CODE UNIVERSITY OF APPLIED SCIENCES

Software Engineering

Bachelor's Thesis

The environmental impact of programming language choice

Name: Paula Dettmann
Address: Paulsborner Straße 91, 10709 Berlin
E-mail: paula.dettmann@code.berlin

Date of Submission: 07.06.2022

Student ID   19.03.024
Semester: Spring 2022

First Assessor: Prof. Dr. Adam Roe
Second Assessor: Dipl-inf. Fabio Fracassi

Climate change is a problem that humanity needs to combat. Software engineers can contribute to the fight against global warming by writing energy-efficient software. This thesis explores which environmental impact the choice of programming language can have. By writing image processing programs in C++ and Python and benchmarking them using different configurations of software and hardware, we found that while a programming language in isolation can affect the energy efficiency of a program, writing software does not happen in isolation. Therefore, every configuration of software and hardware affects the energy efficiency of a program. If one part is not optimised, it can increase the total runtime of the program by a significant amount. Therefore, we can conclude that the choice of programming language can only impact the runtime if all other factors of the program's configuration are optimised.

Contents

List of Figures

iv

List of Listings

Chapter 1

Introduction

This thesis is about the environmental impact of the choice of programming language. Every operation on a computer takes up energy. Extrapolating the usage of a single computer to a global operation, the required energy of single operations becomes significant. Therefore, software engineers need to optimise their programs to take up the least amount of energy possible. This thesis researches which part the choice of programming language plays in energy efficiency. Other research has been done on the most efficient programming languages, testing several algorithms in different languages and determining the most efficient ones, which proved to be C, Rust and C++. We set out to experiment with a more realistic program, with disk I/O (disk input and output) and other operations as well as an algorithm. We tested the same program in Python and C++ and benchmarked the runtime of each operation in the program. According to a study at Queens University in Ontario, Canada, programmers have limited knowledge about energy efficiency, lack knowledge about the best practice to reduce the energy consumption of software, and are often unsure about how software consumes energy [12]. Therefore, our research is essential to perhaps ultimately serve as a recommendation on how to reduce energy usage by making a programming language choice. At the very least, to make developers think about the environmental impact of their software by putting forth observations on its environmental impact. This research hypothesises an environmental impact on the choice of programming language. This hypothesis will be proved or disproved through a series of experiments where we benchmark the same

image processing program in C++ and Python in different hardware and software configurations. We aim to observe which language runs fastest. First, we aim to give a general introduction to green computing and the programming languages we will be exploring. A small part of the paper will report previous research on the energy efficiency of programming languages and how we utilised this research in this thesis. Another chapter will illustrate the research hypothesis and explain how it can be proved or disproved. The next chapter will cover the research methods, including the code and profilers. Then we will display the results of the experiments that were done and discuss the results in the following chapter. Lastly, there will be a conclusion and bibliography.

Chapter 2

Green Computing

Power electronics are an ever-increasing part of the energy sector. Therefore, a way to conserve energy to aid in the fight against global warming has emerged: green computing. Green computing is the environmentally responsible and eco-friendly use of computers and their resources [10]. Green computing first started when the US Environmental Protection Agency launched Energy Star in 1992, a label that stands for energy efficiency in monitors, climate control equipment, and other technologies [5]. Later, Swedish TCO development launched the TCO certified program to promote low magnetic and electrical emissions from CRT-based computer displays, energy consumption, ergonomics, and the use of hazardous materials in construction. It is still active, 30 years later, constantly updating its criteria according to technological improvements [21]. There are several aspects to green computing. First of all, the software itself, meaning algorithm optimisation and programming language choice. Choice of deployments can also play a big part. For example, it matters where servers that developers choose to run their code on are located because colder climates and locations close to a water source save energy needed for cooling. Additionally, it is important what kind of energy is used, namely fossil or renewable energy. Another aspect is the make of the servers and physical tools that a developer uses. Some servers are more efficient than others, and some are made with recycled or eco-friendly materials. The aspect that we would like to focus on in this paper is green software.

2.1 Quantifying environmental impact

One standard to measure environmental impact is using the Greenhouse Gas Protocol (GHG). The GHG is a "comprehensive global standardised framework to measure and manage greenhouse gas (GHG) emissions from private and public sector operations, value chains and mitigation actions." [6] Companies like "AMD, Apple, Facebook, Google, Huawei, Intel, and Microsoft publish annual sustainability reports using the GHG Protocol."[7, p.855] The GHG categorises emissions into three scopes. "Scope one emissions come from fuel combustion (e.g., diesel, natural gas, and gasoline), refrigerants in offices and data centres, transportation, and the use of chemicals and gases in semiconductor manufacturing." [7, p.856]. Scope two emissions "come from purchased energy and heat powering semiconductor fabs, offices, and data-centre operation." [7, p.856] These emissions are relevant to software engineers. Scope three "emissions come from all other activities, including the full upstream and downstream supply chain. They often comprise employee business travel, commuting, logistics, and capital goods". [7, p.856] The scope that we can influence with software is scope two. Using AI as an example, a paper by U. Gupta (et al.) called Chasing Carbon: The Elusive Environmental Footprint of Computing describes that "algorithmic optimisations for scale-down systems will drastically cut emissions." [7, p.863] We can reason that there is a direct correlation between the speed of a program and the amount of data centres. When a program is less optimised, it runs for a more extended period, occupying data centres for a longer time. Therefore, the demand for data centres rises, causing companies to build more data centres, a CO2 intensive process. According to a paper by Md Abu Bakar Siddik, Arman Shehabi and Landon Marston, "approximately 0.5% of total US greenhouse gas emissions are attributed to data centres" [20].

Chapter 3

Programming Languages

Programming languages are different levels of binary abstraction, the ones and zeroes that make up the computer's language. Commands to the computer are written in a particular language and then compiled into binary. These compilers are different for each language. Each language also has a different garbage collection form, redundant data handling, and other specifics. Therefore, each language has certain specialities and operates at different speeds. We make a distinction between high level and low-level languages. High-level languages compile automatically at runtime and have built-in garbage collection. Some low-level languages require developers to handle garbage collection themselves and compile their code into an executable file before running it. That makes lower-level languages more customisable and, as a result, often faster. Therefore, we consider low-level languages like C++ and Rust to be faster for most use-cases, and higher-level languages like Python and JavaScript are considered slower. For this thesis, we chose to pit C++, a low-level language and Python, a high-level language, against each other to explore what difference the choice of language makes on the same application in terms of speed and energy efficiency.

