool for image processing, enabling efficient computation of image transformations. This period also witnessed the development of the first commercial applications, such as medical imaging systems. The introduction of Computed Tomography (CT) in 1972 revolutionized medical diagnostics by providing detailed cross-sectional images of the human body, showcasing the life-saving potential of digital image processing [7, 8].

The 1990s: The Rise of Computer Vision

The 1990s marked a significant shift towards computer vision, a subfield of digital image processing focused on enabling machines to interpret visual data. This era saw the development of algorithms for object recognition, motion detection, and 3D reconstruction. The introduction of the JPEG standard in 1992 facilitated the widespread adoption of digital images by providing an efficient method for image compression, crucial for the burgeoning internet era. The decade also saw advancements in facial recognition technology, which laid the groundwork for future applications in security and personal identification [9].

The 2000s: Machine Learning and Image Processing

The 2000s witnessed the integration of machine learning techniques with digital image processing, leading to significant improvements in image analysis and interpretation. The development of Support Vector Machines (SVM) and neural networks enabled more accurate image classification and pattern recognition. This period also saw the emergence of digital cameras and smartphones, which democratized image capture and sharing, further driving the demand for advanced image processing techniques[9].

The 2010s to Present: Deep Learning and Industrial Innovation

The advent of deep learning in the 2010s revolutionized digital image processing. Convolutional Neural Networks (CNNs), popularized by the success of AlexNet in the ImageNet competition in 2012, dramatically improved the accuracy of image recognition tasks. This breakthrough spurred innovation across various industries. In healthcare, deep learning algorithms are now used for early detection of diseases through medical imaging, improving patient outcomes. In the automotive industry, image processing is a critical component of autonomous vehicle systems, enabling real-time object detection and navigation [10, 9].

In recent years, digital image processing has expanded into areas such as augmented reality (AR) and virtual reality (VR), enhancing user experiences in gaming, education, and training. The integration of image processing with artificial intelligence continues to drive innovation, with applications in fields such as agriculture, where drones equipped with image processing capabilities monitor crop health and optimize yields.

1.1.2 Current Applications of Image Processing in Industry

Image processing, a critical component of computer vision, has become an indispensable tool across various industries, driving advancements in productivity, quality control, and automation. This essay explores the utilization of image processing in several key sectors, emphasizing applications that demand high precision and efficiency.

Manufacturing and Quality Control

In the manufacturing industry, image processing is pivotal for quality control and defect detection. Automated visual inspection systems utilize high-resolution cameras and sophisticated algorithms to detect defects in products at a speed and accuracy unattainable by human inspectors. For instance, in semiconductor manufacturing, image processing is used to inspect wafers for defects, ensuring that only flawless products proceed to the next production stage. This not only enhances product quality but also reduces waste and operational costs. A study by Zhang et al. (2023) [11] highlights the use of convolutional neural networks (CNNs) in detecting surface defects in steel manufacturing, demonstrating significant improvements in detection accuracy and processing speed compared to traditional methods.

Healthcare and Medical Imaging

In healthcare, image processing is revolutionizing diagnostics and treatment planning. Techniques such as MRI, CT scans, and X-rays rely heavily on image processing to enhance image quality and extract meaningful information. For example, in radiology, image processing algorithms help in the early detection of diseases like cancer by improving the clarity and contrast of medical images, allowing for more accurate diagnoses. A research paper by Litjens et al. (2017) [12] reviews the application of deep learning in medical imaging, showcasing its potential in improving diagnostic accuracy and efficiency, thus influencing patient outcomes positively.

Agriculture

Precision agriculture benefits significantly from image processing, where it is used for crop monitoring, disease detection, and yield estimation. Drones equipped with multispectral cameras capture images of fields, which are then processed to assess plant health and detect stress factors such as pests or nutrient deficiencies. This enables farmers to make informed decisions, optimizing resource use and improving crop yields. A case study by Maimaitijiang et al. (2020) [13] demonstrates the use of UAV-based hyperspectral imaging for monitoring crop growth, highlighting its effectiveness in enhancing agricultural productivity.

Automotive Industry

In the automotive sector, image processing is integral to the development of autonomous vehicles. Advanced driver-assistance systems (ADAS) rely on image processing to interpret data from cameras and sensors, enabling features such as lane departure warnings, adaptive cruise control, and automatic parking. These systems enhance vehicle safety and pave the way for fully autonomous driving. A study by Janai et al. (2021) [14] discusses the role of computer vision in autonomous vehicles, emphasizing the importance of real-time image processing in ensuring safe and efficient vehicle operation.

Retail and E-commerce

Retail and e-commerce industries leverage image processing for inventory management, customer analytics, and personalized marketing. In inventory management, image processing systems track stock levels and identify misplaced items, streamlining operations and reducing labor costs. In customer analytics, facial recognition and sentiment analysis provide insights into customer behavior and preferences, enabling personalized marketing strategies. A paper by Ren et al. (2016) [15] explores the application of image processing in retail, highlighting its impact on enhancing customer experience and operational efficiency.

1.1.3 The Strategic Importance of Image Processing Libraries

In the rapidly evolving landscape of industrial applications, the demand for efficient, adaptable, and scalable image processing libraries has become increasingly critical. These libraries serve as the backbone for a myriad of applications ranging from quality control in manufacturing

1.1 The Significance of Image Processing in Modern Industry

to advanced robotics and autonomous systems. The benefits of employing such libraries are manifold, including reduced time-to-market, enhanced product quality, and cost efficiency, all of which are pivotal for maintaining competitive advantage in the industrial sector.

Firstly, efficient image processing libraries significantly reduce the time-to-market for new products and technologies. In industries where innovation cycles are short and competition is fierce, the ability to quickly develop and deploy new solutions is crucial. Efficient libraries streamline the development process by providing pre-built, optimized functions that developers can readily integrate into their systems. This reduces the need for writing complex algorithms from scratch, thereby accelerating the development timeline. For instance, libraries like OpenCV and TensorFlow offer a wide array of tools and functions that can be easily adapted to specific industrial needs, allowing companies to focus on innovation rather than the intricacies of image processing [6].

Adaptability is another critical factor that underscores the importance of these libraries. Industrial environments are often dynamic, with varying requirements and conditions that necessitate flexible solutions. Scalable image processing libraries can be tailored to meet specific needs, whether it involves adjusting to different hardware configurations or integrating with other software systems. This adaptability ensures that companies can respond swiftly to changes in market demands or technological advancements without overhauling their entire system architecture. For example, the modular nature of libraries like Halide allows for easy customization and optimization for different hardware platforms, enhancing their applicability across diverse industrial scenarios [16].

Moreover, the use of scalable image processing libraries contributes to enhanced product quality. In industries such as automotive manufacturing or pharmaceuticals, precision and accuracy are paramount. Advanced image processing capabilities enable more rigorous quality control processes, ensuring that defects are detected and rectified early in the production cycle. This not only improves the quality of the final product but also minimizes waste and reduces the likelihood of costly recalls. Studies have shown that implementing robust image processing solutions can lead to significant improvements in defect detection rates and overall product reliability [17].

Cost efficiency is another significant advantage offered by these libraries. By leveraging open-source or commercially available image processing tools, companies can reduce the costs associated with software development and maintenance. These libraries often come with extensive documentation and community support, which can further reduce the need for specialized training and technical support. Additionally, the ability to scale solutions according to demand means that companies can optimize their resource allocation, investing only in the capabilities they need at any given time. This scalability is particularly beneficial for small and medium-sized enterprises that may not have the resources to develop custom solutions from the ground up [18].

1.2 Aim of the Study and Its Implications for Selecting an Image Processing Tool

The purpose of this study was to compare the performance, functionality, and ease of integration of a wide range of image processing libraries. The primary objective is to establish a general framework for evaluating different tools in the field. As part of this research, key metrics such as conversion speed, pixel iteration efficiency, memory consumption, and development effort will be evaluated in order to provide developers, engineers, and decision-makers with a balanced viewpoint.

1.2.1 Research Goals and Objectives

At its core, the study sought to answer the question: “Which image processing library best meets the diverse needs of modern applications?” To do so, several key objectives were identified:

1. **Provide a Framework for Educated Choices:** The research aimed to create a framework that helps users evaluate image processing tools based on defined metrics. By comparing factors such as processing speed, memory consumption, development effort, and integration ease, the study aimed to demystify the trade-offs that come with adopting any given tool. This approach allows users to align their choices with their performance needs and project constraints, rather than making decisions solely based on cost considerations. As highlighted in the investigation, while saving on licensing fees is beneficial, the broader picture includes aspects like processing efficiency and long-term maintainability.
2. **Compare a Wide Range of Alternatives:** ImageSharp is one of many tools available for image processing. The study examined alternatives including OpenImageIO, SkiaSharp, Magick.NET, Emgu CV, MagicScaler, and several others. Each library was assessed against a set of criteria, such as its ability to handle tasks like image loading, pixel manipulation, resizing, and image composition. By comparing these libraries side-by-side, the study provides a nuanced view that helps practitioners understand not only what each tool can do but also the potential gaps that might exist depending on the application’s requirements.
3. **Define Clear Performance and Functional Metrics:** A significant goal of the study was to establish quantifiable metrics that could be used to assess the performance of each image processing library. Metrics such as image conversion time, pixel iteration efficiency, and memory usage were used as benchmarks. For instance, the study measured how long it takes for a tool to load an image, perform a conversion (e.g., from JPEG to PNG), and iterate through pixels for operations like converting to grayscale. Such detailed benchmarking is instrumental in understanding the real-world performance of each library and is critical for users who need to balance speed with resource consumption.

4. **Assist in Tool Selection for Varied Requirements:** Beyond performance metrics, the study was designed to consider the broader context of software integration. Factors such as ease of implementation, the learning curve for developers, compatibility with existing systems, and community support were all taken into account. This holistic view means that the research is not just about raw performance numbers but also about the practicalities of deploying and maintaining these tools in production environments.

1.2.2 Practical Implications for Tool Selection

The comprehensive evaluation detailed in this study has several practical implications for anyone looking to select an image processing tool:

Balancing Performance with Practicality

The metrics established in the study—ranging from processing times to memory usage—provide a clear picture of the strengths and weaknesses of each library. This information is invaluable when balancing the need for high-performance image processing against practical considerations such as ease of integration and long-term maintenance. For instance, a company that prioritizes rapid image conversion and low memory consumption might lean towards SkiaSharp, while an organization needing advanced image manipulation capabilities and robust community support might find Emgu CV more appealing.

Making Informed Trade-Offs

One of the standout contributions of the study is its ability to help users make informed trade-offs. Rather than making decisions based on a single metric, the evaluation presents a multi-dimensional view that incorporates performance, development effort, and functional capabilities. This approach ensures that users can select a tool that best fits their unique requirements, whether that means prioritizing speed, minimizing development overhead, or ensuring compatibility with existing workflows.

Extending Beyond Cost Savings

While cost savings are certainly a factor, the study underscores that financial considerations should not be the sole driver of decision-making. The true value of an image processing tool lies in its ability to meet specific technical and operational requirements. By providing a detailed comparison of several alternatives, the research emphasizes that factors like ease of integration, scalability, and overall performance are equally, if not more, important. This holistic approach helps organizations avoid the pitfall of selecting a tool based solely on its cost.

Guiding Future Developments and Integrations

The insights gained from the study are not only applicable to current technology choices but also serve as a guide for future developments in image processing. The detailed benchmarks and performance analyses can inform future projects, helping developers understand where

improvements can be made or which features are most critical. Additionally, the study’s approach to evaluating development effort and integration challenges provides a roadmap for how future research can build on these findings to further refine the selection process.

1.3 Related Work

The evaluation of image processing libraries, particularly for industrial applications, has attracted significant research interest over the past decades. Broadly, the field encompasses automated image analysis and computer vision systems designed to handle tasks such as quality control, defect detection, and high-resolution image enhancement. The foundational research in Automated Image Processing has evolved from early, often ad hoc, implementations to sophisticated frameworks that leverage hardware acceleration and advanced algorithms. Early surveys, such as Kulpa’s (1981) [19] seminal review of digital image processing systems in Europe, laid the groundwork for understanding the challenges of standardization and performance evaluation in these systems.

In recent years, the convergence of hardware acceleration and image analysis has been a recurring theme. Sahebi et al. (2023) [20] demonstrate how distributed processing on FPGAs can dramatically enhance computational efficiency—a principle equally applicable to industrial image processing where real-time performance is critical. Similarly, Ma et al. (2024) [21] contribute to the field by presenting an image quality database specifically tailored for industrial processes. Their work emphasizes the importance of aligning objective metrics with human perception in quality assessments, a concern that resonates throughout subsequent research in the area.

Chisholm et al. (2020) [22] and Ferreira et al. (2024) [23] extend these discussions by focusing on the implementation of real-time image processing systems using FPGAs. Chisholm illustrate a real-time crack detection system employing particle filters, highlighting the challenges of meeting stringent timing constraints in industrial settings. Ferreira, on the other hand, propose a generic FPGA-based pre-processing library, emphasizing strategies to minimize memory overhead and improve processing speed. These studies underscore the significant role of hardware acceleration in modern image processing pipelines, setting the stage for more nuanced comparative evaluations.

A critical aspect of the research is the comparative analysis of different image processing libraries. Lai et al. (2001) [24] provide an in-depth review of several libraries, contrasting hardware-specific optimizations with generic, portable solutions. Their work not only identifies the strengths and weaknesses inherent in different design philosophies but also serves as a benchmark against which later approaches can be compared. Kulpa’s early survey (1981) [19] remains an important historical reference, offering insights into the evolution of image processing systems and highlighting persistent issues such as limited standardization and documentation.

Pérez et al. (2014) [25] contribute by investigating super-resolution techniques for plenoptic cameras via FPGA-based implementations, demonstrating that hardware acceleration can significantly improve both processing speed and image quality. Meanwhile, Rao’s (2023) [26] comparative analysis of deep learning frameworks extends the conversation by incorporat-

ing performance metrics, documentation quality, and community support. This approach is particularly valuable as it parallels the metrics used to evaluate traditional image processing libraries, thereby bridging the gap between classical image processing and modern deep learning paradigms.

Several studies have explored niche industrial applications where image processing plays a critical role. Ciora and Simion (2014) [27] provide a broad overview of the applications of image processing in industrial engineering, covering areas from automated visual inspection to process control. Their comprehensive review underscores the necessity of robust, efficient image processing systems that integrate seamlessly with industrial control mechanisms.

In a more focused domain, Sandvik et al. (2024) [28] review machine learning and image processing techniques for wood log scaling and grading. Their systematic categorization of methodologies offers a template for benchmarking approaches that combine computer vision with domain-specific performance metrics. Sardar (2012) [29] examines the use of image processing for quality analysis in agriculture, further highlighting the versatility of these technologies across different industrial sectors.

Vieira et al. (2024) [30] address the challenges of deploying image processing algorithms on Programmable Logic Controllers (PLCs), which are prevalent in industrial control systems. Their work illustrates the trade-offs between processing speed, implementation complexity, and system robustness when operating in resource-constrained environments. Wu et al. (2022) [31] and Zhu et al. (2022) [32] then delve into specific industrial applications—precision control in filament drafting and product appearance quality inspection, respectively—demonstrating the critical impact of real-time processing and integration on system performance.

At the forefront of current research are studies that provide robust benchmarking frameworks. Reis (2023) [33] offers an overview of recent developments in computer vision and image processing methodologies, pointing out the increasing integration of artificial intelligence with classical approaches. This evolution is complemented by Ziaja et al. (2021) [34], whose work on benchmarking deep learning for on-board space applications provides a rigorous framework for evaluating execution time, resource utilization, and overall performance under constrained hardware conditions.

These contemporary evaluations are essential for highlighting the limitations of existing approaches. While many studies focus on performance metrics such as processing speed and memory efficiency, few have systematically integrated these factors with ease of integration and system robustness in industrial settings. This gap in the literature motivates the present study, which aims to establish a comprehensive benchmarking approach that encompasses both hardware acceleration and software flexibility.

In summary, the reviewed literature presents a rich tapestry of methodologies and evaluations that span a broad spectrum of industrial image processing applications. Early foundational works provided historical context and identified critical challenges, while subsequent studies advanced the field by integrating hardware acceleration, deep learning, and niche industrial applications into comprehensive performance evaluations. Despite these advances, a clear gap remains in the standardization of benchmarking protocols that address performance, resource efficiency, and integration challenges in real-world industrial settings. This thesis proposes a novel benchmarking approach that differentiates itself by not only comparing the computational performance of various image processing libraries but also by evaluating their

ease of integration into complex industrial workflows. By doing so, the study seeks to provide actionable insights for practitioners and pave the way for the next generation of robust, efficient, and versatile image processing solutions.

2 Methodology

This chapter outlines the journey and rationale behind the methodology for comparing various image processing libraries. It explains the choice of performance metrics, describes how the metrics were obtained and processed and details the criteria used to select the libraries under investigation. The aim is to provide an approach that not only yields quantitative insights but also connects with real-world applications.

2.1 Selection of Libraries for Comparison

The choice of libraries for this study was driven by several factors, including functionality, licensing, ease of integration, and performance potential. Most of image processing libraries provided wrappers or bindings for .NET, the language of choice for this experiments. The search of libraries revealed a wide range of options—from the commercial ImageSharp to various open-source alternatives such as OpenCvSharp, Emgu CV, SkiaSharp, Magick.NET, and others.

With consideration of real-world image processing applications needs, certain technical features were considered essential for the evaluation, such as support for common image formats (JPEG, PNG, BMP, WebP, etc.), mutative operations (e.g., pixel manipulation, color space conversion), and high-level operations (e.g., image composition, filtering). All libraries were evaluated based on their ability to handle these tasks efficiently by inspecting their APIs and documentation. Also the licensing model, integration effort, and community support were considered to ensure that the selected libraries were not only technically capable but also practical for real-world applications. The data gathered from this is available including the table of feature comparison that was created for each library, and available in the appendix (see Chapter 6).

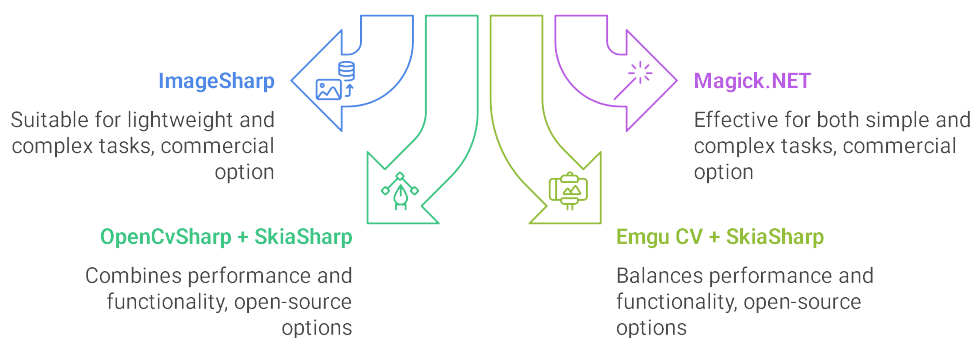


Figure 2.1: This figure shows the selected libraries for comparison and their main features for each or combination of libraries.

As a result of this research, a clear picture of each library’s capabilities was developed, and the most suitable candidates for the performance tests were identified. Consequently, 5 suggested libraries or combinations of libraries were selected for the comparative evaluation: ImageSharp and Magick.NET as single library solutions, given their capabilities to cover both lightweight and complex image processing tasks, and the combinations of OpenCvSharp with SkiaSharp, and Emgu CV with SkiaSharp, as they complement each other in terms of performance and functionality.

2.2 Performance Metrics and Criteria for Comparison

Image processing is an integral part of many modern applications, from web services to real-time computer vision systems. To compare libraries by comparing their performance and practicality using a controlled benchmarking environment. The study focused on two key performance metrics: **Image Conversion** and **Pixel Iteration**. These metrics were selected because they represent foundational operations in image processing workflows, forming the building blocks for more complex tasks.

The decision to focus on image conversion and pixel iteration was based on the need for metrics that could objectively and quantitatively measure core operations while remaining independent of higher-level library-specific features. Image conversion was chosen as it involves loading an image from disk, converting its format, and saving it back. This process mirrors common operations in web applications and desktop software where rapid image display is critical.

Pixel iteration, on the other hand, was selected to capture the efficiency of low-level image manipulation. Many image processing tasks, such as filtering, transformation, and color adjustment, require access to each pixel individually. By measuring the time taken to iterate over all pixels and apply a basic grayscale conversion, a clear indicator of the library’s capability in handling computationally intensive tasks was obtained. These metrics were chosen over alternatives like image saving speed or memory usage because they directly reflect two complementary dimensions: high-level operational overhead and low-level data processing efficiency.

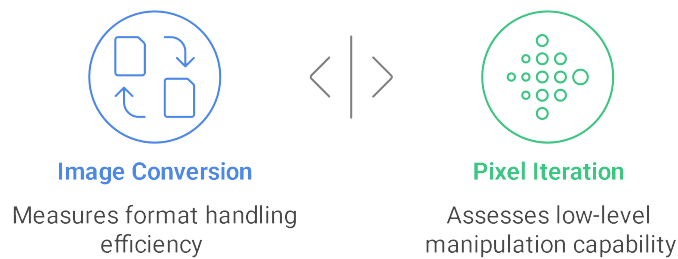


Figure 2.2: Metrics for performance comparison of image processing libraries, including image conversion and pixel iteration and what these metrics represent.

Measurement techniques evolved from initial prototypes and iterative refinements. The importance of warm-up iterations was learned from experiments, as the system needed time to

stabilize before meaningful measurements could be taken. This warm-up phase mitigated the effects of just-in-time compilation and caching, ensuring that subsequent iterations reflected steady-state performance rather than the anomalies of system initialization.

2.2.1 Defining the Image Conversion Metric

The image conversion test was designed as a JPEG image file loaded from disk, converted to PNG format, and then saved. The JPEG and PNG formats were chosen as examples of a common conversion scenario since JPEG is widely used for image storage and PNG is a lossless format suitable for web applications. The entire process is timed from start to finish. This approach involves several steps that are repeated over many iterations. Initially, 5 iterations as warm-ups are executed to allow the system to stabilize. The warm-up durations are recorded separately and then excluded from the main performance analysis. Once the system is in a steady state, a fixed number of 100 iterations is performed, and the time taken for each one is recorded.

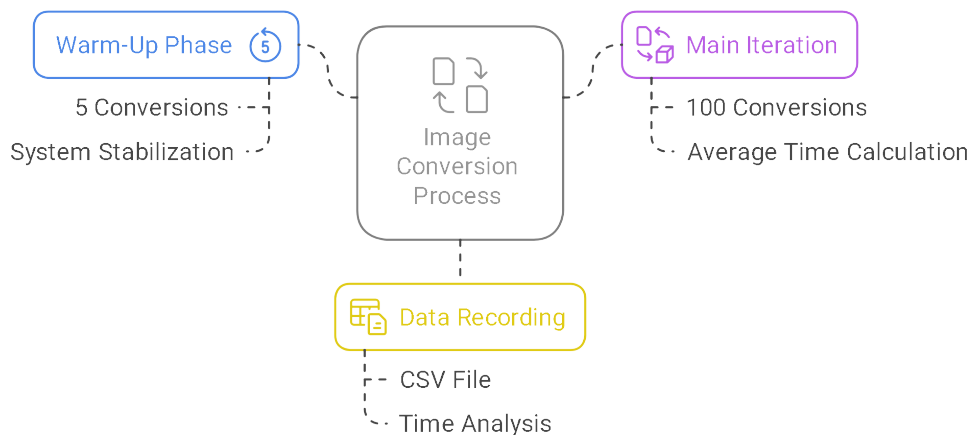


Figure 2.3: Diagram of the Image Conversion Measurement Process, including the phases of warm-up and main iterations and data collection.

The .NET Stopwatch class is used to record the elapsed time for each stage of the process. By repeating this process for a series of iterations—first running several warm-up cycles and then main iterations—a dataset was generated, that could be averaged to produce normalized performance figures.

2.2.2 Defining the Pixel Iteration Metric

The pixel iteration metric targets the efficiency of low-level pixel operations, which are foundational for many advanced image processing techniques. The focus here was on isolating the per-pixel operation, independent of any higher-level image processing abstractions. The measured time provided insights into how efficiently each library handles large amounts of pixel data, a critical factor when scaling to high-resolution images or real-time processing tasks.

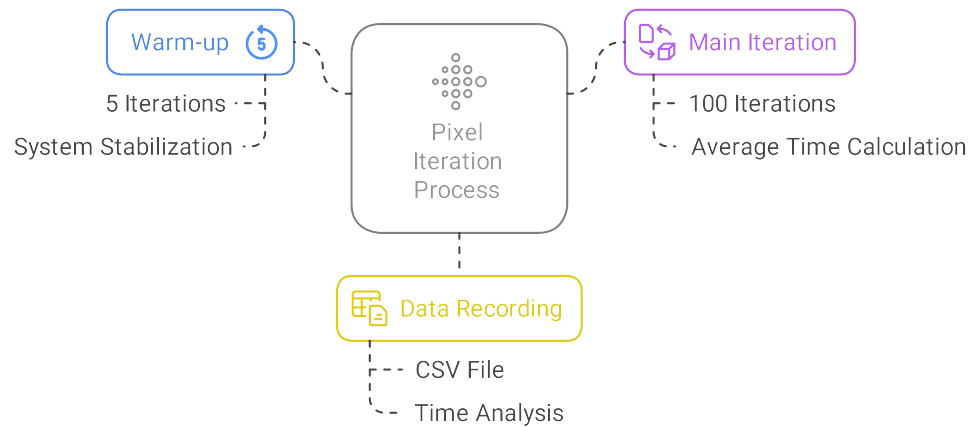


Figure 2.4: Diagram of the Pixel Iteration Measurement Process, including the phases of warm-up and main iterations and data collection.

In this test, the image is loaded into memory, and a nested loop iterates over each pixel. For each pixel, a basic grayscale conversion is applied by computing the average of the red, green, and blue channels, and then rewriting the pixel with the computed grayscale value. Similar to the image conversion test, a series of warm-up iterations is run to ensure the system has reached a stable state. After this phase, the main iterations are executed, and the time for each cycle is recorded. The key metric is the average time per iteration, which serves as an indicator of the library’s efficiency in handling per-pixel operations. The rationale behind this metric is that many advanced image processing tasks, such as filtering or feature extraction, require efficient pixel-level manipulation.

2.2.3 Criteria for Library Comparison

This comparative evaluation was based on a set of well-defined criteria that reflect both technical performance and practical implementation considerations. The primary criteria were performance (as measured by the two key metrics), functionality (including support for a wide range of image processing tasks), ease of integration (the simplicity of adopting the library within a .NET environment), and licensing. In addition to performance, the integration of BenchmarkDotNet for memory profiling adds another layer to the analysis, allowing the evaluation of trade-offs between speed and memory consumption.

Functionality was assessed by mapping each library’s capabilities against a comprehensive feature set that included image loading, pixel manipulation, format conversion, and high-level operations such as image composition. Ease of integration was evaluated by considering the availability of wrappers or bindings, the clarity of documentation, and the level of community support. Licensing was scrutinized not only in terms of costs but also in terms of the freedoms and restrictions imposed by each license (e.g., Apache 2.0 and MIT licenses versus commercial licensing models). Tables of feature comparison that was created for each library, and available in the appendix (see Chapter 6).

2.3 Experimental Setup and Environment

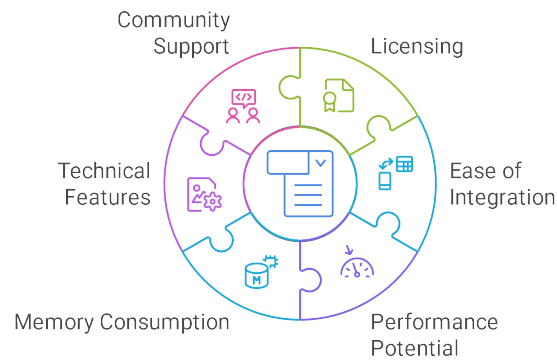


Figure 2.5: Graphical representation of the criteria used for library comparison, including performance, functionality, ease of integration, Community support, and licensing.

Finally, selection of criteria for libraries are grounded in both technical and practical considerations, ensuring that findings are relevant to a wide range of use cases—from small-scale applications to enterprise-level deployments.

2.3 Experimental Setup and Environment

The tests were conducted in a controlled environment to ensure reproducibility and accuracy. Insuring that the hardware setup and software environment were consistent across all experiments by using same machine to eliminate variability due to hardware differences. The software environment was configured with a timer, namely the `Stopwatch` class, which provided millisecond-level precision. And memory profiling was done using `BenchmarkDotNet` in separate tests to capture not only execution times but also memory allocations and garbage collection metrics, even though the primary focus remained on processing speed.

2.4 Data Collection and Processing

The collected data includes the total time taken for the warm-up phase, the average time per iteration during the main phase, and the cumulative time including warm-up. These metrics together provide a comprehensive view for both the image conversion and pixel iteration tests. The simplicity of this test allows for easy replication and clear comparisons among different libraries, which is essential when making performance-based decisions. Each iteration's timing data was recorded using the high-resolution `Stopwatch` class and stored in memory. Following the completion of each test, the raw data was exported to an Excel file using the `EPPlus` library. This allows for statistical analysis later, such as calculating the mean, median, and standard deviation of the performance times. The Excel files also served as a repository for comparative charts and graphs, which will be used to visually represent the findings.

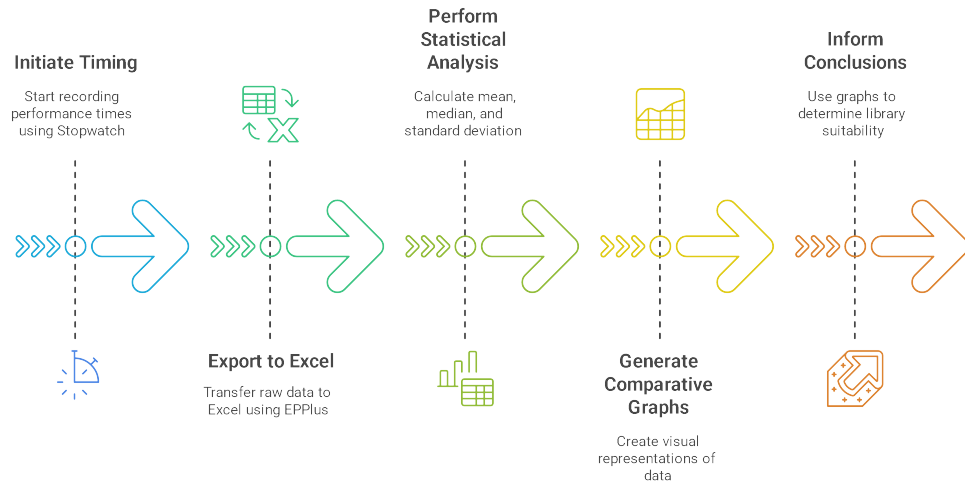


Figure 2.6: Graphical representation of the data collection and processing steps, including the use of the `Stopwatch` class, `EPlus` library, and Excel files.

Additionally for memory profiling, the `BenchmarkDotNet` library was used to measure memory consumption during the tests. `BenchmarkDotNet` provides detailed memory allocation and garbage collection reports in console output, which were captured and stored. Then after analyzing the data, the results were aggregated and visualized to provide another layer of insight into the libraries' performance characteristics. These visuals were then used to form the conclusions regarding speed, memory efficiency, and overall suitability for various tasks.

2.5 Conclusion

The methodology adopted in this study is not only a tool for performance measurement but also an exploration and discovery. Each step—from defining the metrics to processing the data and selecting the libraries—was a choice aimed at isolating the factors that matter most in image processing.

In conclusion, the methodology provides a robust framework for comparing image processing libraries. It highlights the critical trade-offs between speed, memory usage, ease of integration, and licensing costs. The insights derived from this study offer valuable guidance for developers and researchers alike, paving the way for more efficient and cost-effective image processing solutions in both academic and commercial settings.

3 Implementation

This chapter details the implementation of a comprehensive benchmarking framework to evaluate several image processing libraries, including ImageSharp, OpenCvSharp paired with SkiaSharp, Emgu CV coupled with StructureSketching, and Magick.NET integrated with MagicScaler. The objective was to create an end-to-end system that not only measures execution times for common image operations but also provides insights into memory usage.

This has been sought to answer key questions regarding the efficiency of image conversion and pixel iteration operations—two fundamental tasks in image processing. The following sections describe the review process, architectural decisions, and technical implementations in the study. The full implementation, including source code and benchmarking results, is available at Gitlab repository¹.

3.1 System Architecture and Design Rationale

The design of the benchmarking framework was guided by the need for consistency, repeatability, and scientific severity. The system was architected to support multiple libraries through a common interface, ensuring that each library's performance could be measured under identical conditions. At the core of the design was a two-phase benchmarking process: an initial warm-up phase to account for any initialization overhead, followed by a main test phase where the actual performance metrics were recorded.

In constructing the system, several important decisions were made. First, a modular approach was employed, separating the benchmarking routines into distinct components. This allowed the logic for image conversion and pixel iteration to be encapsulated into separate classes, each responsible for executing a series of timed iterations and logging the results.

Code 3.1: Design of the benchmarking framework

```
1  public class ImageConversionBenchmark{
2
3      // Benchmarking logic for image conversion
4  }
5  public class PixelIterationBenchmark{
6
7      // Benchmarking logic for pixel iteration
8  }
```

The architecture also included a dedicated component for result aggregation, which exported data into an Excel file using EPPlus, thereby facilitating further analysis and visualization.

¹https://mygit.th-deg.de/sf07627/fazeli_shahroudi-sepehr-master-sthesis

Code 3.2: Design of the benchmarking framework

```

1  using OfficeOpenXml;
2
3  public class ExcelExporter{
4
5      // Logic for exporting benchmark results to an Excel sheet in a
      structured format
6  }

```

An essential aspect of the design was the uniformity of testing. Despite the differences in methods of implementation among the libraries, the benchmarking framework was designed to abstract away these differences. Each library was integrated by implementing the same sequence of operations: reading an image from disk, processing the image (either converting its format or iterating over its pixels to apply a grayscale filter), and finally saving the processed image back to disk. This uniform methodology ensured that the performance comparisons were both fair and reproducible.

The architecture also accounted for system-level factors such as memory management and garbage collection. For instance, in languages like C#, where unmanaged resources must be explicitly disposed of, the design included rigorous cleanup routines to ensure that each iteration began with a clean slate. This attention to detail was crucial in obtaining accurate measurements, as any residual state from previous iterations could skew the results.