3.1   C++

C++ is based on the traditional C language. We consider C++ a low-level language because it lacks automatic memory management. Many different compilers are

available for C++, making it a portable language. "C++ compiles directly to a machine's native code, allowing it to be one of the fastest languages in the world if optimised" [1].

```
→  ~ clang --version
Apple clang version 13.1.6 (clang-1316.0.21.2)
Target: x86_64-apple-darwin21.4.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/Xco
deDefault.xctoolchain/usr/bin
```

Figure 3.1: clang version on MacBook Pro

We are using an Apple MacBook computer for the research in this thesis. The Operating System (OS) is macOS Monterey, which ships with the C++ compiler clang, as seen in figure 3.1. "The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project" [9]. Clang is also used in production to build software like Chrome or Firefox. The clang compiler is incorporated into the macOS native Xcode IDE (integrated development environment), which we used for the research in this thesis.

3.2   Python

Python is an open-source, interpreted, object-oriented programming language. "It incorporates modules, exceptions, dynamic typing, very high-level dynamic data types, and classes." It runs on many operating systems like macOS, Windows and Linux. The language comes with a sizeable standard library for many different purposes. For example, Python is often used for experimenting and logging, string processing and internet protocols like HTTP. Besides the standard library, there is also a wide variety of third-party extensions available [15].

Chapter 4

Previous Research

We based our choice of programming languages on a study that ranked different programming languages by their energy efficiency. The study pitted ten programming languages against each other, using "rigorous and strict solutions to 10 well-defined programming problems, expressed in (up to) 27 programming languages, from the well known Computer Language Benchmark Game repository." [13, abstract] Results of the study showed that C++, C and Rust were the most energy-efficient languages and time efficient, with Pascal and Go scoring higher on memory efficiency, as seen in figure 4.1. Python consistently scored in second to last place for energy and time efficiency and in 12th place (out of 27 languages) for memory efficiency. We chose to use C++ based on the libraries available for the program we wanted to write and because it was one of the best performing languages in this study. We chose Python because it was one of the three worst-performing languages in the study, and we were already familiar with it.

Total		

	Energy (J)
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time (ms)
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Figure 4.1: Normalised global results for Energy, Time, and Memory [13, p.16]

Chapter 5

Research Hypothesis

This thesis's research hypothesis is that there is an environmental impact on the choice of programming language for a particular application. The hypothesis will be proved correct if one programming language performs measurably better than another. We will measure this by benchmarking the same image processing application in two languages. Observations will be made on the speed of the different parts of the application; the image blurring algorithm, the disk input, the disk output and other operations. We can also measure the overall energy consumption of the program. Testing the hypothesis will be done according to the scientific method. This means performing an experiment and collecting data in a reproducible manner. The experiment will be controlled, meaning that all variables besides the programming language will be the same as far as that is possible. By this method, we can ensure that the result of the experiment pertains to the programming language and no other factors such as the library, OS, IDE or profiler. A weakness of our study might be that it is not always possible to eliminate all variables because of compatibility issues with the programming languages. For example, a profiler that works for C++ will not work in the same capacity for Python code and, therefore, might produce slightly different results. The results of this research could guide environmentally conscious developers on how to reduce their environmental impact for an image processing project.

Chapter 6

Methods

In order to prove the hypothesis, we wrote a small program to blur images. The kind of program was not important, but it was necessary for it to take a significant amount of time and make a measurable impact on the CPU. We benchmarked the program using profilers. The profiler data shows where delays in the code are located and which language is faster at executing the same procedure. This information allows us to conclude which language, for the purpose of image processing in the form of a Gaussian blur, is more efficient and therefore more environmentally friendly.
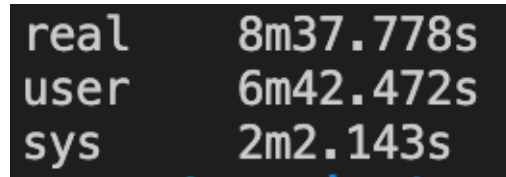
6.1 Hardware

Most of the experiments were performed on an Apple MacBook Pro with a 1,4 GHz Quad-Core Intel Core i5 processor. The computer had 16 GB of RAM and 251 GB of flash storage. It was running version 12.3.1 of the MacOS Monterey operating system.

6.2 Profilers

A profiler is a benchmarking tool. It measures the runtime of a program, and other statistics, depending on the profiler. To benchmark our Python program on macOS, we used a combination of xctrace and cPython. For the C++ code on macOS, we exclusively used xctrace.

6.2.1 *time*

At first, we used the macOS *time* command to measure the runtime of the Python program, as seen in figure 6.1.

Figure 6.1: An example of the *time* command

time is a macOS shell keyword that measures the total time spent running the program. The user row shows the time spent in the CPU in user mode, meaning "it does not have direct access to hardware or reference memory and must rely on APIs of the system for delegation. This is how most code runs on your system, and due to its isolation, crashes are always recoverable" [8]. sys stands for system mode or kernel mode. Kernel mode is "when code being executed has unrestricted access to the system hardware" [8]. The first row, called real, is the total time spent in the two modes.

6.2.2 XCode Instruments

In order to keep the research controlled, we decided to use the same profiler for both programs; Xcode instruments. Xcode is Apple's IDE and contains Instruments amongst other developer tools. "The Instruments app, which is included with Xcode, gathers data from your running app and presents it in a graphical timeline. With Instruments, we can gather data about performance areas such as our app's memory usage, disk activity, network activity, and graphics operations" [3]. The command-line tool for Instruments is called xctrace, and it is used to record, import, export and symbolicate Instruments .trace files. These .trace files can then be opened in the Instruments app and analysed with the graphical user interface. We mostly used the CPU Counter template. It allowed us to track how much time was spent on each part of the program; the algorithm, disk input, disk output and retrieving file names.

6.2.3 cProfile

cProfile is a profiler that is included in the Python standard library. It is a C extension with a reasonable overhead which makes it suitable for profiling long-running programs. It was based on lsprof and contributed by Brett Rosen and Ted Czotter [17]. cProfile measures the total time the program runs, the number of function calls, and the runtime and function calls of each individual function. The measured statistics can be displayed in the command-line interface or a text file.