Code 3.3: Design of the benchmarking framework

```

1  using BenchmarkDotNet.Attributes;
2  using BenchmarkDotNet.Running;
3
4  class Program
5  {
6      static void Main(string[] args)
7      {
8          BenchmarkRunner.Run<Benchmarks>();
9      }
10 }
11
12 [MemoryDiagnoser]
13 public class Benchmarks{
14
15     [Benchmark]
16     public void ImageConversionBenchmark(){
17         // Image conversion logic
18     }
19
20     [Benchmark]
21     public void PixelIterationBenchmark(){
22         // Pixel iteration logic
23     }
24 }

```

3.2 Benchmarking Implementation

The implementation of the benchmarking framework is divided into two main tests: the image conversion benchmark and the pixel iteration benchmark. Both tests follow a similar structure, starting with a warm-up phase to mitigate initialization effects, followed by a series of iterations where performance metrics are recorded.

3.2.1 Image Conversion Benchmark Implementation

The image conversion benchmark is designed to measure the time it takes to load an image from disk, convert its format, and save the result. This process is critical in many image processing pipelines, where quick and efficient conversion between formats can significantly impact overall throughput.

The code snippet below illustrates the core routine for this benchmark. The process begins with a series of warm-up iterations, during which the system's just-in-time (JIT) compilation and caching mechanisms are activated. After the warm-up phase, the main iterations are executed, with each iteration logging the time taken for the conversion.

Code 3.4: Image conversion benchmark implementation (ImageSharp-Testing.cs)

```

1 public class ImageConversionBenchmark
2 {
3     public static (double warmupTime, double averageTime, double totalTime
4         ) RunBenchmark(string inputPath, string outputPath, int iterations
5         )
6     {
7         long totalElapsedMilliseconds = 0;
8         long warmupTime = 0;
9         int warmupIterations = 5;
10        Stopwatch stopwatch = new Stopwatch();
11
12        // Warm-up iterations to allow the system to reach steady state.
13        for (int i = 0; i < warmupIterations; i++)
14        {
15            stopwatch.Reset();
16            stopwatch.Start();
17            using (Image image = Image.Load(inputPath))
18            {
19                using (FileStream fs = new FileStream(outputPath, FileMode
20                    .Create))
21                {
22                    image.Save(fs, new PngEncoder());
23                }
24            }
25            stopwatch.Stop();
26            warmupTime += stopwatch.ElapsedMilliseconds;
27        }
28
29        // Main iterations where actual performance data is collected.
30        for (int i = 0; i < iterations; i++)
31        {

```

```

29         stopwatch.Reset();
30         stopwatch.Start();
31         using (Image image = Image.Load(inputPath))
32         {
33             using (FileStream fs = new FileStream(outputPath, FileMode
34                 .Create))
35             {
36                 image.Save(fs, new PngEncoder());
37             }
38         }
39         stopwatch.Stop();
40         totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
41         Console.WriteLine($"Iteration {i + 1}: Image conversion took {
42             stopwatch.ElapsedMilliseconds} ms");
43     }
44
45     double averageTime = totalElapsedMilliseconds / (double)iterations
46     ;
47     double totalTime = warmupTime + totalElapsedMilliseconds;
48     Console.WriteLine($"Warm-up: {warmupTime} ms, Average: {
49         averageTime} ms, Total: {totalTime} ms");
50
51     return (warmupTime, averageTime, totalTime);
52 }
53 }

```

In the implementation, the warm-up phase executes for five iterations, during which the image is loaded, converted to PNG format, and the elapsed time is accumulated. Subsequently, the main benchmark conducts 100 iterations of the identical operation, enabling the calculation of a statistically significant average execution time. This methodological approach ensures the isolation of steady-state performance from initialization overhead, providing metrics that accurately reflect the operational cost of image conversion.

This benchmark design evolved through iterative refinement. Initial experimental observations revealed significantly higher latency during the first iterations, necessitating the introduction of a dedicated warm-up phase. The benchmark methodology has been progressively optimized to ensure maximum isolation between iterations, thereby minimizing the influence of transient system states on measurement accuracy.

While this benchmarking approach is primarily analytical in nature, it's implicit in practical applications as well. For persistent server applications or batch processing systems, the steady-state performance metrics post-warm-up represent the most relevant operational characteristics. Conversely, for interactive applications, command-line utilities, or serverless computing environments, the initial performance represented by the warm-up phase may be more indicative of user-perceived responsiveness, as these contexts typically experience the full initialization cost with each invocation.

3.2.2 Pixel Iteration Benchmark Implementation

The pixel iteration benchmark measures the time taken to perform a basic image processing operation—converting an image to grayscale by iterating over each pixel. While modern image

3.2 Benchmarking Implementation

processing often employs vectorized operations on entire matrices for efficiency, pixel-by-pixel iteration remains relevant in several scenarios: when implementing custom filters with complex logic, when working with specialized pixel formats, or when memory constraints limit bulk operations. Additionally, this benchmark provides insight into the underlying performance characteristics of image libraries even if vectorized alternatives would be preferred in production environments. By examining the performance of this fundamental operation, a better understanding of the efficiency trade-offs in various image processing contexts is achieved.

For ImageSharp, the implementation involves loading the image as an array of pixels, processing each pixel to compute its grayscale value, and then updating the image accordingly. The following snippet provides a glimpse into this process:

Code 3.5: Image conversion benchmark implementation (ImageSharp-Testing.cs)

```
1 using SixLabors.ImageSharp;
2 using SixLabors.ImageSharp.Formats.Png;
3 using SixLabors.ImageSharp.PixelFormats;
4 public class PixelIterationBenchmark
5 {
6     public static (double warmupTime, double averageTime, double totalTime)
7         RunBenchmark(string imagePath, int iterations)
8     {
9         long totalElapsedMilliseconds = 0;
10        long warmupTime = 0;
11        int warmupIterations = 5;
12        Stopwatch stopwatch = new Stopwatch();
13
14        // Warm-up phase for pixel iteration
15        for (int i = 0; i < warmupIterations; i++)
16        {
17            stopwatch.Reset();
18            stopwatch.Start();
19            using (Image<Rgba32> image = Image.Load<Rgba32>(imagePath))
20            {
21                int width = image.Width;
22                int height = image.Height;
23                for (int y = 0; y < height; y++)
24                {
25                    for (int x = 0; x < width; x++)
26                    {
27                        Rgba32 pixel = image[x, y];
28                        byte gray = (byte)((pixel.R + pixel.G + pixel.B) / 3);
29                        image[x, y] = new Rgba32(gray, gray, gray, pixel.A);
30                    }
31                }
32            }
33            stopwatch.Stop();
34            warmupTime += stopwatch.ElapsedMilliseconds;
35        }
36
37        // Main iterations to measure pixel iteration performance
```

```

37     for (int i = 0; i < iterations; i++)
38     {
39         stopwatch.Reset();
40         stopwatch.Start();
41         using (Image<Rgba32> image = Image.Load<Rgba32>(imagePath))
42         {
43             int width = image.Width;
44             int height = image.Height;
45             for (int y = 0; y < height; y++)
46             {
47                 for (int x = 0; x < width; x++)
48                 {
49                     Rgba32 pixel = image[x, y];
50                     byte gray = (byte)((pixel.R + pixel.G + pixel.B) /
51                                     3);
52                     image[x, y] = new Rgba32(gray, gray, gray, pixel.A);
53                 }
54             }
55             stopwatch.Stop();
56             totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
57             Console.WriteLine($"Iteration {i + 1}: Pixel iteration took {
58                 stopwatch.ElapsedMilliseconds} ms");
59         }
60         double averageTime = totalElapsedMilliseconds / (double)iterations;
61         ;
62         double totalTime = warmupTime + totalElapsedMilliseconds;
63         Console.WriteLine($"Warm-up: {warmupTime} ms, Average: {
64             averageTime} ms, Total: {totalTime} ms");
65     }
66 }

```

The code measures the performance of a grayscale conversion operation by iterating over each pixel of an image. As in the image conversion, it uses a timer (Stopwatch) and divides the process into two phases: a warm-up phase and a measurement phase. During the warm-up phase, the image is loaded and processed five times. This phase helps stabilize performance by mitigating any startup overheads. Each iteration involves loading the image, iterating over its width and height, reading each pixel, computing the grayscale value by averaging the red, green, and blue channels, and assigning the new grayscale value back while preserving the alpha channel. The use of the `using` statement ensures that the image is properly disposed after processing.

In the measurement phase, the same processing occurs over a user-specified number of iterations (100 iterations). After running all iterations, the code calculates the average time per iteration and the total time including warm-up. This approach isolates the steady-state performance from any one-time overhead, resulting in more accurate measurements that reflect the true cost of pixel-by-pixel manipulations.

The design emphasizes clear resource management, detailed timing, and separation of initialization costs from the main measurement, which are crucial when every microsecond of processing time matters in image manipulation scenarios.

The main focus of the implementation was to capture the interplay between algorithmic efficiency and system-level resource management. Every pixel operation is executed in a closed loop, and even minor inefficiencies can accumulate over hundreds of iterations. The loop structure is designed and a stopwatch is used to measure elapsed time to matter of attention that should be paid to details during development. Because even in high-level libraries such as ImageSharp, every microsecond counts when processing large images.

3.3 Libraries Implementation

As discussed in the Methodology chapter, a comprehensive evaluation was undertaken to assess the strengths and limitations of various image processing libraries. This analysis informed the decision to implement integrations for frameworks: OpenCvSharp with SkiaSharp, and Emgu CV with Structure.Sketching, and Magick.NET with MagicScaler. The following excerpt presents representative code segments that illustrate the implementation strategies developed for these libraries. These segments not only capture the theoretical rationale behind each implementation approach but also reflect the practical constraints and performance considerations addressed throughout the thesis.

3.3.1 OpenCvSharp and SkiaSharp Implementation

The following implementation shows how the OpenCvSharp and SkiaSharp libraries are integrated to perform image conversion and pixel iteration tasks. Image conversion was implemented using OpenCvSharp, while pixel iteration was implemented using SkiaSharp.

Code 3.6: SkiaSharp Implementation (RunBenchmark Method)

```
1 using OpenCvSharp;
2 using SkiaSharp;
3
4 // Image Conversion logic using SkiaSharp
5 public class ImageConversionBenchmark
6 {
7     public static (double warmupTime, double averageTimeExcludingWarmup,
8         double totalTimeIncludingWarmup) RunBenchmark(string inputPath,
9         string outputPath, int iterations)
```

The ImageConversionBenchmark class contains a static method RunBenchmark that takes the input image path, output image path, and number of iterations as input parameters. The method returns a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up, which will be used to form the results Excel file.

Code 3.7: SkiaSharp Implementation (Initialization)

```
1 {
2     long totalElapsedMilliseconds = 0;
```

```

3 long warmupTime = 0;
4 int warmupIterations = 5;
5 Stopwatch stopwatch = new Stopwatch();

```

First, the `totalElapsedMilliseconds` and `warmupTime` variables are initialized and as discussed in the methodology chapter, the `warmupIterations` are set to 5. A stopwatch object is created to measure the elapsed time for each iteration.

Code 3.8: SkiaSharp Implementation (Warm-up Iterations)

```

1 // Warm-up iterations
2 for (int i = 0; i < warmupIterations; i++)
3 {
4     stopwatch.Reset();
5     stopwatch.Start();
6
7     using (var image = Cv2.ImRead(inputPath, ImreadModes.Color))
8     {
9         Cv2.ImWrite(outputPath, image);
10    }
11
12    stopwatch.Stop();
13    warmupTime += stopwatch.ElapsedMilliseconds;
14 }

```

The warm-up phase is executed five times to ensure that the libraries are fully initialized before the main iterations begin. In each iteration, the code reads an image using `Cv2.ImRead`, and writes the image using `Cv2.ImWrite`. The elapsed time for each iteration is recorded using the stopwatch object.

Code 3.9: SkiaSharp Implementation (Main Iterations)

```

1 // Main iterations
2 for (int i = 0; i < iterations; i++)
3 {
4     stopwatch.Reset();
5     stopwatch.Start();
6
7     using (var image = Cv2.ImRead(inputPath, ImreadModes.Color))
8     {
9         Cv2.ImWrite(outputPath, image);
10    }
11
12    stopwatch.Stop();
13    totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
14    Console.WriteLine($"Iteration {i + 1}: Image conversion took {stopwatch.
15        ElapsedMilliseconds} ms");
15 }

```

After the warm-up phase, the main iterations are executed, and the elapsed time for each iteration is recorded. The results are then aggregated and returned as a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up.

Code 3.10: SkiaSharp Implementation (Results Calculation)

```

1  double averageTimeExcludingWarmup = totalElapsedMilliseconds / (double)
    iterations;
2  double totalTimeIncludingWarmup = warmupTime + totalElapsedMilliseconds;
3
4  Console.WriteLine($"Warm-up time for image conversion: {warmupTime} ms");
5  Console.WriteLine($"Average time excluding warm-up for image conversion:
    {averageTimeExcludingWarmup} ms");
6  Console.WriteLine($"Total time including warm-up for image conversion: {
    totalTimeIncludingWarmup} ms");
7
8  return (warmupTime, averageTimeExcludingWarmup, totalTimeIncludingWarmup)
    ;
9  }
10 }
```

Finally, the average time excluding warm-up, total time including warm-up, and warm-up time are calculated. These values are then printed to the console and returned as a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up.

The pixel iteration benchmark, on the other hand, uses SkiaSharp to perform pixel-wise operations on the image.

Same as the image conversion benchmark, the pixel iteration benchmark is implemented as a static method `RunBenchmark` that takes the image path and the number of iterations as input parameters. The method returns a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up. And in the same way variables are initialized.

Code 3.11: OpenCvSharp Implementation (Warm-up Iterations)

```

1  // Warm-up iterations
2  for (int i = 0; i < warmupIterations; i++)
3  {
4      stopwatch.Reset();
5      stopwatch.Start();
6
7      using (var image = Cv2.ImRead(imagePath, ImreadModes.Color))
8      {
9          for (int y = 0; y < image.Rows; y++)
10         {
11             for (int x = 0; x < image.Cols; x++)
12             {
13                 var pixel = image.At<Vec3b>(y, x);
14                 byte gray = (byte)((pixel.Item0 + pixel.Item1 + pixel.Item2) / 3);
15                 image.Set(y, x, new Vec3b(gray, gray, gray));
16             }
17         }
18     }
19
20     stopwatch.Stop();
21     warmupTime += stopwatch.ElapsedMilliseconds;
22 }
```

The warm-up phase is executed five times to ensure that the libraries are fully initialized before the main iterations begin. In each iteration, the code reads an image using `Cv2.ImRead`, iterates over each pixel, calculates the grayscale value, and then sets the pixel value using `image.At<Vec3b>` and `image.Set`. The elapsed time for each iteration is recorded using the stopwatch object.

Code 3.12: OpenCvSharp Implementation (Main Iterations)

```

1  // Main iterations
2  for (int i = 0; i < iterations; i++)
3  {
4      stopwatch.Reset();
5      stopwatch.Start();
6
7      using (var image = Cv2.ImRead(imagePath, ImreadModes.Color))
8      {
9          for (int y = 0; y < image.Rows; y++)
10         {
11             for (int x = 0; x < image.Cols; x++)
12             {
13                 var pixel = image.At<Vec3b>(y, x);
14                 byte gray = (byte)((pixel.Item0 + pixel.Item1 + pixel.Item2) / 3);
15                 image.Set(y, x, new Vec3b(gray, gray, gray));
16             }
17         }
18     }
19
20     stopwatch.Stop();
21     totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
22     Console.WriteLine($"Iteration {i + 1}: Pixel iteration took {stopwatch.
        ElapsedMilliseconds} ms");
23 }

```

After the warm-up phase, the main iterations are executed, using the same logic as the warm-up phase. The elapsed time for each iteration is recorded, and the results are then aggregated and returned as a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up.

Code 3.13: OpenCvSharp Implementation (Results Calculation)

```

1  double averageTimeExcludingWarmup = totalElapsedMilliseconds / (double)
    iterations;
2  double totalTimeIncludingWarmup = warmupTime + totalElapsedMilliseconds;
3
4  Console.WriteLine($"Warm-up time for pixel iteration: {warmupTime} ms");
5  Console.WriteLine($"Average time excluding warm-up for pixel iteration: {
    averageTimeExcludingWarmup} ms");
6  Console.WriteLine($"Total time including warm-up for pixel iteration: {
    totalTimeIncludingWarmup} ms");
7
8  return (warmupTime, averageTimeExcludingWarmup, totalTimeIncludingWarmup)
    ;
9  }

```

```
10 }
```

Finally, the average time excluding warm-up, total time including warm-up, and warm-up time are calculated. These values are then printed to the console and returned as a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up. The returned values are then used to generate the results in an Excel file.

3.3.2 Magick.NET Implementation

In the implementation of both image conversion and pixel iteration benchmarks, Magick.NET library was used. This decision was based on Magick.NET's comprehensive functionality, which includes support for high-quality image conversion and efficient pixel-wise operations.

Similar to the previous section on OpenCvSharp and SkiaSharp, the ImageConversionBenchmark class for Magick.NET features a static RunBenchmark method. In this method, the necessary variables are initialized to measure and record the performance of image conversion operations. This consistent approach across libraries facilitates a clear comparison of their performance under similar conditions.

In logic for the warm-up phase and main iterations, change was only the library-specific functions used for image conversion and pixel iteration. Implementing image conversion using Magick.NET involved reading an image using `new MagickImage(inputPath)` to read an image and `image.Write(outputPath, MagickFormat.Png)` to write an image, and the image conversion benchmark was implemented.

Code 3.14: Magick.NET Implementation (Image Conversion)

```
1 // Warm-up iterations
2 for (int i = 0; i < warmupIterations; i++)
3 {
4     stopwatch.Reset();
5     stopwatch.Start();
6
7     using (var image = new MagickImage(inputPath))
8     {
9         image.Write(outputPath, MagickFormat.Png);
10    }
11
12    stopwatch.Stop();
13    warmupTime += stopwatch.ElapsedMilliseconds;
14 }
15
16 // Main iterations
17 for (int i = 0; i < iterations; i++)
18 {
19     stopwatch.Reset();
20     stopwatch.Start();
21
```

```

22     using (var image = new MagickImage(inputPath))
23     {
24         image.Write(outputPath, MagickFormat.Png);
25     }
26
27     stopwatch.Stop();
28     totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
29     Console.WriteLine($"Iteration {i + 1}: Image conversion took {
30         stopwatch.ElapsedMilliseconds} ms");

```

The pixel iteration benchmark was implemented by first retrieving the pixel data using the `image.GetPixels()` method. Then, for each pixel, the color channels were set to the same gray value using the `pixels.SetPixel(x, y, new ushort[] { gray, gray, gray })` function. This process was repeated for each pixel in the image for both the warm-up phase and the main iterations.

Code 3.15: Magick.NET Implementation (Pixel Iteration)

```

1  // Warm-up iterations
2  for (int i = 0; i < warmupIterations; i++)
3  {
4      stopwatch.Reset();
5      stopwatch.Start();
6
7      using (var image = new MagickImage(imagePath))
8      {
9          var pixels = image.GetPixels();
10         for (int y = 0; y < image.Height; y++)
11         {
12             for (int x = 0; x < image.Width; x++)
13             {
14                 var pixel = pixels.GetPixel(x, y); // Get pixel data
15                 ushort gray = (ushort)((pixel[0] + pixel[1] + pixel[2]) /
16                     3); // Convert to grayscale
17                 pixels.SetPixel(x, y, new ushort[] { gray, gray, gray });
18                 // Set pixel data with ushort[]
19             }
20         }
21     }
22     stopwatch.Stop();
23     warmupTime += stopwatch.ElapsedMilliseconds;
24 }
25 // Main iterations
26 for (int i = 0; i < iterations; i++)
27 {
28     stopwatch.Reset();
29     stopwatch.Start();
30
31     using (var image = new MagickImage(imagePath))
32     {

```

3.3 Libraries Implementation

```
33     var pixels = image.GetPixels();
34     for (int y = 0; y < image.Height; y++)
35     {
36         for (int x = 0; x < image.Width; x++)
37         {
38             var pixel = pixels.GetPixel(x, y); // Get pixel data
39             ushort gray = (ushort)((pixel[0] + pixel[1] + pixel[2]) /
40                                     3); // Convert to grayscale
41             pixels.SetPixel(x, y, new ushort[] { gray, gray, gray });
42             // Set pixel data with ushort[]
43         }
44     }
45     stopwatch.Stop();
46     totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
47     Console.WriteLine($"Iteration {i + 1}: Pixel iteration took {stopwatch
48         .ElapsedMilliseconds} ms");
49 }
```

The results of the image conversion and pixel iteration benchmarks were then like the previous libraries, aggregated and returned as a tuple containing the warm-up time, average time excluding warm-up, and total time including warm-up. These values were then used to generate the results in an Excel file.

3.3.3 Emgu CV and Structure.Sketching Implementation

The implementation of Emgu CV and Structure.Sketching libraries in the benchmarking framework are shown in the following code snippet. The code demonstrates how the Emgu CV library is used for image conversion, while Structure.Sketching is used for pixel iteration.

For image conversion, the code reads an image using `CvInvoke.Imread` and writes the image using `CvInvoke.Imwrite`. The warm-up phase and main iterations are executed in a similar manner to the previous libraries, with the elapsed time for each iteration recorded using a stopwatch object.

Code 3.16: Emgu CV Implementation (Image Conversion)

```
1 using Emgu.CV;
2 using Emgu.CV.CvEnum;
3 using Emgu.CV.Structure;
4 using Structure.Sketching;
5 using Structure.Sketching.Formats;
6 using Structure.Sketching.Colors;
7
8 // Warm-up iterations
9 for (int i = 0; i < warmupIterations; i++)
10 {
11     stopwatch.Reset();
12     stopwatch.Start();
13
14     using (Mat image = CvInvoke.Imread(inputPath, ImreadModes.Color))
```

```

15     {
16         CvInvoke.Imwrite(outputPath, image);
17     }
18
19     stopwatch.Stop();
20     warmupTime += stopwatch.ElapsedMilliseconds;
21 }
22
23 // Main iterations
24 for (int i = 0; i < iterations; i++)
25 {
26     stopwatch.Reset();
27     stopwatch.Start();
28
29     using (Mat image = CvInvoke.Imread(inputPath, ImreadModes.Color))
30     {
31         CvInvoke.Imwrite(outputPath, image);
32     }
33
34     stopwatch.Stop();
35     totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
36     Console.WriteLine($"Iteration {i + 1}: Image conversion took {
37         stopwatch.ElapsedMilliseconds} ms");
37 }

```

For pixel iteration, it uses the `Structure.Sketching` and the code reads an image using `new Structure.Sketching.Image(imagePath)` and iterates over each pixel, calculating the grayscale value and setting the pixel value using `image.Pixels[(y * width) + x]`. The warm-up phase and main iterations are executed in a similar manner to the previous libraries, with the elapsed time for each iteration recorded using a stopwatch object.

Code 3.17: Structure.Sketching Implementation (Pixel Iteration)

```

1 // Warm-up iterations
2 for (int i = 0; i < warmupIterations; i++)
3 {
4     stopwatch.Reset();
5     stopwatch.Start();
6
7     var image = new Structure.Sketching.Image(imagePath);
8     int width = image.Width;
9     int height = image.Height;
10
11     for (int y = 0; y < height; y++)
12     {
13         for (int x = 0; x < width; x++)
14         {
15             var pixel = image.Pixels[(y * width) + x];
16             byte gray = (byte)((pixel.Red + pixel.Green + pixel.Blue) / 3);
17             ;
18             image.Pixels[(y * width) + x] = new Color(gray, gray, gray,
19                 pixel.Alpha);
18         }
18     }

```

```

19     }
20
21     stopwatch.Stop();
22     warmupTime += stopwatch.ElapsedMilliseconds;
23 }
24
25 // Main iterations
26 for (int i = 0; i < iterations; i++)
27 {
28     stopwatch.Reset();
29     stopwatch.Start();
30
31     var image = new Structure.Sketching.Image(imagePath);
32     int width = image.Width;
33     int height = image.Height;
34
35     for (int y = 0; y < height; y++)
36     {
37         for (int x = 0; x < width; x++)
38         {
39             var pixel = image.Pixels[(y * width) + x];
40             byte gray = (byte)((pixel.Red + pixel.Green + pixel.Blue) / 3);
41             ;
42             image.Pixels[(y * width) + x] = new Color(gray, gray, gray,
43                 pixel.Alpha);
44         }
45     }
46
47     stopwatch.Stop();
48     totalElapsedMilliseconds += stopwatch.ElapsedMilliseconds;
49     Console.WriteLine($"Iteration {i + 1}: Pixel iteration took {stopwatch
50         .ElapsedMilliseconds} ms");
51 }

```

Grayscale conversion is performed on each pixel by computing the average of the red, green, and blue components using the formula $(\text{byte})(\text{pixel.Red} + \text{pixel.Green} + \text{pixel.Blue}) / 3$. The grayscale value is then assigned to each color channel to create a grayscale image. The benchmarking process collects the results from both image conversion and pixel iteration. These results are aggregated into a tuple containing the warm-up time, the average time (excluding the warm-up phase), and the total time (including the warm-up phase). Finally, this data is used to generate an Excel file that summarizes the performance metrics.

3.4 Memory Profiling and Performance Analysis

In any high-performance image processing application, it is not enough to measure raw execution time; memory consumption is equally critical. This section describes the integration of memory profiling into the benchmarking framework to provide a comprehensive view of the performance characteristics of each library and complement the time-based measurements. Using BenchmarkDotNet—a powerful tool for .NET performance analysis—detailed metrics on memory allocation and garbage collection behavior were captured. This implementation

allowed the trade-offs between processing speed and resource utilization to be better understood.

The memory profiling is designed to evaluate not only the mean execution times but also the memory allocated during both image conversion and pixel iteration tasks. Using BenchmarkDotNet's [MemoryDiagnoser], [Orderer], and [RankColumn] attributes, data on memory consumption, garbage collection events, and total allocated memory were collected for each benchmarked operation. The BenchmarkDotNet analyzer for each method by default is configured to automatically determine how many warmup and measurement iterations to run based on the workload, environment, and statistical requirements for accurate measurements. So there is no need to implement a fixed iteration count for each method manually.

The following framework demonstrates the implementation of memory profiling and an example of how the memory diagnostics were implemented for the image conversion and pixel iteration using ImageSharp:

Code 3.18: Memory Profiling and Performance Analysis (ImageSharp)

```

1 using BenchmarkDotNet.Attributes;
2 using BenchmarkDotNet.Order;
3 using BenchmarkDotNet.Running;
4 using SixLabors.ImageSharp;
5 using SixLabors.ImageSharp.Formats.Png;
6 using SixLabors.ImageSharp.PixelFormats;
7
8 [MemoryDiagnoser]
9 [Orderer(SummaryOrderPolicy.FastestToSlowest)]
10 [RankColumn]
11 public class Benchmarks
12 {
13     private const string InputImagePath = "../../../x11.jpg";
14     private const string OutputImagePath = "../../../o.png";
15
16     [Benchmark]
17     public void ImageConversionBenchmark()
18     {
19         using (Image image = Image.Load(InputImagePath))
20         {
21             using (FileStream fs = new FileStream(OutputImagePath,
22                 FileMode.Create))
23             {
24                 image.Save(fs, new PngEncoder());
25                 Console.WriteLine("ImageConversionBenchmark completed");
26             }
27         }
28     }
29 }

```

Same logic is used for image conversion, but there were no need for iterations and warm-up phase to be implemented manually. For configuring the MemoryDiagnoser results, Orderer(SummaryOrderPolicy.FastestToSlowest) and RankColumn attributes were used to order the results based on the fastest to slowest execution times and to rank the results in the summary table, respectively to provide a better and clearer view of the

results.

Code 3.19: Memory Profiling and Performance Analysis (ImageSharp)

```
1 [Benchmark]
2 public void PixelIterationBenchmark()
3 {
4     using (Image<Rgba32> image = Image.Load<Rgba32>(InputImagePath))
5     {
6         int width = image.Width;
7         int height = image.Height;
8
9         for (int y = 0; y < height; y++)
10        {
11            for (int x = 0; x < width; x++)
12            {
13                Rgba32 pixel = image[x, y];
14                byte gray = (byte)((pixel.R + pixel.G + pixel.B) / 3);
15                image[x, y] = new Rgba32(gray, gray, gray, pixel.A);
16            }
17        }
18        Console.WriteLine("PixelIterationBenchmark completed");
19    }
20 }
```

The pixel iteration benchmark was implemented in a similar manner with the same memory diagnostics attributes. The code snippet above demonstrates the pixel iteration benchmark for ImageSharp, where each pixel in the image is converted to grayscale. The memory diagnostics provided by BenchmarkDotNet enabled tracking of the memory consumption and garbage collection events during the pixel iteration operation, providing valuable insights into the resource utilization of each library.

This code exemplifies the approach to memory diagnostics. By annotating the benchmark class with `[MemoryDiagnoser]`, BenchmarkDotNet automatically collects data on memory usage—including the number of garbage collection (GC) events and the total allocated memory during each benchmarked operation. Similar implementations were done for other libraries as well.

This level of granularity provided insights that went beyond raw timing metrics, revealing, for example, that while Emgu CV might be faster in certain operations, its higher memory consumption could be a concern for applications running on memory-constrained systems.

3.5 Result Export and Data Aggregation

Once the performance and memory metrics were collected, the next challenge was to present the results in a coherent and accessible manner. For this purpose, Excel was chosen as the output format due to its widespread adoption and ease of use for further analysis. `OfficeOpenXml` namespace, which is part of the EPPlus library, allows for the creation and manipulation of Excel files in .NET applications. The `ExcelExporter` class was implemented to aggregate the benchmark results and export them to an Excel file.

The code snippet below illustrates how the benchmark results are aggregated and exported to an Excel file:

Code 3.20: Result Export and Data Aggregation

```

1 using OfficeOpenXml;
2
3 public class ExcelExporter
4 {
5     public static void ExportResults(string excelOutputPath,
6         (double warmupTime, double averageTime, double totalTime)
7         imageConversionResults,
8         (double warmupTime, double averageTime, double totalTime)
9         pixelIterationResults)
10    {
11        using (var package = new ExcelPackage())
12        {
13            var worksheet = package.Workbook.Worksheets.Add("Benchmark
14                Results");
15            worksheet.Cells[1, 1].Value = "Benchmark";
16            worksheet.Cells[1, 2].Value = "Warm-Up Time (ms)";
17            worksheet.Cells[1, 3].Value = "Average Time (ms)";
18            worksheet.Cells[1, 4].Value = "Total Time (ms)";
19
20            worksheet.Cells[2, 1].Value = "Image Conversion";
21            worksheet.Cells[2, 2].Value = imageConversionResults.
22                warmupTime;
23            worksheet.Cells[2, 3].Value = imageConversionResults.
24                averageTime;
25            worksheet.Cells[2, 4].Value = imageConversionResults.totalTime
26                ;
27
28            worksheet.Cells[3, 1].Value = "Pixel Iteration";
29            worksheet.Cells[3, 2].Value = pixelIterationResults.warmupTime
30                ;
31            worksheet.Cells[3, 3].Value = pixelIterationResults.
32                averageTime;
33            worksheet.Cells[3, 4].Value = pixelIterationResults.totalTime;
34
35            package.SaveAs(new FileInfo(excelOutputPath));
36        }
37    }
38 }

```

The `ExcelExporter` class creates a structured Excel file with separate sheets for each benchmark operation. The results are organized into columns for the warm-up time, average time, and total time for each operation. The resulting Excel file provides a clear and concise summary of the benchmark results, making it easy to compare the performance and memory characteristics of each library.

By automating the process of result aggregation, the framework not only saves time but also minimizes the risk of manual errors. Each cell in the generated Excel file is carefully populated with benchmark data, and the resulting spreadsheet can be easily imported into analytical

3.5 Result Export and Data Aggregation

tools for further exploration. This process of exporting results serves as a bridge between the raw performance data and the actionable insights that drive decision-making in software optimization.

4 Results

This chapter presents findings from the benchmarking experiments conducted to evaluate the performance of alternative image processing libraries. The results include quantitative data on image conversion and pixel iteration times, as well as memory consumption for each library or combination tested. The data generated will be used to answer the research question and support the hypotheses formulated in the previous chapters. The benchmarking approach consisted of running two primary tests on each library: an image conversion test that measured the time taken to load, process, and save images, and a pixel iteration test that recorded the time required to process every pixel in an image for a grayscale conversion. These experiments were performed in a controlled environment, with warm-up iterations included to reduce the impact of initial overhead. Memory consumption was tracked alongside processing times using BenchmarkDotNet, thereby offering a complete picture of both speed and resource utilization.

Before discussing the results in detail, it is important to review the benchmarking design. In this study, each library was tested under the same conditions: the same input image was used, a fixed number of warm-up iterations were performed to reduce the effects of just-in-time compilation and caching, and finally, 100 main iterations were executed to ensure reliable statistics. For the image conversion test, the time measured was the duration needed to load a JPEG image, convert it to PNG, and save it back to disk. In the pixel iteration test, the focus was on recording the time required to access and change each pixel for producing a grayscale version of the image.

Memory diagnostics were captured concurrently, with particular attention to allocated memory and garbage collection events. This dual approach ensured that the results were not solely focused on speed but also took into account the resource efficiency of each solution.

4.1 Image Conversion Benchmark Results

The image conversion benchmark was performed using ImageSharp and Magick.NET as well as SkiaSharp and Structure.Sketching which were the chosen libraries in their combinations with OpenCvSharp and Emgu CV, respectively, for the conversion task. Using the same 4k resolution image, the benchmark measured the time taken to convert the image from JPEG to PNG format. Comparing the results of these libraries provides insights into their performance and efficiency in application scenarios where rapid image conversion is required—such as real-time image processing pipelines or high-volume batch processing environments. The data thus answer one of the central questions regarding which library can provide significantly faster image conversion, thereby supporting the hypothesis discussed in earlier chapters.

ImageSharp recorded an average conversion time of approximately 2,754 milliseconds. In contrast, the combination of OpenCvSharp with SkiaSharp delivered an average conversion

time of only 539 milliseconds. Similarly, Emgu CV integrated with Structure.Sketching achieved an average time of 490 milliseconds, while Magick.NET registered an average conversion time of 4,333 milliseconds.

Library	Warm-Up Time (ms)	Avg. Time Excl. Warm-Up (ms)	Total Time Incl. Warm-Up (ms)
ImageSharp	2754	480.86	50840
OpenCvSharp + SkiaSharp	539	100.31	10570
Magick.NET	4333	845.46	88879
Emgu CV + Structure.Sketching	490	59.43	6433

Table 4.1: The Image Conversion Benchmark Results in milliseconds, showing the warm-up time, average time excluding warm-up, and total time including warm-up for each library or combination.

The table 4.1, is the final dataset that been constructed by merging multiple Excel files produced by the framework described in the Implementation chapter. These results shows lightweight libraries such as SkiaSharp and Structure.Sketching outperforming ImageSharp and Magick.NET in terms of image conversion time. The data also reveals that Emgu CV with Structure.Sketching is the most efficient combination for image conversion, with the lowest average time of 490 milliseconds. On the other hand, ImageSharp and Magick.NET are significantly slower, with average times of 2,754 and 4,333 milliseconds, respectively.