6.3 Energy benchmarking

We can measure the energy used by the programs using the Intel® Power Gadget. It is a software-based power usage monitoring tool enabled for Intel® Core™ processors. It is supported on Windows and macOS and includes an application, driver, and libraries to monitor and estimate real-time processor package power information in watts using the energy counters in the processor [22]. The command-line tool PowerLog allows us to measure the energy usage of a program and write it to a log file. Specifically, the log file will include the elapsed time, package power limit, processor frequency, GT (Energy of the processor graphics) frequency, processor temperature, and average and cumulative power of the processor.

6.4 Code

We are experimenting with the same algorithm in both languages: Gaussian Blur. In image processing, a Gaussian blur is a result of blurring an image by a two-dimensional Gaussian function which was named after mathematician and scientist Carl Friedrich Gauss. Applying the algorithm to the image causes a blurring effect on the image, the intensity of the blur effect depending on variables like the kernel and standard deviation. We made sure to write the code for the two languages to execute as similar as possible.

6.4.1 Python

```python
#!/usr/bin/env python

import cv2 as cv
import os
import cProfile

pr = cProfile.Profile()
pr.enable()

img_names = os.listdir("/Volumes/RAMDisk/raw")

for name in img_names:
    path = "/Volumes/RAMDisk/raw/{}".format(name)

    if ".DS_Store" in path:4
        continue

    img = cv.imread(path)
    blur = cv.blur(img,(5,5))

    cv.imwrite("/Volumes/RAMDisk/python_blurred/blurred_" + name, blur)

pr.disable()
pr.print_stats()
```

Listing 1: The Python program

Looking at listing 1, os is a module that provides a portable way of using operating system dependent functionality [16]. It is used in line 10 to list all image names in the raw images directory. In line 12, a for loop is used to iterate through that list of names, therefore every picture can be blurred, one at a time. Every picture first is "read" in line 18, after determining the path in line 13 and dismissing a possible macOS bug in case the file ".DS_Store" exists in the directory in lines 15 and 16. In line 19, the Gaussian blur filter is applied to the image using the OpenCV function with parameters of the image variable and Gaussian kernel size. Line 21 shows the blurred image being written to the disk. In listing 1, we can also see the profiler cProfile being used. It is imported in line 5 and initialised in line 7. Line 8 enables the profiler and line 23 disables it so it is wrapped around the code we want

to profile. In line 24 the collected statistics are printed to the console or a file.

```cpp
1   #include <opencv2/opencv.hpp>
2   #include <iostream>
3   #include <string>
4   #include <filesystem>
5   #include <vector>
6
7   using namespace cv;
8   using namespace std;
9
10  namespace fs = std::__fs::filesystem;
```

Listing 2: The C++ program requirements

Listing 2 shows the beginning of the C++ code. First, we import the OpenCV and C++ standard libraries and assign the namespaces.

```cpp
1   int main(int argc, char* argv[])
2   {
3     if (argc < 3) {
4       cout << "Usage: ./green_apple_blur <input_path> <output_path>" << endl;
5       return 1;
6     }
7
8     string base_path = argv[1];
9     string ds = ".DS_Store";
10
11    for (const auto & file : std::__fs::filesystem::directory_iterator(base_pat
12    {
13      fs::path path = file.path();
14      string filename = path.filename().string();
15
16      if (find(path.begin(),path.end(),ds)!=path.end())
17        {
18          continue;
19        }
```

Listing 3: Reading the image file

In listing 3, the path to the images is added as a parameter to the main function when the program is run in line 1. To prevent a bug we ignore the file ".DS_Store" in the path in line 16.

```
1    // Read the image file
2    Mat image = imread(path);
3
4    // Check for failure
5    if (image.empty())
6      {
7        cout << "Could not open or find the image" << endl;
8        cin.get(); //wait for any key press
9        return -1;
10     }
11
12   //Blur the image with 5x5 Gaussian kernel
13   Mat image_blurred;
14   GaussianBlur(image, image_blurred, Size(5, 5), 0);
15
16   string writePath = argv[2];
17
18   string blur_path = writePath + "/blurred_" + filename;
19   bool isSuccess = imwrite(blur_path, image_blurred); //write the image to
20
21   if (isSuccess == false)
22     {
23       cout << "Failed to save the image" << endl;
24       cin.get(); //wait for a key press
25       return -1;
26     }
27   }
28
29   return 0;
30 }
```

Listing 4: Processing and writing the image file to the disk

In line 2 of listing 4, we read the image file with the OpenCV imread function. In line 14 we blur the image with the GaussianBlur filter function from OpenCV [11]. Size (5, 5) is the Gaussian kernel size, it should be an odd number. The last zero controls the sigma value which is the standard deviation in the X direction and the Y direction of the Gaussian distribution. It will be calculated based on the size of the kernel [19]. In line 19 we save the image to the path specified in the parameter and check whether that succeeded. At the end, we return 0 to terminate the function successfully.

Chapter 7

Results

In order to discern whether the choice of programming language has an environmental impact, we did experiments, each time changing a different aspect of the experiment to see if there is a significant difference between the performance of the languages. Most experiments used the programs to blur 78 high definition images.

7.1   C++ and Python with OpenCV

The first experiment we did was to run our programs with the chosen profiler of xctrace and cProfile to determine the runtime of each function in the program, blurring 78 images in total. We ran each benchmark five times and then calculated the average. We see the result of these experiments in figure 7.1.
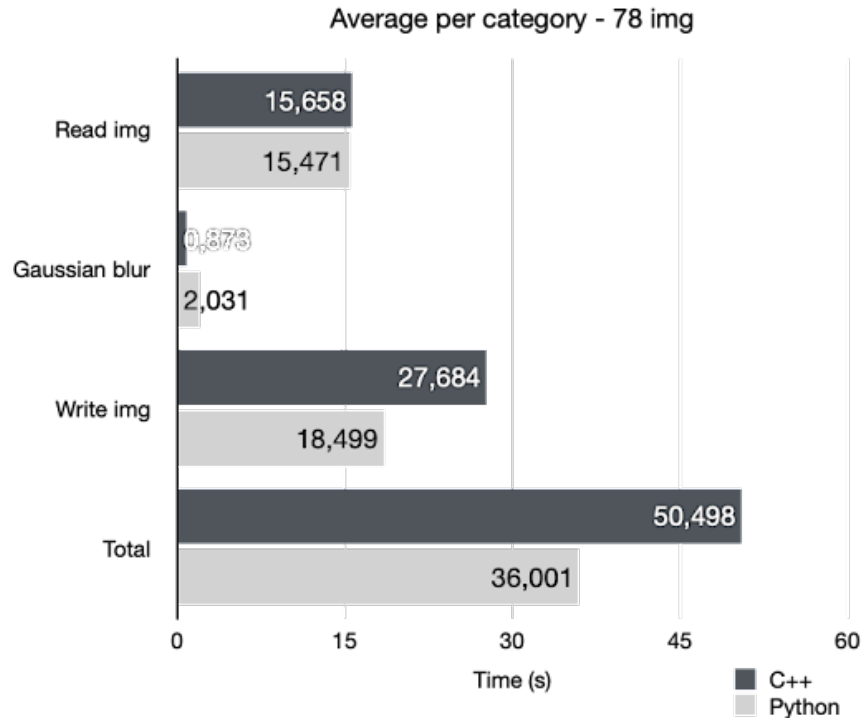
Figure 7.1: The average of 5 runs for each language