To visually summarize these findings, Figure 4.1 presents a bar chart that depicts the conversion times across the evaluated libraries. The graph clearly demonstrates that the conversion times for the OpenCvSharp+SkiaSharp and Emgu CV+Structure.Sketching combinations are positioned at the lower end of the performance spectrum, while ImageSharp exhibits considerably higher times. A logarithmic scale has been employed to effectively represent the significant differences in total times—comprising both the warm-up periods and the average conversion times. This three-color graphical representation enables a thorough comparison of library performance in various contexts, such as real-time image processing and batch conversion tasks, thereby reinforcing the quantitative analysis presented earlier.

4.2 Pixel Iteration Benchmark Results

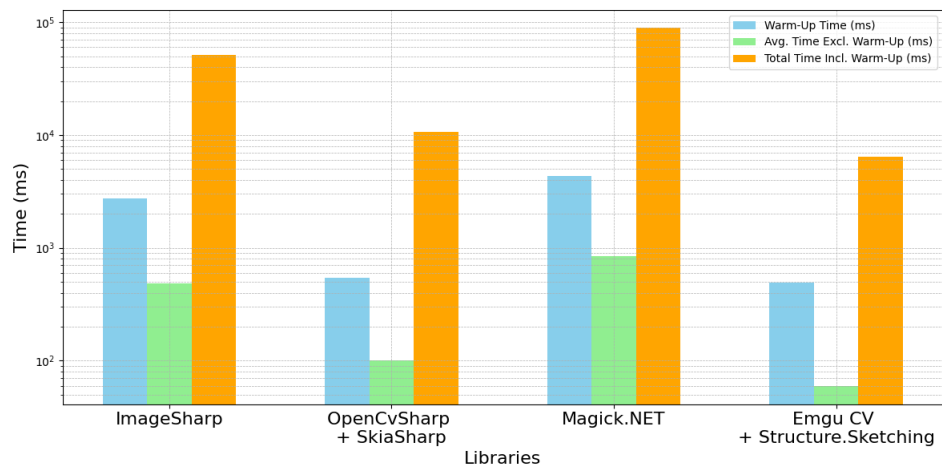


Figure 4.1: Bar chart showing the Image Conversion Benchmark Results in milliseconds, with a logarithmic scale to highlight the differences in total times. X-axis represents the libraries or combinations, while Y-axis shows the time in milliseconds.

4.2 Pixel Iteration Benchmark Results

On the other hand, the pixel iteration benchmark aimed to assess the libraries' abilities to process each pixel of an image. For ImageSharp, the warm-up phase for pixel iteration took an average of 755 milliseconds, with the main iteration averaging 117.06 milliseconds per cycle and a cumulative total of 12,461 milliseconds over 100 iterations.

Library	Warm-Up Time (ms)	Avg. Time Excl. Warm-Up (ms)	Total Time Incl. Warm-Up (ms)
ImageSharp	755	117.06	12461
OpenCvSharp + SkiaSharp	813	159.44	16757
Magick.NET	12149	2054.18	217567
Emgu CV + Structure.Sketching	1118	118.87	13005

Table 4.2: Pixel Iteration Benchmark Results in milliseconds, showing the warm-up time, average time excluding warm-up, and total time including warm-up for each library or combination.

The performance landscape changed upon examining the results for Magick.NET. This con-

figuration recorded a warm-up time of approximately 12,149 milliseconds, and the main iterations averaged 2,054.18 milliseconds, resulting in an astronomical total of 217,567 milliseconds. As discussed earlier, OpenCvSharp and Emgu CV were chosen in combinations with SkiaSharp and Structure.Sketching, respectively, for the pixel iteration task. The results of these tests provide insights into the performance of these libraries in scenarios where pixel-level operations are required, such as image processing algorithms or computer vision applications. The performance landscape also shifted upon examining the results for OpenCvSharp. This configuration recorded a warm-up time of approximately 813 milliseconds, and the main iterations averaged 159.44 milliseconds, resulting in a total of 16,757 milliseconds. In contrast, Emgu CV delivered impressive results with a warm-up time of 1,118 milliseconds and an average main iteration time of 118.87 milliseconds, culminating in a total of 13,005 milliseconds.

The table 4.2 summarizes the pixel iteration benchmark results, highlighting the warm-up and average times for each library combination. The data clearly show that Emgu CV is the most efficient library for pixel iteration, with the lowest average time of 118.87 milliseconds. ImageSharp and OpenCvSharp follow closely behind, with average times of 117.06 and 159.44 milliseconds, respectively. In contrast, Magick.NET is significantly slower, with an average time of 2,054.18 milliseconds. Graphical 4.2 depictions further highlight these performance differences.

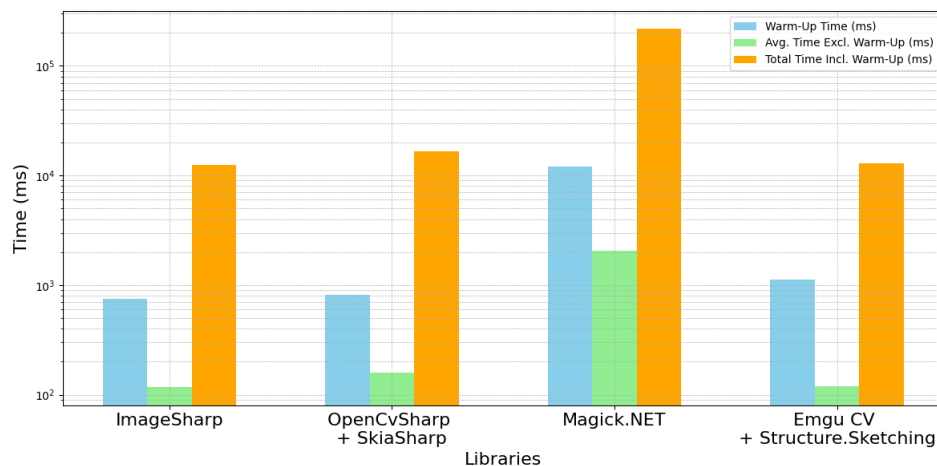


Figure 4.2: Bar chart showing the Pixel Iteration Benchmark Results in milliseconds, with a logarithmic scale to highlight the differences in total times. X-axis represents the libraries or combinations, while Y-axis shows the time in milliseconds.

The disparity between these figures 4.2 is telling. While Magick.NET excels in some aspects of image conversion, it appears less suited for tasks involving pixel-by-pixel iteration, given the significantly higher processing times. On the other hand, Emgu CV and ImageSharp produce comparable main iteration times; however, when considering the overall picture, the lower cumulative times of Emgu CV make it a more appealing choice for pixel-level operations. The visual comparisons elucidate that while ImageSharp and Emgu CV+Structure.Sketching are closely matched in main iteration performance, the excessive warm-up and overall times associated with Magick.NET underscore its limitations for this specific task.

4.3 Memory Benchmarking Results

In parallel with the time benchmarks, memory consumption was a critical parameter in the evaluation. For the image conversion tasks, SkiaSharp, as part of the OpenCvSharp+SkiaSharp configuration, exhibited the lowest memory allocation, with values approximating 58 KB. ImageSharp, in comparison, required about 5.67 MB, which is substantially higher. In the context of pixel iteration, the memory profiles were similarly divergent. ImageSharp was extremely efficient in this regard, consuming roughly 20 KB on average, whereas Emgu CV + StructureSketching, that performed exceptionally well in terms of speed for pixel iteration, in memory terms, was less efficient. It consumed around 170 MB of memory, which is significantly higher than the other libraries tested. SkiaSharp,

Library	Allocated Memory	Gen0/Gen1/Gen2 Collections
EmguCV	0.00068 MB (712 bytes)	- / - / -
ImageSharp	5.67 MB (5,805.41 KB)	1,000 / 1,000 / 1,000
SkiaSharp	0.05612 MB (58,864 bytes)	- / - / -

Table 4.3: Memory benchmarking results for the image conversion task, detailing the allocated memory (in MB) along with the associated Gen0, Gen1, and Gen2 garbage collection counts.

The table 4.3 summarizes the memory benchmarking results for image conversion. It is evident that ImageSharp has the highest memory allocation, with approximately 5.67 MB, while SkiaSharp has the lowest. Emgu CV falls in between, with a memory allocation of 0.00068 MB. These figures provide a clear indication of the memory efficiency of each library for image conversion tasks. Garbage collection counts are also included to provide additional context on the memory management behavior of each library. Gen0, Gen1, and Gen2 collections means the number of times each generation was collected during the benchmarking process. These metrics are essential for understanding how each library manages memory and how it impacts performance.

The large memory footprint of Emgu CV during pixel iteration is a noteworthy trade-off. While its performance in terms of speed is excellent, the high memory consumption must be considered when deploying the solution in memory-constrained environments. The benchmarking data collected here is critical because it provides a balanced view—speed alone does not define an optimal library, but rather the ratio of processing time to memory usage does. For a clear summary of these findings, table 4.4 provides a concise overview of the memory metrics for each library configuration.

Library	Allocated Memory	Gen0/Gen1/Gen2 Collections
EmguCV	170.00 MB (177,976,185 bytes)	33,142 / 1,571 / 1,571
ImageSharp	0.01932 MB (20.26 KB)	- / - / -
SkiaSharp	384.00 MB (403,300,552 bytes)	85 / - / -

Table 4.4: Memory Benchmarking Results for Pixel Iteration Task, detailing the allocated memory (in MB) along with the associated Gen0, Gen1, and Gen2 garbage collection counts.

The table 4.4 indicates that while SkiaSharp has the highest memory allocation for pixel iteration of approximately 384 MB, ImageSharp is the most memory-efficient, with a memory allocation of 0.01932 MB. Emgu CV falls in between, with a memory allocation of 170 MB. These figures provide a clear indication of the memory efficiency of each library for pixel iteration tasks. Garbage collection counts are also included to provide additional context on the memory management behavior of each library. Gen0, Gen1, and Gen2 collections means the number of times each generation was collected during the benchmarking process. This means that the garbage collector had to run 33,142 times for Gen0, 1,571 times for Gen1, and 1,571 times for Gen2.

4.4 Analysis and Interpretation of Results

As the final benchmarking results were collected and plotted, the emerging trends provided critical insights into the efficiency of various image processing libraries. The raw numerical data from the benchmarking suite provided an answer to the research question, but a deeper interpretation of these results allowed refinement of the understanding of the trade-offs and strengths of each alternative. This section explores the relationship between speed and memory usage, compares the empirical findings with theoretical expectations, and discusses the implications for real-world applications.

4.4.1 Comparison of Performance Trends

The performance hierarchy observed in the benchmarking results closely aligns with expectations based on each library's internal architecture. Libraries such as OpenCvSharp and Emgu CV, both built upon OpenCV's optimized C++ backend, showcased superior execution times for pixel iteration tasks. This efficiency is largely attributed to OpenCV's reliance on low-level SIMD (Single Instruction, Multiple Data) optimizations and hardware-accelerated processing paths.

Conversely, ImageSharp—despite its clean API and pure C# implementation—demonstrated significantly higher processing times, reinforcing the general principle that managed code introduces overhead compared to native libraries. In memory-constrained environments, the

4.4 Analysis and Interpretation of Results

trade-off between speed and memory usage should be carefully considered as ImageSharp's memory efficiency may outweigh its slower execution times. ImageSharp remains a viable option for applications prioritizing ease of use and portability over raw performance or in scenarios where memory conservation is critical.

Magick.NET, though powerful and highly flexible in terms of format support, performed noticeably worse in pixel iteration tasks. This result was somewhat anticipated due to the internal structure of ImageMagick, which prioritizes format conversions and high-quality rendering over raw pixel access speed. The excessive processing times observed in the Magick.NET pixel iteration benchmark further support the hypothesis that it is not optimized for this type of operation. However, its range of features and extensive format support make it a compelling choice for applications requiring advanced image processing capabilities.

The trends in memory consumption were particularly revealing. In the image conversion test, SkiaSharp exhibited the lowest memory usage, also demonstrating competitive processing times. This result is consistent with SkiaSharp's reputation for being lightweight and efficient, making it an excellent choice for applications need high performance and low memory overhead. In the pixel iteration test, Emgu CV memory usage was significantly higher than ImageSharp, highlighting the trade-off between speed and memory efficiency. This finding underscores the importance of selecting the right library based on the specific requirements of the application. This observation is consistent with Emgu CV's underlying OpenCV core, which relies on large temporary buffers and matrix structures for intermediate computations. In contrast, ImageSharp demonstrated exceptional memory efficiency during pixel iteration but was significantly slower, suggesting that its architecture prioritizes memory conservation over execution speed.

4.4.2 Trade-Offs Between Speed and Memory Usage

The relationship between speed and memory consumption is a recurring theme in performance optimization. Results underscore that achieving optimal speed often comes at the cost of increased memory usage. Emgu CV+Structure.Sketching exemplifies this trade-off: while its pixel iteration speed was among the best recorded, it consumed significantly more RAM than ImageSharp.

The implications of these trade-offs depend heavily on the intended application. For environments where processing speed is paramount—such as real-time video processing or AI-powered image enhancement—Emgu CV's increased memory footprint may be an acceptable compromise. However, in resource-constrained applications (e.g., embedded systems, mobile devices, or cloud-based deployments with strict memory limits), a lower-memory alternative like ImageSharp may be more suitable despite its lower speed.

Library	Task	Speed	Memory Usage
ImageSharp	Image Conversion	Slow	Low
	Pixel Iteration	Fast	Low

Library	Task	Speed	Memory Usage
Emgu CV	Pixel Iteration	Fast	High
SkiaSharp	Image Conversion	Fast	Low

Table 4.5: Table of Speed and Memory Trade-Offs for Image Processing Libraries, the fast/slow and high/low are relative to the other libraries.

One particularly interesting finding was that OpenCvSharp+SkiaSharp consistently delivered both high speed and low memory usage for image conversion. This anomaly suggests that this combination strikes an optimal balance, leveraging OpenCV's native optimizations while maintaining a lightweight footprint in memory. The fact that this hybrid approach outperformed even standalone OpenCV libraries further supports the notion that combining high-performance native libraries with efficient rendering engines can yield superior results.

4.5 Summary

The benchmarking results provide a comprehensive overview of the performance and efficiency of the image processing libraries tested. The data clearly show that Emgu CV + StructureSketching is the most efficient combination for image conversion, with the lowest average time of 490 milliseconds. In contrast, ImageSharp and Magick.NET are significantly slower, with average times of 2,754 and 4,333 milliseconds, respectively. For pixel iteration, Emgu CV+StructureSketching is again the most efficient, with the lowest average time of 118.87 milliseconds. ImageSharp and OpenCvSharp+SkiaSharp follow closely behind, with average times of 117.06 and 159.44 milliseconds, respectively. In contrast, Magick.NET is significantly slower, with an average time of 2,054.18 milliseconds. The memory benchmarking results further highlight the efficiency of ImageSharp and SkiaSharp in terms of memory consumption, with Emgu CV exhibiting higher memory usage. Developers can use these findings to select the most suitable library for their particular needs based on their specific requirements and constraints regarding speed and resource utilization.

5 Discussion

This chapter interprets the results obtained in the benchmarking experiments, placing them in a broader theoretical and practical context. It examines the implications of the performance metrics in terms of computational efficiency, implementation complexity, licensing considerations, and the overall usability of the evaluated image processing libraries. Moreover, this discussion extends to address the broader impact these results have on advancements in software engineering and the evolving field of image processing.

5.1 Interpreting the Results: Performance vs. Practicality

The results obtained from the benchmarking study reveal a clear hierarchy of performance among the tested libraries. However, performance alone does not determine the best library for a given use case. The ideal choice depends on a variety of factors, including memory efficiency, ease of integration, licensing constraints, and the specific needs of the application.

5.1.1 Performance Trade-offs and Suitability for Real-World Applications

From performance standpoint, OpenCvSharp + SkiaSharp and Emgu CV + StructureSketching outperform ImageSharp in both image conversion and pixel iteration tasks. However, ImageSharp showed better memory efficiency during pixel iteration, making it a viable option for applications with limited memory resources. SkiaSharp, with its lightweight architecture and cross-platform compatibility, demonstrated remarkable performance in image conversion tasks. It consistently outperformed ImageSharp while consuming significantly less memory. This makes SkiaSharp an ideal choice for applications requiring efficient format conversion without extensive manipulation of individual pixels. Emgu CV, despite its high memory usage, proved to be the fastest option for pixel iteration. This is unsurprising, given its reliance on OpenCV's highly optimized C++ backend. However, its higher memory footprint may be a drawback for applications running on constrained systems. Magick.NET, on the other hand, didn't perform well in both image conversion and pixel iteration tasks. This suggests that while Magick.NET is a robust tool for high-quality image manipulation and format conversion, it may not be suitable for performance-critical applications requiring low-latency processing. In graph 4.1 and 4.2 the performance comparison of the libraries in image conversion and pixel iteration tasks respectively can be seen.

5.1.2 The Impact of Licensing on Library Selection

Licensing can be a key consideration in selecting an image processing library. The cost of proprietary solutions can be prohibitive, particularly for small businesses or open-source projects.

ImageSharp, while powerful, requires a yearly cost of a couple of thousand dollars for commercial use. This cost must be weighed against its performance limitations. Open-source alternatives like OpenCvSharp and SkiaSharp, which are licensed under MIT and Apache 2.0 respectively, offer a compelling alternative by providing high performance at no cost. Emgu CV, although based on the open-source OpenCV framework, requires a one-time fee (version specific) of less than thousand dollars, with additional costs for future upgrades. While this is significantly more affordable than ImageSharp, it still represents an investment that must be justified by superior performance. On the other hand, Magick.NET was licensed under Apache 2.0, and provides extensive functionality for free, making it an attractive option for projects that require advanced image processing features but cannot afford proprietary licenses.

Library Combination	Licensing Model	Cost	Usage Restrictions / Remarks
ImageSharp	Proprietary (Commercial)	\$5,000/year	Requires a subscription; higher conversion times
OpenCvSharp + SkiaSharp	Open-source (Apache-2.0 & MIT)	Free	No recurring fees; excellent conversion performance
Magick.NET	Open-source (Apache-2.0)	Free	Good for advanced processing; slower pixel iteration
Emgu CV + Structure.Sketching	Open-source with paid tier	\$799 (Emgu CV only)	Cost-effective; strong for pixel manipulation and processing

Table 5.1: Library Licensing, Costs, and Usage Restrictions Comparison Table

5.2 Strengths and Weaknesses of the Different Libraries

ImageSharp's biggest advantage is its simple API and pure .NET implementation. It is easy to integrate and requires minimal setup. However, benchmarks show that it lags behind other libraries in performance. Its relatively high memory efficiency during pixel iteration is a plus, but for tasks requiring fast image conversion or pixel-level modifications, other options are preferable. The combination of OpenCvSharp and SkiaSharp offers a mix of high performance and moderate complexity. This combination provides the best balance between speed and memory efficiency. OpenCvSharp offers the power of OpenCV's optimized image processing, while SkiaSharp enhances its rendering and format conversion capabilities. However, using

5.3 Considerations for Future Research

these libraries effectively requires familiarity with both OpenCV and SkiaSharp APIs, making them less beginner-friendly than ImageSharp. Emgu CV's performance in pixel iteration tasks is unmatched, making it ideal for applications involving real-time image analysis, such as AI-driven image recognition. However, its high memory consumption may pose a problem for resource-limited environments. Structure.Sketching complements Emgu CV by providing efficient image creation and drawing capabilities, making this combination well-suited for applications requiring both processing speed and graphical rendering. In contrast, Magick.NET excels in high-quality image manipulation and resampling but falls short in raw speed. The high processing times recorded for pixel iteration indicate that Magick.NET is best suited for batch processing or scenarios where quality takes precedence over execution time. And MagicScaler provides advanced image scaling capabilities, making it a valuable tool for applications requiring precise image resizing and enhancement.

Overall There is no single library that is best for all use cases. The optimal choice depends on the application's specific requirements. If ease of implementation and maintainability are priorities, ImageSharp remains a solid choice despite its performance drawbacks. For performance-intensive applications where raw speed is essential, OpenCvSharp+SkiaSharp or Emgu CV+Structure.Sketching are superior choices.

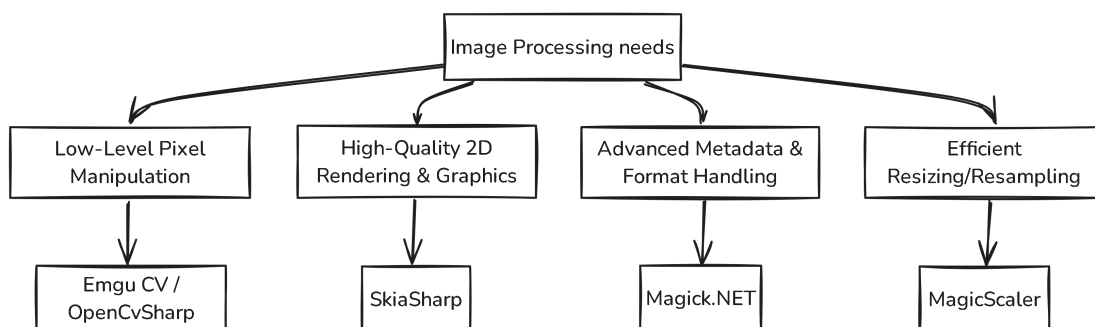


Figure 5.1: Diagram showing the ideal use cases for each library or library combination based on this study's findings.

5.3 Considerations for Future Research

Image processing is a fundamental component of many industries, including medical imaging, computer vision, digital content creation, and web applications. The performance gains demonstrated by OpenCvSharp and Emgu CV suggest that these libraries can benefit a wide range of applications, from autonomous vehicle navigation to medical diagnostics.

Moreover, the balance between speed and memory efficiency is a recurring challenge in computational imaging. This study highlights the need for hybrid approaches—such as combining OpenCvSharp with SkiaSharp to achieve optimal performance while minimizing resource consumption.

Future research could explore the following areas to further enhance the capabilities of image processing libraries:

Expanding the Scope of Benchmarking: While the study focused on image conversion and pixel iteration, real-world applications often require additional operations such as filtering, blending, and object detection. Future research could expand the benchmarking scope to include these tasks, providing a more comprehensive evaluation of each library's capabilities.

Cross-Language Compatibility: Many image processing libraries are available in multiple programming languages, such as Python, Java, and C++. Investigating the performance of these libraries across different languages could provide valuable insights into the impact of language-specific optimizations on computational efficiency.

Format-Specific Performance: Different image formats have unique compression algorithms and color spaces, which can impact the performance of image processing libraries. Future research could investigate how each library performs with specific formats, such as TIFF, BMP, or PNG, to identify any format-specific optimizations or bottlenecks.

GPU Acceleration and Parallel Processing: One limitation of this study is that all benchmarks were conducted on a CPU. Many modern image processing tasks benefit from GPU acceleration, which libraries like OpenCV support. Investigating the performance of these libraries on GPU-accelerated hardware could yield valuable insights into their scalability and efficiency.

Cloud-Based Processing: With the growing adoption of cloud computing, it would be beneficial to evaluate how these libraries perform in cloud-based environments such as AWS Lambda or Azure Functions. Factors such as cold start times, scalability, and integration with cloud-based storage solutions would be critical considerations for enterprise applications.

Further Optimizations in Memory Usage: Although Emgu CV was the fastest in pixel iteration, its high memory consumption remains a concern. Future research could explore memory optimization techniques, such as reducing redundant data structures or leveraging memory-efficient algorithms, to improve its efficiency without compromising speed.

5.4 Closing Thoughts

The findings of this study offer clear guidance for developers seeking to optimize their image processing workflows. While ImageSharp remains a user-friendly option, open-source alternatives such as OpenCvSharp and SkiaSharp provide superior performance at no cost. Emgu CV excels in computationally intensive tasks but requires careful memory management, while Magick.NET remains a powerful tool for applications prioritizing high-quality output.

Ultimately, the choice of an image processing library should be guided by the specific needs of the application. Whether prioritizing speed, memory efficiency, ease of integration, or licensing freedom, developers now have a well-defined framework for making informed decisions.

6 Appendix

Evaluation of Image Processing Libraries

This appendix provides a detailed analysis of various image processing libraries considered for the implementation phase of this thesis. Each library is evaluated based on key technical criteria, licensing considerations, and integration requirements.

1. OpenImageIO (OIIO)

- **Type:** Open-source
- **Key Features:** Supports numerous image formats, extensive image processing functionalities
- **Licensing:** Free (BSD license)
- **Performance:** Known for high performance in professional pipelines
- **Integration Effort:** Moderate, requires familiarity with C++ or Python bindings
- **Community and Support:** Active community, well-documented

There might be a need to use P/Invoke or shell out to OIIO command-line utilities if there's no direct C# wrapper.

Feature Category	Supported by OIIO	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none">- ImageInput:open: Opens an image file for reading.- ImageInput:read_image: Reads image data.- ImageOutput:create: Creates a new image file for writing.- ImageOutput:write_image: Writes the image data.- ImageBuf: Can create an empty image buffer.	<ul style="list-style-type: none">- Asynchronous image loading (no equivalent to Image.LoadAsync).

Feature Category	Supported by OIIO	Not Natively Supported / Requires Custom Implementation
Image Processing and Manipulation	<ul style="list-style-type: none"> - Basic pixel manipulation through scanlines/tiles. - ImageBufAlgo provides some algorithms for manipulation. 	<ul style="list-style-type: none"> - No built-in functions for Clone, Mutate, Resize, Grayscale. - Requires external libraries (e.g., OpenCV) or custom code for advanced processing.
Pixel Formats	<ul style="list-style-type: none"> - Supports various pixel formats including RGBA, RGB, L (grayscale), YUV, etc. - ImageBufAlgo:colorconvert allows conversion between formats. 	<ul style="list-style-type: none"> - Specific formats like Byte4, YCbCr may require manual handling.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - Provides pixel access through ImageInput:read_scanline and ImageInput:read_tile. - Can manipulate pixels by reading and writing scanlines/tiles. 	<ul style="list-style-type: none"> - No direct method like ProcessPixelRows; manual processing required.
Image Metadata and Conversion	<ul style="list-style-type: none"> - ImageSpec: Handles image metadata. - ImageBuf: Manages pixel data and metadata. - ImageBufAlgo: Offers conversion algorithms. 	<ul style="list-style-type: none"> - Handling complex metadata and conversions might require custom implementation depending on needs.
Creating and Disposing Instances	<ul style="list-style-type: none"> - ImageBuf: Creates and manages image instances. - Resources automatically managed in C++ (via destructors). 	<ul style="list-style-type: none"> - Explicit disposal may be needed for resource-intensive operations, especially in languages without automatic garbage collection.
Cropping and Resizing	<ul style="list-style-type: none"> - ImageBufAlgo:crop: Crops images. - ImageBufAlgo:resize: Resizes images with various techniques. 	<ul style="list-style-type: none"> - No direct equivalent to Mutate for fluent transformations.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - Supports encoding in multiple formats (BMP, JPEG, PNG, TIFF, WebP, etc.) via ImageOutput. 	<ul style="list-style-type: none"> - none.
Composing Image Layers	<ul style="list-style-type: none"> - ImageBufAlgo:paste: Combines image layers or tiles. 	<ul style="list-style-type: none"> - No built-in methods equivalent to ImageSharp's Stitch method.

Feature Category	Supported by OIIO	Not Natively Supported / Requires Custom Implementation
Resampling Methods	- ImageBufAlgo:resample: Provides resampling techniques.	- Lacks a direct equivalent to the IResampler interface; resampling management is manual.
Saving the Image	- ImageOutput:write_image: Saves images to a file.	- No asynchronous saving; requires standard async techniques for implementation.

2. SkiaSharp

- **Type:** Open-source
- **Key Features:** High-performance 2D graphics library, various image processing tasks
- **Licensing:** Free (MIT License)
- **Performance:** High performance, optimized for cross-platform use
- **Integration Effort:** Easy, seamless integration with .NET Core
- **Community and Support:** Active community, extensive documentation and examples
- **Restricted Countries contributors:** Need a check!

Feature Category	Supported by SkiaSharp	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - SKBitmap.Decode: Loads images from byte arrays or streams. - SKBitmap: Creates new bitmaps with specified dimensions and color types. - SKImage.FromBitmap: Creates an image from a bitmap. - SKImage.FromEncodedData: Loads images from encoded data. 	- No built-in asynchronous image loading (no equivalent to Image.LoadAsync).

Feature Category	Supported by SkiaSharp	Not Natively Supported / Requires Custom Implementation
Image Processing and Manipulation	<ul style="list-style-type: none"> - SKBitmap.Copy: Clones the bitmap. - SKBitmap.Resize: Resizes bitmaps with various filtering modes. - SKBitmap.ExtractSubset: Crops bitmaps. - SKImage.FilterImage: Applies filters like grayscale or other color transformations. - SKColorFilter.CreateColorMatrix: Provides additional color transformations. 	<ul style="list-style-type: none"> - No high-level fluent API like ImageSharp's Mutate for chaining multiple operations.
Pixel Formats	<ul style="list-style-type: none"> - Supports various pixel formats like SKColorType.Rgba8888, SKColorType.Bgra8888, SKColorType.Gray8, etc. - SKColorSpace: Manages color space conversions. 	<ul style="list-style-type: none"> - Specific formats like Byte4, YCbCr may require custom conversion.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - SKBitmap.GetPixel / SetPixel: Accesses and sets individual pixels. - SKBitmap.Pixels: Provides access to the pixel data for bulk manipulation. 	<ul style="list-style-type: none"> - No direct method like ProcessPixelRows; manual processing of pixel rows is required.
Image Metadata and Conversion	<ul style="list-style-type: none"> - SKImageInfo: Manages basic image properties such as dimensions and color type. - SKImage.Encode: Converts images into various formats like PNG, JPEG, WebP, etc. 	<ul style="list-style-type: none"> - Limited support for complex metadata handling compared to ImageSharp and OIIO.
Creating and Disposing Instances	<ul style="list-style-type: none"> - SKBitmap and SKImage: Create and manage image instances. - Proper resource management using the Dispose method is necessary to free resources. 	<ul style="list-style-type: none"> - No direct equivalent to Image<Rgba32>; pixel format and dimensions must be managed manually.

Feature Category	Supported by SkiaSharp	Not Natively Supported / Requires Custom Implementation
Cropping and Resizing	<ul style="list-style-type: none"> - SKBitmap.ExtractSubset: Crops images. - SKBitmap.Resize: Resizes images with different resampling techniques. 	<ul style="list-style-type: none"> - No built-in equivalent to ImageSharp's Mutate method for complex transformations.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - SKImage.Encode: Encodes images in multiple formats (e.g., PNG, JPEG, BMP, WebP). 	<ul style="list-style-type: none"> - Custom handling might be needed for less common formats.
Composing Image Layers	<ul style="list-style-type: none"> - SKCanvas.DrawBitmap: Composes images by drawing one bitmap onto another. - SKPicture: Records a sequence of drawing commands for later playback and compositing. 	<ul style="list-style-type: none"> - No built-in method equivalent to ImageSharp's Stitch for seamless image stitching.
Resampling Methods	<ul style="list-style-type: none"> - SKBitmap.Resize: Provides resampling techniques during resizing operations. 	<ul style="list-style-type: none"> - No direct equivalent to IResampler interface; resampling techniques are more basic.
Saving the Image	<ul style="list-style-type: none"> - SKImage.Encode: Saves images to a stream or byte array in the desired format. 	<ul style="list-style-type: none"> - No built-in asynchronous saving method; async saving requires custom implementation.

3. Magick.NET

- **Type**: Open-source
- **Key Features**: .NET wrapper for ImageMagick, extensive image manipulation capabilities
- **Licensing**: Free (Apache 2.0 License)
- **Performance**: Excellent for complex image processing
- **Integration Effort**: Moderate, straightforward API
- **Community and Support**: Large user base, comprehensive documentation
- **Restricted Countries contributors**: No

Feature Category	Supported by Magick.NET	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - MagickImage: Loads images from byte arrays, files, or streams. - MagickImageCollection: Handles multiple images, useful for formats like GIFs. - MagickImage: Creates new images with specified dimensions and colors. 	<ul style="list-style-type: none"> - No built-in asynchronous image loading (no equivalent to Image.LoadAsync).
Image Processing and Manipulation	<ul style="list-style-type: none"> - MagickImage.Clone: Clones the image. - MagickImage.Resize: Resizes images with various filtering and resampling options. - MagickImage.Crop: Crops images. - MagickImage.Resize: Resizes images with customizable resampling options. - MagickImage.Grayscale: Converts images to grayscale. - MagickImage.ColorSpace: Converts between different color spaces. - MagickImage.Rotate, MagickImage.Flip, MagickImage.Flop: Performs various image transformations. 	<ul style="list-style-type: none"> - Magick.NET supports extensive image processing, similar to ImageSharp's Mutate method. Custom implementation is rarely needed.
Pixel Formats	<ul style="list-style-type: none"> - Supports a wide range of pixel formats, including RGBA, RGB, Gray, CMYK, and more. - MagickColor: Handles color conversions and supports various color profiles. 	<ul style="list-style-type: none"> - Fully supports advanced pixel formats and color management, so custom implementation is minimal.

Feature Category	Supported by Magick.NET	Not Natively Supported / Requires Custom Implementation
Pixel Access and Manipulation	<ul style="list-style-type: none"> - MagickImage.GetPixels: Provides access to individual pixels or pixel regions. - MagickImage.SetPixels: Sets individual pixels. - MagickImage.ToByteArray: Converts pixel data to a byte array. 	<ul style="list-style-type: none"> - No direct method like ProcessPixelRows; however, pixel manipulation is flexible and powerful.
Image Metadata and Conversion	<ul style="list-style-type: none"> - MagickImage.Attribute: Accesses and manipulates image metadata such as EXIF, IPTC, and XMP. - MagickImage.Format: Converts images to various formats. 	<ul style="list-style-type: none"> - Magick.NET offers comprehensive metadata handling and format conversion.
Creating and Disposing Instances	<ul style="list-style-type: none"> - MagickImage: Creates and manages image instances. - Proper resource management using the Dispose method is necessary to free resources. - MagickImageCollection: Manages multiple image instances, useful for animations or multi-layer images. 	<ul style="list-style-type: none"> - Fully supports instance creation and disposal, with comprehensive memory management.
Cropping and Resizing	<ul style="list-style-type: none"> - MagickImage.Crop: Crops images. - MagickImage.Resize: Resizes images with customizable resampling options. - MagickImage.AdaptiveResize: Provides advanced resizing techniques. 	<ul style="list-style-type: none"> - Fully supports cropping and resizing with advanced options, no need for custom implementation.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - MagickImage.Write: Encodes images in a wide array of formats including PNG, JPEG, TIFF, BMP, GIF, WebP, and more. - MagickImage.Format: Specifies the output format. 	<ul style="list-style-type: none"> - Supports a wider range of formats than ImageSharp, with built-in encoding capabilities.