These results were highly uncharacteristic. Based on previous research, we expected C++ to be faster than Python, not slower. Most of the delay for C++ seems to come from writing the blurred image to the disk, as seen in the write img bars in figure 7.1.

7.2   Saving to BMP instead of JPEG

One approach to solving the significant delay of C++ when writing images to the disk was to change the file type of the image for both languages. The default settings of the OpenCV package saved the image to whichever file type the image had initially been; in our case, that was JPEG (Joint Photographic Experts Group). Changing the file extension from JPEG to BMP (bitmap image file) would change the image file type on saving the image. Strangely, as seen in figure 7.2, this warped the data in such a way that reading the image now took longer for C++ and writing the image was quite a bit faster. We assume this might be attributed to the library version, but further research is needed to confirm this.
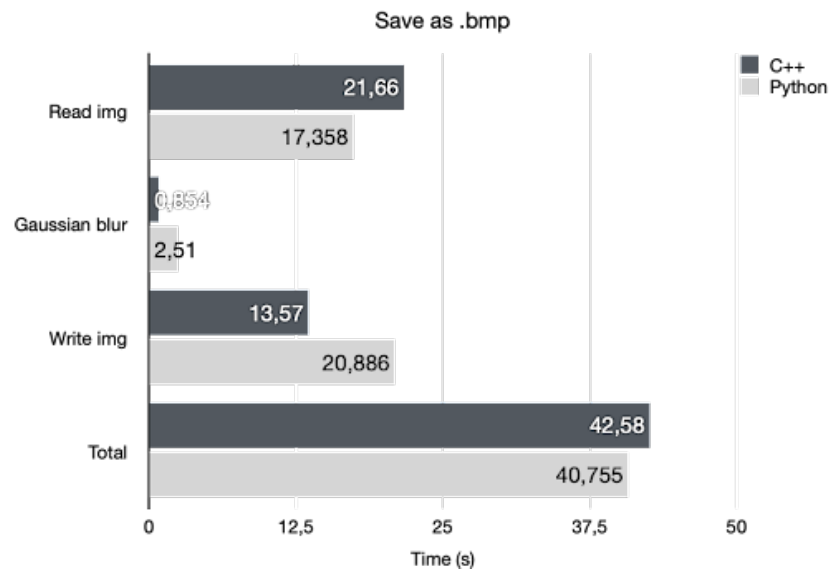
Figure 7.2: Saving the image as a BMP file

7.3 Using a script to benchmark

The following experiment was to benchmark the programs using a script. A script is a sequence of instructions that are carried out subsequently by a computer, as seen in figure 7.3.

Figure 7.3: The script used to profile the different programs

We did this to minimise the time between running each benchmark and, therefore, minimise the difference between computer operations. It also allowed us to randomise the different programs to see how one program running before or after another might affect its performance.
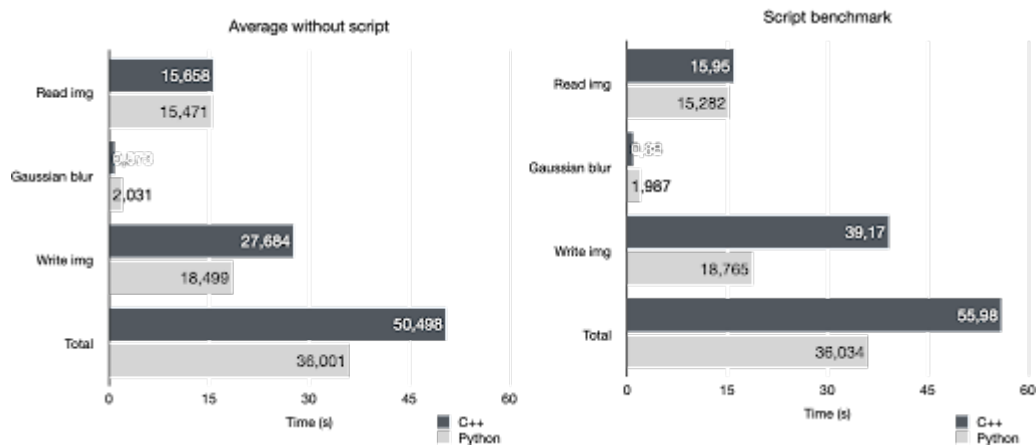
Figure 7.4: Comparing the results of a script versus no script

We can see in figure 7.4-left, when compared to figure 7.4-right, that there is a slight difference between the readings when using a script. This is especially visible when writing the image in C++. We might attribute this to the system load caused by running all programs so quickly after each other.
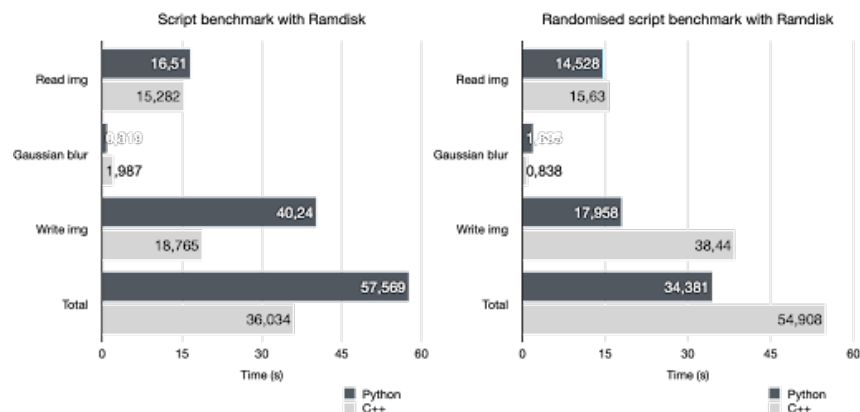

Figure 7.5: Comparing two different scripts

Figure 7.5 shows two benchmarks with a different script order, both using a RAMdisk. Each was measured by blurring 78 images. There is a slight difference between the two benchmarks, the randomised one running about two seconds faster. Overall, the proportions of the different benchmarks stayed the same.