Feature Category	Supported by Magick.NET	Not Natively Supported / Requires Custom Implementation
Composing Image Layers	<ul style="list-style-type: none"> - MagickImage.Composite: Composes one image over another. - MagickImage.Mosaic: Combines multiple images into a mosaic. - MagickImageCollection: Handles layering for complex compositions. 	<ul style="list-style-type: none"> - Fully supports complex image compositions, with built-in methods for layering and merging.
Resampling Methods	<ul style="list-style-type: none"> - MagickImage.Resample: Provides advanced resampling methods and filtering options. - MagickImage.AdaptiveResize: Offers specialized resampling techniques. 	<ul style="list-style-type: none"> - Extensive support for resampling, surpassing basic needs and requiring no custom implementation.
Saving the Image	<ul style="list-style-type: none"> - MagickImage.Write: Saves images to files, streams, or byte arrays in the desired format. - MagickImage.Save: Provides simple saving options. 	<ul style="list-style-type: none"> - No built-in asynchronous saving method; async saving requires custom implementation.

4. Emgu CV

- **Type:** Open-source
- **Key Features:** .NET wrapper for OpenCV, robust image processing and computer vision
- **Licensing:** 799\$ (for version 4, with additional costs for upgrades).
- **Performance:** High performance, suitable for advanced computer vision tasks
- **Integration Effort:** Moderate to high, depending on complexity of use
- **Community and Support:** Active community, extensive tutorials
- **Restricted Countries contributors:** Need a check

Feature Category	Supported by Emgu CV	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - CvInvoke.Imread: Loads an image from a file. - CvInvoke.Imdecode: Loads an image from a byte array. - Mat: Creates a new image with specified dimensions and type. - Image<TColor, TDepth>: Generic class for creating images with specific color and depth. 	<ul style="list-style-type: none"> - No built-in asynchronous image loading (no equivalent to Image.LoadAsync).
Image Processing and Manipulation	<ul style="list-style-type: none"> - Mat.Clone: Clones the image matrix. - CvInvoke.Resize: Resizes the image with various interpolation methods. - CvInvoke.CvtColor: Converts the image to grayscale or other color spaces. - CvInvoke.Rotate: Rotates the image. - CvInvoke.Flip: Flips the image vertically or horizontally. - CvInvoke.WarpAffine: Applies affine transformations. 	<ul style="list-style-type: none"> - Emgu CV provides extensive support for image processing operations, similar to ImageSharp's Mutate method.
Pixel Formats	<ul style="list-style-type: none"> - Supports various pixel formats including BGR, RGBA, RGB, and Gray. - Mat.Depth and Mat.NumberOfChannels: Specify pixel depth and channels. - Image<TColor, TDepth>: Allows pixel data manipulation in a type-safe manner. 	<ul style="list-style-type: none"> - Default BGR format might differ from ImageSharp's RGBA, requiring format conversion in some cases.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - Mat.GetData: Accesses individual pixels or pixel regions. - Mat.SetTo: Sets individual pixels or regions with specific values. - Image<TColor, TDepth>.Data: Provides access to pixel data. 	<ul style="list-style-type: none"> - Pixel manipulation is supported, but the approach differs from ImageSharp's more abstracted methods.

Feature Category	Supported by Emgu CV	Not Natively Supported / Requires Custom Implementation
Image Metadata and Conversion	<ul style="list-style-type: none"> - CvInvoke.Imencode: Converts the image to various formats (PNG, JPEG, etc.). - CvInvoke.Imwrite: Saves images to files. 	- Does not extensively handle metadata like EXIF or IPTC, focusing more on basic properties and format conversion.
Creating and Disposing Instances	<ul style="list-style-type: none"> - Mat: Can be used to create empty images, with proper disposal via Dispose to free resources. - Image<TColor, TDepth>: Manages images with type safety. 	- Fully supports instance creation and disposal, requiring careful memory management due to OpenCV's low-level handling.
Cropping and Resizing	<ul style="list-style-type: none"> - CvInvoke.GetRectSubPix: Crops the image. - CvInvoke.Resize: Resizes the image with advanced interpolation options. 	- Emgu CV supports cropping and resizing extensively, similar to ImageSharp's Mutate method.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - CvInvoke.Imwrite: Saves images in formats like PNG, JPEG, BMP, WebP, etc. - CvInvoke.Imencode: Encodes images for various formats and uses. 	- none.
Composing Image Layers	<ul style="list-style-type: none"> - CvInvoke.AddWeighted: Blends two images together, allowing for composition. - CvInvoke.CopyMakeBorder: Combines images into a larger canvas. 	- Supports basic layer composition, though complex operations may require additional code or OpenCV functions.
Resampling Methods	<ul style="list-style-type: none"> - CvInvoke.Resize: Provides multiple resampling methods (linear, cubic, nearest-neighbor). 	- Resampling techniques are fully supported, similar to ImageSharp's capabilities.
Saving the Image	<ul style="list-style-type: none"> - CvInvoke.Imwrite: Saves images to files in the desired format. 	- No built-in asynchronous saving method; async operations require custom implementation.

5. MagicScaler

- **Type**: Open-source

- **Key Features:** High-performance image processing, optimized for resizing
- **Licensing:** Free
- **Performance:** Excellent for image resizing with high quality
- **Integration Effort:** Easy, designed for high-performance scenarios
- **Community and Support:** Active, good documentation
- **Restricted Countries contributors:** No

Feature Category	Supported by MagicScaler	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - MagicImageProcessor.ProcessImage: Loads and processes an image from a file, stream, or byte array. - ImageFileInfo: Provides basic details about the image without fully loading it into memory. 	<ul style="list-style-type: none"> - No direct method for creating new images from scratch. - Lacks asynchronous methods like Image.LoadAsync in ImageSharp.
Image Processing and Manipulation	<ul style="list-style-type: none"> - MagicImageProcessor.ProcessImage: Supports resizing, cropping, and color adjustments. - ProcessImageSettings: Allows specifying transformations such as resizing, cropping, and color adjustments. 	<ul style="list-style-type: none"> - Lacks complex manipulation like cloning, rotating, flipping, or direct pixel manipulation. - Does not support advanced image editing features found in ImageSharp.
Pixel Formats	<ul style="list-style-type: none"> - Supports various pixel formats including RGBA, RGB, Gray, etc. - Automatically handles color space conversions. 	<ul style="list-style-type: none"> - Handling of pixel formats is abstracted, with no direct pixel manipulation like in ImageSharp.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - Provides high-level processing but no direct pixel access. 	<ul style="list-style-type: none"> - No direct access or manipulation of individual pixels, unlike ImageSharp's ProcessPixelRows.
Image Metadata and Conversion	<ul style="list-style-type: none"> - Handles basic image properties and metadata. - Supports conversion between different image formats. 	<ul style="list-style-type: none"> - Limited metadata handling compared to libraries like ImageSharp, focusing more on image processing efficiency.

Feature Category	Supported by MagicScaler	Not Natively Supported / Requires Custom Implementation
Creating and Disposing Instances	<ul style="list-style-type: none"> - Focuses on processing existing images rather than creating new ones. - Managed disposal of resources. 	<ul style="list-style-type: none"> - Does not support creating images from scratch. - No manual instance creation like in ImageSharp.
Cropping and Resizing	<ul style="list-style-type: none"> - ProcessImageSettings.Crop: Specifies cropping options. - ProcessImageSettings.Width and Height: Specifies resizing dimensions. - Uses high-quality resampling algorithms for resizing. 	<ul style="list-style-type: none"> - Cropping and resizing are part of the processing pipeline, not standalone operations.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - Supports encoding images into formats like PNG, JPEG, BMP, WebP, etc. - Can control compression, quality, and other encoding parameters. 	<ul style="list-style-type: none"> - Lacks format-specific customization that ImageSharp provides through various encoders.
Composing Image Layers	<ul style="list-style-type: none"> - Not supported; focuses on single-image processing. 	<ul style="list-style-type: none"> - Lacks capabilities for composing or layering multiple images, unlike ImageSharp's Stitch or DrawImage.
Resampling Methods	<ul style="list-style-type: none"> - Provides high-quality resampling techniques for resizing. - ProcessImageSettings.Interpolation: Allows specifying the interpolation method. 	<ul style="list-style-type: none"> - Resampling is integrated into the processing pipeline, without a direct interface like IResampler in ImageSharp.
Saving the Image	<ul style="list-style-type: none"> - MagicImageProcessor.ProcessImage: Saves the processed image to a file, stream, or byte array. 	<ul style="list-style-type: none"> - No asynchronous saving method, with all operations handled synchronously.

6. SimpleITK

- **Type:** Open-source
- **Key Features:** Interface to the Insight Toolkit (ITK), simplifies complex image analysis
- **Licensing:** Free
- **Performance:** Suitable for medical and scientific image processing

- **Integration Effort:** Moderate, specialized usage
- **Community and Support:** Good community support, extensive resources
- **Restricted Countries contributors:** Need a check!

Feature Category	Supported by SimpleITK	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - sitk.ReadImage: Loads an image from a file. - sitk.Image: Creates new images with specified dimensions and pixel types. - sitk.ImportImageFilter: Loads image data from NumPy arrays. 	<ul style="list-style-type: none"> - No asynchronous loading methods like Image.LoadAsync. - More focused on scientific image formats and lacks direct support for common web image formats.
Image Processing and Manipulation	<ul style="list-style-type: none"> - sitk.Clone, sitk.Resample, sitk.Cast: For cloning, resizing, and pixel type conversion. - sitk.Transform, sitk.Crop: Supports geometric transformations and cropping. - sitk.Grayscale: Converts the image to grayscale. 	<ul style="list-style-type: none"> - Lacks the high-level abstraction of ImageSharp's Mutate for chaining operations. - More tailored to medical image processing than artistic or graphical manipulations.
Pixel Formats	<ul style="list-style-type: none"> - Supports a variety of formats, including scalar, vector, and label images. - Automatic pixel type conversions. 	<ul style="list-style-type: none"> - Limited in handling non-medical specific color spaces and formats compared to ImageSharp.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - sitk.GetPixel/SetPixel: Access and manipulate individual pixels. - Full image data accessible via NumPy arrays for batch processing. 	<ul style="list-style-type: none"> - Less intuitive pixel manipulation compared to ImageSharp's ProcessPixelRows. - Focus on scientific data rather than general-purpose image formats.
Image Metadata and Conversion	<ul style="list-style-type: none"> - sitk.Image: Handles metadata such as image origin, spacing, and direction. - sitk.Cast: Converts images to various pixel types. 	<ul style="list-style-type: none"> - Limited advanced metadata handling, focusing on medical image properties.

Feature Category	Supported by SimpleITK	Not Natively Supported / Requires Custom Implementation
Creating and Disposing Instances	<ul style="list-style-type: none"> - sitk.Image: Creates empty images. - Automatic resource management with Python's garbage collector. 	<ul style="list-style-type: none"> - Does not offer the fine-grained control over image creation seen in ImageSharp.
Cropping and Resizing	<ul style="list-style-type: none"> - sitk.Crop: Crops the image. - sitk.Resample: Provides resizing with multiple interpolation methods. 	<ul style="list-style-type: none"> - Requires more manual setup compared to the intuitive API of ImageSharp's Mutate.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - sitk.WriteImage: Supports encoding and saving images in formats like PNG, JPEG, and TIFF. 	<ul style="list-style-type: none"> - Focuses more on medical image formats, and does not natively support WebP.
Composing Image Layers	<ul style="list-style-type: none"> - Can handle multi-channel images, simulating layer composition. - sitk.LabelOverlay: Can overlay labels on grayscale images. 	<ul style="list-style-type: none"> - Lacks direct support for layer-based compositions found in ImageSharp. - Limited artistic composition tools.
Resampling Methods	<ul style="list-style-type: none"> - sitk.Resample: Offers various resampling methods with advanced interpolation options. 	<ul style="list-style-type: none"> - Resampling is powerful but less user-friendly than ImageSharp's high-level options.
Saving the Image	<ul style="list-style-type: none"> - sitk.WriteImage: Saves images to files in various formats. - Supports scientific formats like NIfTI, DICOM. 	<ul style="list-style-type: none"> - Limited optimization for modern web formats compared to ImageSharp.

7. Structure.Sketching

- **Type**: Open-source
- **Key Features**: Image transformations, filters, and format support
- **Licensing**: Free (Apache 2.0 License)
- **Performance**: Competitive, with various resampling filters and transformations
- **Integration Effort**: Easy to moderate, supports .NET Core and .NET Framework
- **Community and Support**: Growing community, good initial documentation
- **Restricted Countries contributors**: No

Feature Category	Supported by Image Package	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - png.loadPNG: Typically read an image file and decode it using the appropriate decoder from a format-specific sub-package. - image.NewRGBA: Can create new images using the image package, which provides several types like image.RGBA, image.NRGBA, image.Gray, etc. - png.Encode: Can save it using an encoder from the appropriate format-specific package. 	<ul style="list-style-type: none"> - Limited support for creating images from scratch in more diverse formats. - Requires third-party libraries for formats like WebP. - No inherent support for asynchronous operations, can achieve asynchronous behavior by running image-related operations in separate goroutines.
Image Processing and Manipulation	<ul style="list-style-type: none"> - Clone, Resize, ConvertToGrayscale: Supports advanced image processing tasks. - Transformations, Annotations: Extensive transformations and annotation capabilities. - Image Enhancement: High-level enhancement tools for noise reduction, contrast adjustment, etc. 	<ul style="list-style-type: none"> - Requires custom code implementation.
Pixel Formats	<ul style="list-style-type: none"> - image.RGB: Represents a color image with 8-bit RGBA values per pixel. - image.Gray: Represents a grayscale image with 8-bit gray values per pixel. - image.YCbCr: Represents a color image using the Y'CbCr color model, typically used in JPEG images. 	<ul style="list-style-type: none"> - Basic support for different image formats but does not explicitly handle pixel formats like image processing libraries.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - No direct methods like getpixel, setpixel or RasterImageData. 	<ul style="list-style-type: none"> - No direct pixel access, unlike ImageSharp's ProcessPixelRows.

Feature Category	Supported by Image Package	Not Natively Supported / Requires Custom Implementation
Image Metadata and Conversion	<ul style="list-style-type: none"> - Does not support comprehensive metadata handling directly. - Focuses on image manipulation and format conversion but lacks built-in tools for managing or accessing metadata like EXIF data or other detailed image properties. 	<ul style="list-style-type: none"> - Lack of support requires custom implementations.
Creating and Disposing Instances	<ul style="list-style-type: none"> - Use functions from the image package along with specific image formats (like image/png, image/jpeg). - Go's garbage collector handles memory management, so no need to manually dispose of images. 	<ul style="list-style-type: none"> - Does not directly provide CreateImage or Dispose functions as in other libraries or languages with more explicit image management.
Cropping and Resizing	<ul style="list-style-type: none"> - Does not include built-in support for more advanced operations like cropping and resizing directly. 	<ul style="list-style-type: none"> - Requires custom implementations.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - Save: Create a file using os.Create, then use png.Encoder or jpeg.Encoder to encode and save the image. 	<ul style="list-style-type: none"> - For other image formats like WebP, need to import the corresponding encoding package.
Composing Image Layers	<ul style="list-style-type: none"> - Does not directly support complex operations like combining images or drawing one image onto another. 	<ul style="list-style-type: none"> - Requires custom implementations.
Resampling Methods	<ul style="list-style-type: none"> - Does not natively support advanced resampling techniques. 	<ul style="list-style-type: none"> - Provides basic image manipulation capabilities, including resizing, but does not include advanced resampling algorithms.
Saving the Image	<ul style="list-style-type: none"> - Does not directly support saving images or providing asynchronous save functionality. 	<ul style="list-style-type: none"> - To perform asynchronous saving, need to use Go's concurrency features such as goroutines.

8. OpenCvSharp

- **Type:** Open-source
- **Key Features:** .NET wrapper for OpenCV, similar to Emgu CV

- **Licensing:** Free (Apache-2.0 license)
- **Performance:** High performance, supports a wide range of tasks
- **Integration Effort:** Moderate, different API style from Emgu CV
- **Community and Support:** Active community, good documentation
- **Restricted Countries contributors:** Need a check!

Feature Category	Supported by OpenCvSharp	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - Cv2.ImRead, Cv2.ImDecode: Loads images from files or byte arrays. - Mat, Cv2.ImCreate: Creates new images with specified dimensions and types. 	<ul style="list-style-type: none"> - No asynchronous loading methods like Image.LoadAsync in ImageSharp.
Image Processing and Manipulation	<ul style="list-style-type: none"> - Mat.Clone, Cv2.Resize: For cloning and resizing. - Cv2.CvtColor: Converts between color spaces. - Cv2.Rotate, Cv2.Flip, Cv2.WarpAffine: Provides extensive geometric transformations. - Cv2.Crop: Crops images using a Rect object. 	<ul style="list-style-type: none"> - Lacks some of the more advanced image effects and filters available in ImageSharp.
Pixel Formats	<ul style="list-style-type: none"> - Supports multiple formats, including CvType.CV_8UC3 (BGR), CvType.CV_8UC1 (Grayscale), CvType.CV_8UC4 (BGRA). - Handles automatic color space conversion. 	<ul style="list-style-type: none"> - May require custom implementations for less common pixel formats.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - Mat.At, Mat.Set: Direct pixel access and manipulation. - Mat.Data: Provides low-level access to pixel data arrays. 	<ul style="list-style-type: none"> - Less intuitive pixel manipulation compared to ImageSharp's ProcessPixelRows.

Feature Category	Supported by OpenCvSharp	Not Natively Supported / Requires Custom Implementation
Image Metadata and Conversion	<ul style="list-style-type: none"> - Mat: Stores image properties like size and type. - Cv2.ImEncode, Cv2.ImWrite: Converts and saves images in various formats. 	<ul style="list-style-type: none"> - Limited metadata handling compared to ImageSharp's extensive metadata support.
Creating and Disposing Instances	<ul style="list-style-type: none"> - Mat: Creates and initializes images. - Dispose: Required for freeing unmanaged resources. - Effective memory management through manual disposal. 	<ul style="list-style-type: none"> - More complex resource management due to reliance on unmanaged resources.
Cropping and Resizing	<ul style="list-style-type: none"> - Cv2.GetRectSubPix: For precise cropping. - Cv2.Resize: Resizing with various interpolation methods. 	<ul style="list-style-type: none"> - Lacks some of the more advanced cropping techniques like those in ImageSharp.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - Cv2.ImWrite: Saves images in multiple formats including WebP. - Cv2.ImEncode: For encoding images to byte arrays. 	<ul style="list-style-type: none"> - Less flexible format optimization compared to ImageSharp's encoders.
Composing Image Layers	<ul style="list-style-type: none"> - Cv2.AddWeighted, Cv2.Add, Cv2.Subtract: For image blending and compositing. 	<ul style="list-style-type: none"> - Less comprehensive layering system than ImageSharp.
Resampling Methods	<ul style="list-style-type: none"> - Cv2.Resize: Offers multiple interpolation methods. - Cv2.PyrUp, Cv2.PyrDown: Pyramid methods for scaling. 	<ul style="list-style-type: none"> - Fewer options for specialized resampling methods compared to ImageSharp.
Saving the Image	<ul style="list-style-type: none"> - Cv2.ImWrite, Cv2.ImEncode: Saves images to files or byte arrays. 	<ul style="list-style-type: none"> - Lacks some advanced saving options like those in ImageSharp.

9. Microsoft.Maui.Graphics

- **Type**: Open-source
- **Key Features**: Graphics functionalities across platforms, uses SkiaSharp
- **Licensing**: Free (MIT)
- **Performance**: Optimized for cross-platform use within MAUI framework

- **Integration Effort:** Easy if using MAUI
- **Community and Support:** Active support from Microsoft, good documentation
- **Restricted Countries contributors:** No

This repository has been archived by the owner on Dec 21, 2023. It is now read-only.

Feature Category	Supported by Microsoft.Maui.Graphics	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - ImageLoadingService.LoadImageAsync: Asynchronous loading from various sources. - GraphicsPlatform.CreateImage: Creates images with specified dimensions and formats. 	<ul style="list-style-type: none"> - Unlike ImageSharp, there is limited support for creating images from scratch in more diverse formats.
Image Processing and Manipulation	<ul style="list-style-type: none"> - Image.Clone, ICanvas.DrawImage: For cloning, resizing, cropping, and applying transformations. - ICanvas.SetFillColor, ICanvas.DrawRectangle: Supports color transformations and selective drawing for cropping. 	<ul style="list-style-type: none"> - Lacks extensive image manipulation tools available in libraries like ImageSharp.
Pixel Formats	<ul style="list-style-type: none"> - Supports RGBA, RGB and other common formats. - Designed for higher-level graphics tasks rather than extensive pixel format diversity. 	<ul style="list-style-type: none"> - Limited to basic pixel formats compared to the wide range in ImageSharp.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - Focuses on high-level operations, without direct pixel access. - Does not offer low-level pixel manipulation like GetPixel and SetPixel. 	<ul style="list-style-type: none"> - No direct pixel access, unlike ImageSharp's ProcessPixelRows.
Image Metadata and Conversion	<ul style="list-style-type: none"> - Handles basic image properties and conversion via Image.Save. - Can save to various formats. 	<ul style="list-style-type: none"> - Less detailed metadata handling compared to ImageSharp.

Feature Category	Supported by Microsoft.Maui.Graphics	Not Natively Supported / Requires Custom Implementation
Creating and Disposing Instances	<ul style="list-style-type: none"> - GraphicsPlatform.CreateImage: Easily creates new images. - Disposal managed via .NET's garbage collection. 	- Lacks manual resource management options, unlike some lower-level libraries.
Cropping and Resizing	<ul style="list-style-type: none"> - ICanvas.DrawImage: Supports cropping and resizing with simple interfaces. 	- Less advanced cropping and resizing techniques compared to ImageSharp.
Encoding Images in Various Formats	<ul style="list-style-type: none"> - IImage.Save: Supports encoding in multiple formats. - Provides functionality to save directly to files, streams, or byte arrays. 	- Limited format-specific encoding settings compared to ImageSharp and does not mention WebP support.
Composing Image Layers	<ul style="list-style-type: none"> - ICanvas.DrawImage: Allows layering images over one another. - Basic support for image compositing and blending. 	- More limited compositing features compared to ImageSharp.
Resampling Methods	<ul style="list-style-type: none"> - ICanvas.DrawImage: Automatically handles resampling during resizing. 	- Fewer advanced resampling methods compared to ImageSharp.
Saving the Image	<ul style="list-style-type: none"> - IImage.Save: Saves images to files, streams, or byte arrays in various formats. 	- Lacks advanced saving options and optimizations available in ImageSharp.

10. LeadTools

- **Type:** Commercial
- **Key Features:** Extensive image processing features, supports numerous formats
- **Licensing:** Paid (More than 17k\$)
- **Performance:** High performance, enterprise-grade reliability
- **Integration Effort:** Moderate to high, depending on features used
- **Community and Support:** Excellent support, extensive documentation
- **Restricted Countries contributors:** Need a check!

Feature Category	Supported by LEADTOOLS	Not Natively Supported / Requires Custom Implementation
Image Loading and Creation	<ul style="list-style-type: none"> - LoadImage, LoadAsync: Robust image loading with asynchronous support. - RasterImage.Create: Flexible image creation with control over dimensions and pixel formats. 	<ul style="list-style-type: none"> - None; LEADTOOLS provides comprehensive loading and creation features.
Image Processing and Manipulation	<ul style="list-style-type: none"> - Clone, Resize, ConvertToGrayscale: Supports advanced image processing tasks. - Transformations, Annotations: Extensive transformations and annotation capabilities. - Image Enhancement: High-level enhancement tools for noise reduction, contrast adjustment, etc. 	<ul style="list-style-type: none"> - While comprehensive, some specific manipulation tasks might require custom scripting or code.
Pixel Formats	<ul style="list-style-type: none"> - Supports multiple formats: Extensive support for various pixel formats such as RGB, RGBA, Grayscale, CMYK, and more. 	<ul style="list-style-type: none"> - None; pixel format support is exhaustive.
Pixel Access and Manipulation	<ul style="list-style-type: none"> - GetPixel, SetPixel, RasterImageData: Direct access and manipulation of pixel data. - Lock/Unlock: For precise pixel-level operations. 	<ul style="list-style-type: none"> - Pixel data manipulation can be complex, especially when locking/unlocking is required.
Image Metadata and Conversion	<ul style="list-style-type: none"> - Comprehensive Metadata Handling: Extracts, edits, and writes metadata for a wide range of formats. - Advanced Conversion: Extensive format support including DICOM. 	<ul style="list-style-type: none"> - None; LEADTOOLS excels in metadata and conversion capabilities.
Creating and Disposing Instances	<ul style="list-style-type: none"> - CreateImage, Dispose: Facilitates creation and proper resource disposal of images. 	<ul style="list-style-type: none"> - Proper disposal requires careful management, similar to .NET libraries.
Cropping and Resizing	<ul style="list-style-type: none"> - Crop, Resize: Advanced cropping and resizing with various resampling methods. 	<ul style="list-style-type: none"> - None; cropping and resizing are robustly supported.

Feature Category	Supported by LEADTOOLS	Not Natively Supported / Requires Custom Implementation
Encoding Images in Various Formats	<ul style="list-style-type: none"> - Save: Encodes images into numerous formats including WebP, including advanced options for compression and format-specific settings. - Multimedia and Document Formats: Extends to multimedia and document encoding. 	<ul style="list-style-type: none"> - Format-specific settings might require additional configuration.
Composing Image Layers	<ul style="list-style-type: none"> - Combine, DrawImage: Supports complex image layering with detailed control over blending and transparency. 	<ul style="list-style-type: none"> - Layer management might need additional custom code for complex use cases.
Resampling Methods	<ul style="list-style-type: none"> - Advanced Resampling: Provides high-quality resampling techniques during resizing. 	<ul style="list-style-type: none"> - Resampling methods are extensive but need careful selection for optimal results.
Saving the Image	<ul style="list-style-type: none"> - Save, SaveAsync: Saves images with extensive control over parameters. - Asynchronous Saving: Supports performance-enhanced operations. 	<ul style="list-style-type: none"> - None; saving features are comprehensive and powerful.

Bibliography

- [1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice Hall, 2008, google-Books-ID: 8uGOnjRGEzoC.
- [2] A. K. Jain, *Fundamentals of digital image processing*. Englewood Cliffs, NJ : Prentice Hall, 1989. [Online]. Available: <http://archive.org/details/fundamentalsofdi0000jain>
- [3] J. C. Russ, *The Image Processing Handbook*. CRC Press, Apr. 2016, google-Books-ID: gxXXRJWfEsoC.
- [4] R. Szeliski, "Introduction," in *Computer Vision: Algorithms and Applications*, R. Szeliski, Ed. Cham: Springer International Publishing, 2022, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-030-34372-9_1
- [5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, Nov. 2016, google-Books-ID: Np9SDQAAQBAJ.
- [6] G. R. Bradski, *Learning OpenCV: computer vision with the OpenCV library*. Sebastopol, CA : O'Reilly, 2008. [Online]. Available: <http://archive.org/details/learningopencvco0000brad>
- [7] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965, publisher: American Mathematical Society. [Online]. Available: <https://www.jstor.org/stable/2003354>
- [8] G. N. Hounsfield, "Computerized transverse axial scanning (tomography): Part 1. Description of system," *British Journal of Radiology*, vol. 46, no. 552, pp. 1016–1022, Dec. 1973. [Online]. Available: <https://doi.org/10.1259/0007-1285-46-552-1016>
- [9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, place: United Kingdom Publisher: Nature Publishing Group.
- [10] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," Jul. 2012, arXiv:1207.0580 [cs]. [Online]. Available: <http://arxiv.org/abs/1207.0580>
- [11] D. Zhang, X. Hao, D. Wang, C. Qin, B. Zhao, L. Liang, and W. Liu, "An efficient lightweight convolutional neural network for industrial surface defect detection," *Artificial Intelligence Review*, vol. 56, no. 9, pp. 10 651–10 677, Sep. 2023. [Online]. Available: <https://doi.org/10.1007/s10462-023-10438-y>

- [12] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. v. d. Laak, B. v. Ginneken, and C. I. Sánchez, “A Survey on Deep Learning in Medical Image Analysis,” *Medical Image Analysis*, vol. 42, pp. 60–88, Dec. 2017, arXiv:1702.05747 [cs]. [Online]. Available: <http://arxiv.org/abs/1702.05747>
- [13] M. Maimaitijiang, “Soybean yield prediction from UAV using multimodal data fusion and deep learning,” *Remote Sensing of Environment*, Jan. 2020. [Online]. Available: https://www.academia.edu/84238554/Soybean_yield_prediction_from_UAV_using_multimodal_data_fusion_and_deep_learning
- [14] J. Janai, F. Güney, A. Behl, and A. Geiger, “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art,” Mar. 2021, arXiv:1704.05519 [cs]. [Online]. Available: <http://arxiv.org/abs/1704.05519>
- [15] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” Jan. 2016, arXiv:1506.01497 [cs]. [Online]. Available: <http://arxiv.org/abs/1506.01497>
- [16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines*, Jun. 2013, vol. 48, journal Abbreviation: ACM SIGPLAN Notices Pages: 530 Publication Title: ACM SIGPLAN Notices.
- [17] R. Szeliski, “Image Processing,” in *Computer Vision: Algorithms and Applications*, R. Szeliski, Ed. Cham: Springer International Publishing, 2022, pp. 85–151. [Online]. Available: https://doi.org/10.1007/978-3-030-34372-9_3
- [18] S. J. Russell, P. Norvig, E. Davis, and D. Edwards, *Artificial intelligence a modern approach*, third edition, global edition ed. Boston: Pearson, 2016. [Online]. Available: <http://www.gbv.de/dms/tib-ub-hannover/848811429.pdf>
- [19] Z. Kulpa, “Universal digital image processing systems in europe — A comparative survey,” in *Digital Image Processing Systems*, L. Bloc and Z. Kulpa, Eds. Berlin, Heidelberg: Springer, 1981, pp. 1–20.
- [20] A. Sahebi, M. Barbone, M. Procaccini, W. Luk, G. Gaydadjiev, and R. Giorgi, “Distributed large-scale graph processing on FPGAs,” *Journal of Big Data*, vol. 10, no. 1, p. 95, Jun. 2023. [Online]. Available: <https://doi.org/10.1186/s40537-023-00756-x>
- [21] X. Ma, Y. Jiang, H. Liu, C. Zhou, and K. Gu, “A New Image Quality Database for Multiple Industrial Processes,” Feb. 2024, arXiv:2401.13956 [cs]. [Online]. Available: <http://arxiv.org/abs/2401.13956>
- [22] T. Chisholm, R. Lins, and S. Givigi, “FPGA-Based Design for Real-Time Crack Detection Based on Particle Filter,” *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 5703–5711, Sep. 2020, conference Name: IEEE Transactions on Industrial Informatics. [Online]. Available: <https://ieeexplore.ieee.org/document/8888239>

- [23] D. Ferreira, F. Moutinho, J. P. Matos-Carvalho, M. Guedes, and P. Deusdado, "Generic FPGA Pre-Processing Image Library for Industrial Vision Systems," *Sensors (Basel, Switzerland)*, vol. 24, no. 18, p. 6101, Sep. 2024.
- [24] B.-C. Lai, Phillip, and P. McKerrow, "Image Processing Libraries," Jan. 2001.
- [25] J. Pérez, E. Magdaleno, F. Pérez, M. Rodríguez, D. Hernández, and J. Corrales, "Super-Resolution in Plenoptic Cameras Using FPGAs," *Sensors*, vol. 14, no. 5, pp. 8669–8685, May 2014, number: 5 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/1424-8220/14/5/8669>
- [26] M. N. Rao, "A Comparative Analysis of Deep Learning Frameworks and Libraries," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 11, no. 2s, pp. 337–342, Jan. 2023, number: 2s. [Online]. Available: <https://ijisae.org/index.php/IJISAE/article/view/2707>
- [27] R. A. Ciora and C. M. Simion, "Industrial Applications of Image Processing," *ACTA Universitatis Cibiniensis*, vol. 64, no. 1, pp. 17–21, Nov. 2014. [Online]. Available: <https://www.sciendo.com/article/10.2478/aucts-2014-0004>
- [28] Y. J. Sandvik, C. M. Futsæther, K. H. Liland, and O. Tomic, "A Comparative Literature Review of Machine Learning and Image Processing Techniques Used for Scaling and Grading of Wood Logs," *Forests*, vol. 15, no. 7, p. 1243, Jul. 2024, number: 7 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/1999-4907/15/7/1243>
- [29] H. Sardar, "A role of computer system for comparative analysis using image processing to promote agriculture business," *International journal of engineering research and technology*, Nov. 2012. [Online]. Available: <https://www.semanticscholar.org/paper/A-role-of-computer-system-for-comparative-analysis-Sardar/6e2fd48a1025b68951f511abe05f8451f753eb47>
- [30] R. Vieira, D. Silva, E. Ribeiro, L. Perdigoto, and P. J. Coelho, "Performance Evaluation of Computer Vision Algorithms in a Programmable Logic Controller: An Industrial Case Study," *Sensors*, vol. 24, no. 3, p. 843, Jan. 2024, number: 3 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/1424-8220/24/3/843>
- [31] S. Wu, H. Yang, X. Liu, and R. Jia, "Precision control of polyurethane filament drafting and winding based on machine vision," *Frontiers in Bioengineering and Biotechnology*, vol. 10, Sep. 2022, publisher: Frontiers. [Online]. Available: <https://www.frontiersin.org/journals/bioengineering-and-biotechnology/articles/10.3389/fbioe.2022.978212/full>
- [32] Q. Zhu, Y. Zhang, J. Luan, and L. Hu, "A Machine Vision Development Framework for Product Appearance Quality Inspection," *Applied Sciences*, vol. 12, no. 22, p. 11565, Jan. 2022, number: 22 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2076-3417/12/22/11565>

- [33] M. J. C. S. Reis, “Developments of Computer Vision and Image Processing: Methodologies and Applications,” *Future Internet*, vol. 15, no. 7, p. 233, Jul. 2023, number: 7 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/1999-5903/15/7/233>
- [34] M. Ziaja, P. Bosowski, M. Myller, G. Gajoch, M. Gumiela, J. Protich, K. Borda, D. Jayaraman, R. Dividino, and J. Nalepa, “Benchmarking Deep Learning for On-Board Space Applications,” *Remote Sensing*, vol. 13, no. 19, p. 3981, Oct. 2021. [Online]. Available: <https://www.mdpi.com/2072-4292/13/19/3981>