7.4 RAMdisk

A RAMdisk is a block of random-access memory that a computer's software is treating as if the memory were a disk drive. It can be used to accelerate the reading and writing of files. We installed a RAMdisk using the command "diskutil erasevolume HFS+ "RAMDisk" 'hdiutil attach -nomount ram://2097152'" in the terminal. This creates a RAMdisk with 1 GB of storage space.
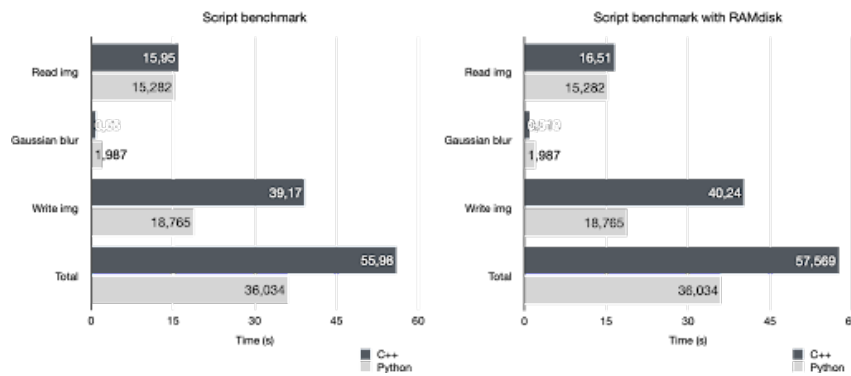

Figure 7.6: Comparing benchmarks without and with the use of a RAMdisk using the same script

As seen in figure 7.6, compared to the benchmarks with no RAMdisk, running the scripted benchmarks with a RAMdisk surprisingly returns a decreased performance. However, the difference is so slight that we cannot confirm whether this is due to the RAMdisk or another influence.

7.5 Running the programs on a different computer

In order to find out whether the computer hardware influenced the disk I/O, we decided to test the programs on a different computer. We utilised the same computer type in a different configuration: the MacBook Pro from 2021 with an M1 Pro Apple Silicon CPU, 32 GB of RAM and a 1 TB SSD. To run the programs on this new configuration, we needed to install Python version 3.10.3 instead of the previously used 3.8.12. Our software was not compatible with the new M1 chip, so it had to be recompiled for Apple Silicon.
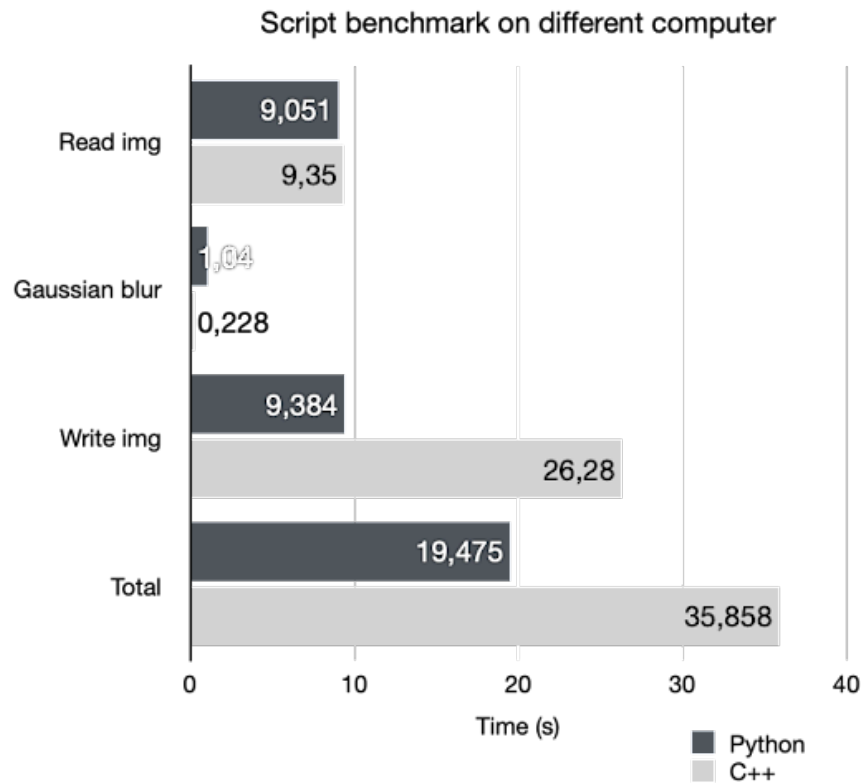
Figure 7.7: Script benchmark on a different computer

Figure 7.7 shows a similar distribution of runtime for the benchmark but an overall acceleration of 15 to 20 seconds compared to the other computer. The results are expected since this computer is a newer model and has more powerful hardware.

7.6 Profiling C++ and Python on Linux

In order to rule out the possibility of the operating system influencing the unexpectedly slow C++ runtime, we decided to experiment with our programs on a different operating system. We created an Ubuntu Linux instance using a virtual machine to benchmark our programs. This required some adjusting as the executable made for macOS was not compatible with Linux, and Linux obviously does not support the macOS instruments profiler. We compiled the C++ code with the cmake tool to create the new executable. We used the GCC (GNU Compiler Collection) compiler native fprofile command as a profiler. For Python, it was still possible to use the cProfile profiler. We ran each program five times and calculated the average as seen in figure 7.8.
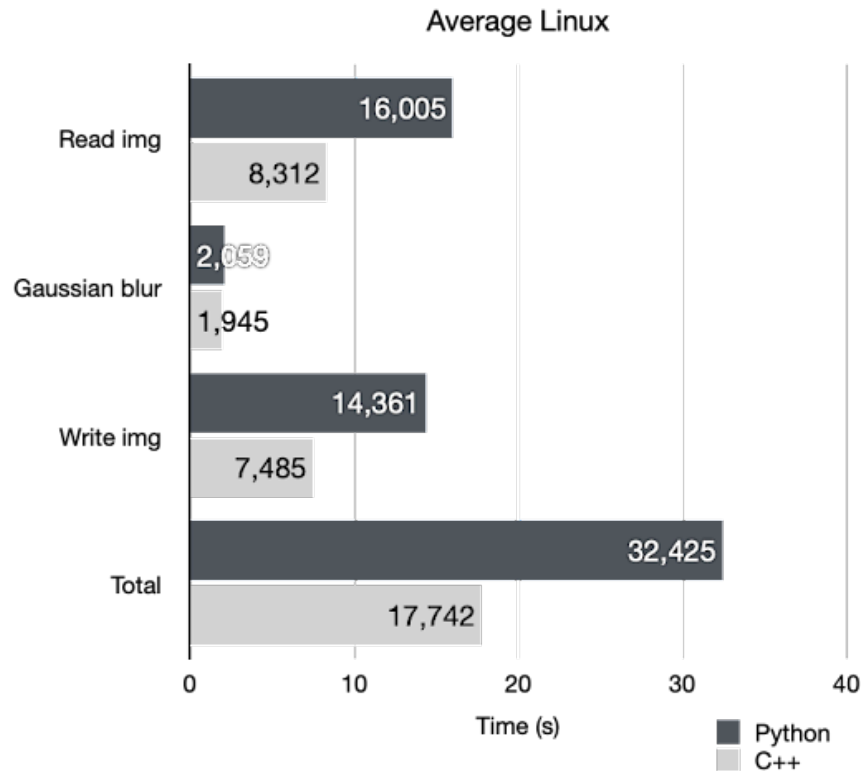
Figure 7.8: The average for the programs run on Linux

Interestingly, on Linux, the C++ program performed a lot faster than Python, 14,683 seconds faster, to be exact. This might once again be attributed to OpenCV being optimised differently for a different operating system and therefore performing better on Linux.

7.7 C++ using OpenCV from VCPKG

One theory we had for the delay of C++ for disk output was that it was due to how the OpenCV library function handled writing to the disk. After doing further research, we found that the standard OpenCV package for Python is a different version from what is standard for C++. Using VCPKG, a C/C++ dependency manager [23], we were able to download the same version as used for Python and use it with C++. Benchmarking the programs with 100 randomly generated images shows the results in figure 7.9. Now, we can see only a very insignificant difference between the two programming languages compared to before.
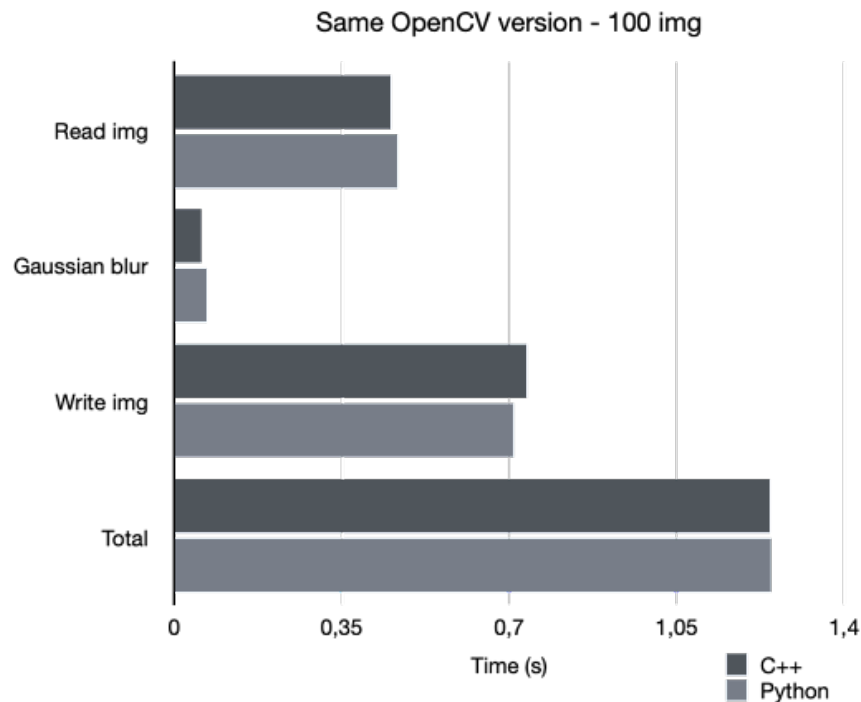
Figure 7.9: Average of results using the same OpenCV version

We can see that the total runtime here is significantly less than when using the previous images. This can probably be attributed to the randomly generated images, which seem to have a lower resolution and are therefore faster to be processed. However, this is not important for this study as only the comparison of C++ and Python is of any consequence and not the overall performance.

7.8   The error interval

An error interval is a property of the instrument and the user and will remain the same for all readings taken, provided the scale is linear [14]. It can be determined by finding the fraction of the smallest readable division on the instrument. In the case displayed in figure 7.10, benchmarking Python has yielded the error intervals in the last row, which is the same for each reading, using the same instrument.

	List all images (s)	Read img (s)	Gaussian blur (s)	Write img (s)	Total (s)
1	0,001	15,602	2,032	18,422	36,058
2	0	15,472	2,039	18,510	36,022
3	0	15,460	2,016	18,484	35,960
4	0	15,382	2,052	18,567	36,002
5	0,001	15,437	2,014	18,512	35,964
Average	0,0004	15,4706	2,0306	18,499	36,0012
Error interval	0,0005	0,0005	0,0005	0,0005	0,0005

Figure 7.10: Python profiler average with error interval

From this, we can determine that if we have a reading of, e.g. 18,500, with an error interval, it is actually between 18,4995 and 18,5005. Whilst this allows us to have a realistic interpretation of our measurements, it also does not affect our observations since all proportions are still the same.

7.9 Energy consumption

To confirm that runtime was congruous with energy consumption, we used the Intel Power Gadget to approximately measure the energy consumption of each program, both using the standard form of OpenCV with no RAMdisk. C++ had a runtime of 58,233 seconds in the experiment, and Python had a runtime of 38,889 seconds.
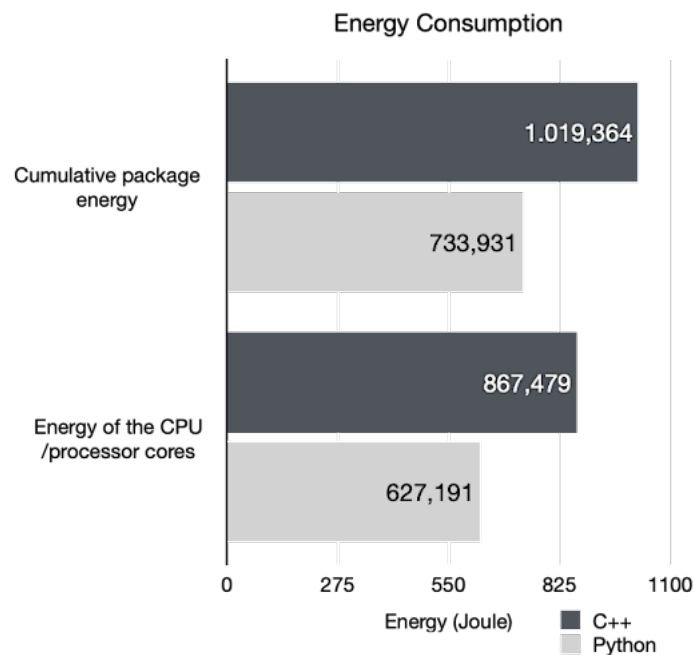

Figure 7.11: Energy consumption of the programs in Joule

We can see in figure 7.11 that the proportion of runtime to energy consumption is approximately the same. C++ used more energy than Python for the same program. Python's runtime is 66,78% of the C++ runtime, and Python's energy consumption is 72,00% of C++'s energy consumption overall and 72,30% of the energy of the CPU/processor cores. These results indicate that runtime is an approximate indicator of energy consumption.

Chapter 8

Discussion

Having experimented with many different configurations of the same program for both languages, we can make several observations. These observations are made only based on the image blurring programs for C++ and Python and cannot be generalised. For one, we can say that on the MacBook Pros used for the experiments, Python performs faster by about 14 seconds than C++ for the standard OpenCV versions. Secondly, we can say that there is a difference in disk I/O for different image file types, which differ for the different languages. We also observed that using a script can provide more consistent data, and randomising said script can provide different results. Using a RAMdisk for disk I/O changes the total runtime, but the function runtimes' proportions stay the same. When running the programs on a different MacBook Pro, we observed a faster runtime overall but similar proportions yet again. Running the programs on a different operating system causes different results. On Linux, C++ ran approximately 14,7 seconds faster than Python. This is the direct opposite of running the programs on macOS. When running the scripted C++ program on macOS, we observed that we needed to change the version of the OpenCV library to measure similar results to the scripted Python program. Lastly, we primarily focus our research on measured proportions, so the error interval does not hold weight in this study. These observations tell us that we cannot conclude from these experiments that the programming language itself makes a massive difference in optimisation. However, we can conclude that each minor change to the configuration of a program can make a big difference in the runtime and, therefore,

the environmental impact.

8.1 Influencing factors

One part of the configuration of our program is the operating system. Operating systems control the computer's hardware and manage software resources. There are many different types of operating systems, such as real-time, multitasking and embedded operating systems which all operate differently and therefore affect running a program differently. Computers also have varying hardware. They can have a CPU, RAM or disk space of different sizes and speeds. They can also have different hardware altogether, such as having or not having an SSD (solid-state drive). For example, our standard MacBook Pro did not have an SSD, whilst the other MacBook Pro used for an experiment did have one, which made a difference in speed that might be attributed to that SSD. We could also observe that different types of images might have influenced the runtime of the program. We used high-resolution space images for most experiments and we used automatically generated images for the experiment running OpenCV from VCPKG on C++, as seen in figure 8.1.
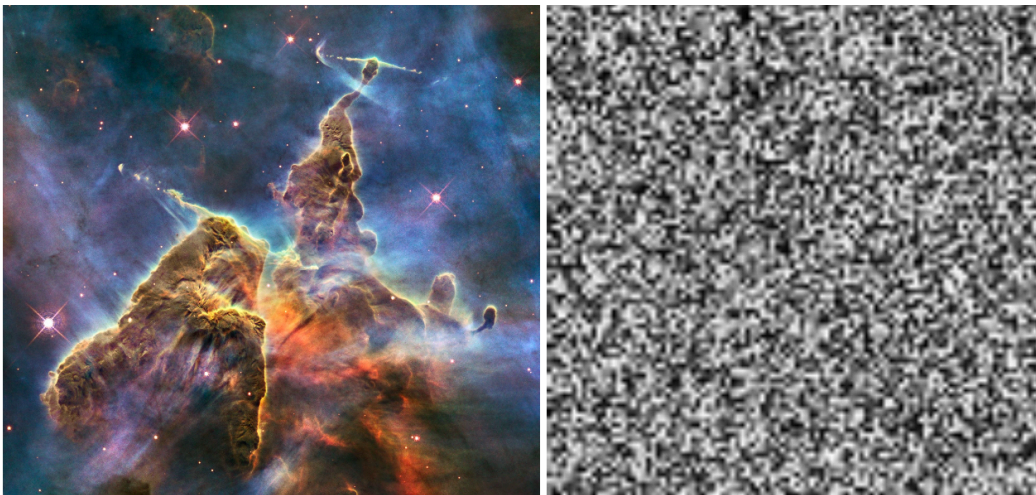

Figure 8.1: high-resolution image (left) next to a randomly generated image (right)

We can see that image 8.1-left is a bit more complex than image 8.1-right, having colours and higher resolution. This means that blurring image 8.1-right might have been a bit faster than image 8.1-left, represented by the shorter runtime for

the experiment running OpenCV from VCPKG on C++. We observed a significant change in performance regarding different versions of a library. We also experimented with a different image processing library for Python called scikit-image [18]. Scikit-image is written predominantly in Python. As seen in figure 8.2, scikit-image caused Python to perform significantly worse than C++ with OpenCV. Whereas when Python did use OpenCV, it performed better than C++. This proves to a certain extent that the type and version of a library are essential factors in optimisation.
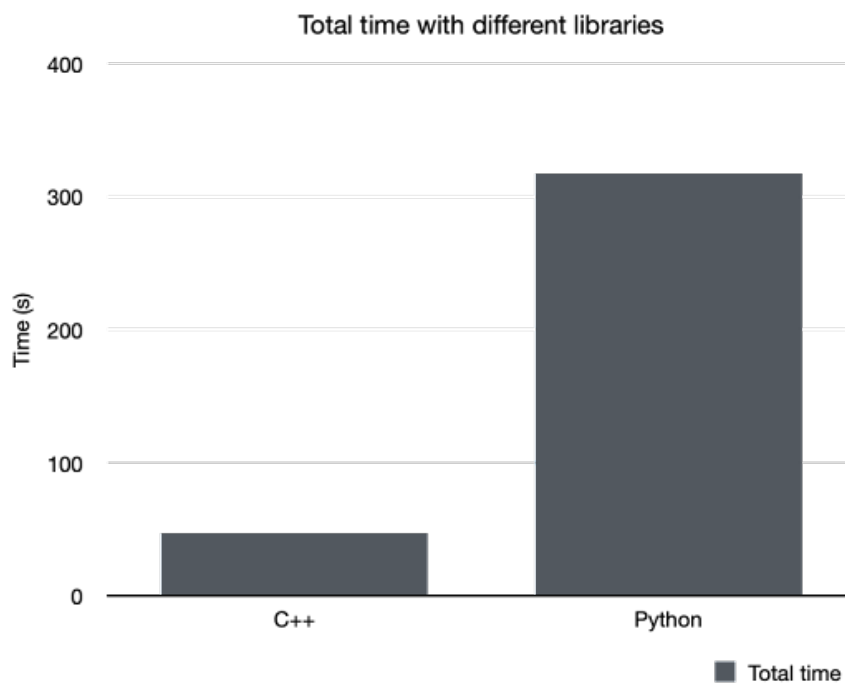

Figure 8.2: Python total runtime using scikit-image to blur 79 images

Another factor that influences the runtime of a program is how the libraries are linked. The compilation of C++ involves three steps. First is preprocessing, which handles the preprocessor directives, such as "#include" and "#define" and, explained simply, replaces them with the content of the respective files. The next step is compilation, which takes the preprocessor's output and produces an object file. The last step is linking. The linker takes the object files produced by the compiler and produces either a library or an executable file [2]. Linking can take two different forms. For one, static linking is the process of copying all library modules used in the program into the final executable image. Dynamic (or shared) linking

adds the names of the libraries in the executable and then links them at runtime. Programs that use statically linked libraries are usually faster than those using shared libraries. When looking at the profile for C++, dyld was shown as taking up a lot of the total runtime. Dyld is a macOS pre-linking of dynamic libraries. Chandler Caruth, a principal software developer at Google, mused in a Twitter feed about the general slow runtime of a program on macOS as compared to Linux and found that it was due to spending a vast amount of time on dyld [4]. This matches what we observed with the xctrace profiler CPU counter. Of course, the programming language also plays a role in program optimisation. We already saw from previous research that the programming language can influence the runtime for a specific algorithm. Lack of knowledge of the language can cause it to be written in a way that delays the runtime. Additionally, in real-life programming, programmers hardly ever write the entirety program themselves; they use external libraries, packages or models. Choosing the wrong library for our configuration can lead to significant delays, as we have proved in this study. Therefore, choosing a language for a project might be based on available libraries but should also be chosen on the characteristics of the language and if they are optimised for the project. In conclusion, we can say that every program needs its own particular configuration of hardware, OS, language, compilation, external libraries and whichever other factors might contribute.

Chapter 9

Conclusion

This research aimed to identify the environmental impact of the choice of programming language. Based on a series of experiments, we can conclude that the choice of programming language can only impact the runtime if all other factors of the program's configuration are optimised. We benchmarked the same program in two different languages and in a multitude of different configurations to find which impact the language would make. We expected to see C++ perform better than Python for the image processing program. The results did not match our expectations, and experiments were done to explain these results. These experiments resulted in the observation that minor differences in the configuration of a program can make a big difference in total runtime. A limitation of the research is the scale. Provided with more time and resources, it would have been interesting to experiment with the program on more operating systems, different hardware, and more library variations. Additionally, to better understand the pure impact of the programming language, it would have been interesting to forgo libraries altogether and write all functionality from scratch to receive ultimate control over the program. Another approach could be to experiment with more programming languages to find the nuances that set them apart for different purposes. This research contributes the knowledge that when we want to make an environmental impact with our software, we need to optimise every part of the configuration of a program, including choosing the best programming language for the purpose.

Bibliography

[1] Albatross. A Brief Description, 2022.

[2] Alex Allain. Compiling and Linking, 2019.

[3] Apple Inc. Measure Energy Impact with Instruments, 2018.

[4] Chandler Carruth. no title, 2022.

[5] ENERGYSTAR. ENERGY STAR Overview, 2022.

[6] GHG Protocol. About Us, 2022.

[7] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[8] joshtronic. How To Time Command Execution with the time Command, 2019.

[9] LLVM. Clang: a C language family frontend for LLVM, 2022.

[10] Maria Salama. Green Computing, a contribution to save the environment, 2020.

[11] Open Source Computer Vision. GaussianBlur(), 2022.

[12] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about the energy consumption of software? 07 2015.

[13] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[14] Physics Wiki. MEASUREMENTS AND ERRORS, 2013.

[15] Python Software Foundation. General Python FAQ, 2022.

[16] Python Software Foundation. os — Miscellaneous operating system interfaces, 2022.

[17] Python Software Foundation. The Python Profilers, 2022.

[18] scikit-image development team. Scikit Image, 2022.

[19] Shermal Fernando. Gaussian Blur, 2019.

[20] Md Abu Bakar Siddik, Arman Shehabi, and Landon Marston. The environmental footprint of data centers in the united states. *Environmental Research Letters*, 16(6):064017, May 2021.

[21] TCO Certified. TCO Certified — like an ecolabel but so much more, 2022.

[22] Timothy McKay and Patrick Christian Konsor. Intel® Power Gadget, 2019.

[23] VCPKG. vcpkg, 2022.

Affidavit

I hereby confirm that I have written the thesis titled "The environmental impact of programming language choice" by myself, without contributions from any sources other than those cited in the text and bibliography. This also applies to all graphics, drawings, and images included in the thesis.

Furthermore, I confirm that neither this work nor parts of it have been previously or concurrently used as an assessment submission in other courses or in other examination proceedings.

Berlin, 07.06.2022,