CV-Lab Container

Bachelor's Thesis Image Processing and Computer Vision FS23

Andreas Kuhn

June 2, 2023

Abstract

Problem

After initial prototyping of computer vision algorithms in Python, it often makes sense to implement the time critical section of the algorithm in C/C++, to be able to assess the full performance potential. For development of computer vision applications in general and especially for a mixed language workflow as described above, many different tools and technologies are involved. Building, configuring and integrating these technologies into a cohesive development environment for the project specific needs is an elaborate, time consuming and system dependent task, which has made exploring this area of computer vision development burdensome in practical labs if the computer software is being updated or changed.

Approach

The Docker containerization technology is used to package the complex toolchain and the practicality of such a solution is evaluated. To streamline development on the container, options for accessing and interacting with the container are evaluated and integrated. The development environment is extended beyond Docker by including a CI/CD pipeline on GitHub, where the integrity of the container as well as the application developed within it are tested.

Result

Docker was proven to be a valid solution for the posed problem as well as for streamlining computer vision software development in general. The final solution provides a local-quality development environment including access to local hardware such as the GPU, while simultaneously being highly portable. Two container images were built: The first is a lightweight solution providing the base toolset for developing mixed language projects using OpenCV, CMake and Pybind11. The second image additionally integrates the CUDA Toolkit as well as the OpenCV CUDA modules for building custom CUDA kernels to be executed on a Nvidia graphics card. For each of the containers, a sample application was developed, showcasing usage as well as the efficacy of C++ and CUDA acceleration.

Contents

1 Introduction

Context

In the practical labs for the IPCV course, students currently use Python to implement image processing algorithms. Because of its ease of use and high development speed, this is generally a good fit as lab time is limited and Python is highly relevant in image processing and computer vision to begin with. However after initial prototyping in Python, in practice it often makes sense to implement only the critical section of the algorithm in C/C++, to be able to assess performance potential of the final algorithm. Combining the simplicity of Python for handling data and the performance of C++ for algorithm execution. Such a mixed language workflow comes at the cost of a more complex development environment with more system dependent dependencies and significant configuration effort. Maintaining such an environment on lab PCs which undergo frequent updates and changes to the system setup, has so far been infeasable, making exploring this area of algorithm development in the practical labs impossible.

Goals

This work aims to remove these hurdles by providing a stable, easy to use and portable development environment for integrating C++ into Python. Using this same environment, an example image processing application shall be implemented to serve as reference for students as well as to demonstrate the performance gain motivating the use of C++ in the first place.

Methods

The Docker containerization technology is used to package the complex toolchain and the practiclity of such a solution is evaluated. To streamline development on the container, options for accessing and interacting with the container are evaluated and integrated. The development environment is extended beyond Docker by including a CI/CD pipeline on GitHub, where the integrity of the container as well as the application developed within it are tested.

1.1 Toolchain Requirements and the Case for the Docker Container

The primary use-case for the development environment toolchain to be created are the IPCV courses' practical labs. Aside from the considerations regarding the software development itself, this imposes some additional requirements, which fall into two categories: Those imposed by the lab infrastructure and those imposed by the intended usage in lab exercises.

From the infrastructure standpoint, the toolchain needs to be...

Infrastructure requirements

- **Portable:** The environment will be used by many different users and on many machines.
- **Infrastructure independent:** The PC setups may be slightly different from user to user and may also change over time.
- **Self-contained:** Minimal external dependencies to maximize reliability and minimize upkeep.

For use in practical lab exercises, it needs to be...

Lab exercice requirements

- **Straight forward:** Add as few extra steps as possible to get the development environment up and running.
- **Familiar:** Ideally, the student will find some level familiarity to the way he's been working throughout the rest of the courses' labs to be able to work efficiently.

For the software development itself we consider:

Technical requirements

- **Operating System:** Switch to a Linux OS, as this is the standard in the image processing industry. Benefits include its package manager (apt-get), better support from the toolchains used, faster compile times and more.
- **Toolchain Packaging:** Have the entire toolchain and dependencies for mixed language development installed, configured and ready to build with.
- **Performance:** To remain real-time viable, the way the environment is packed should not get in the way of performance.
- **GPU enabled:** Ideally, parallel computing platforms such as CUDA or OpenGL can be used.

To be able to switch to a Linux OS on lab PCs, some sort of virtualization is needed. A full Virtual Machine, however is neither very portable due to the large file sizes, nor particularly performant. In recent years however, a new virtualization concept has become increasingly interesting for problems such as this: The Docker container. Particularly for the use on Windows hosts, a lot of advancements have been made recently, which promise to be able to cover all the points listed above.

Docker Container

2 Working with Docker

2.1 Overview

What is a Docker container

In a broad sense, a Docker container can be understood as a very lightweight virtual machine(VM). However the goal of containers is usually to *containerize* just a single application to make it system independent and easily deployable. Any machine that can run a Docker daemon will be able to run the container with no additional setup. Another interesting prospect is to build the app directly on the container, using the container as a development environment. This makes it very easy to share your exact environment with others, which is exactly what we are interested in doing in this project.

Container vs Virtual Machine

Aside from how they are used, there are some important conceptual differences between VMs and Docker containers: A traditional VM virtualizes the hardware using a*hypervisor*, where each virtualized instance contains a full guest operating system, including the kernel abstracting the hardware to the operating system. Containers, on the other hand, virtualize the operating system, and that all hosted containers share the operating system kernel with the host. The container itself only needs to include a minimal guest OS, or none at all, if the host is using the same UNIX distribution. Sharing the system kernel also gives the container complete access to the system's hardware resources. While a VM would be allocated a fixed amount of CPU, RAM and disk space, a container will use as few or as many resources as necessary[1] at a given time, just like a regular application would.

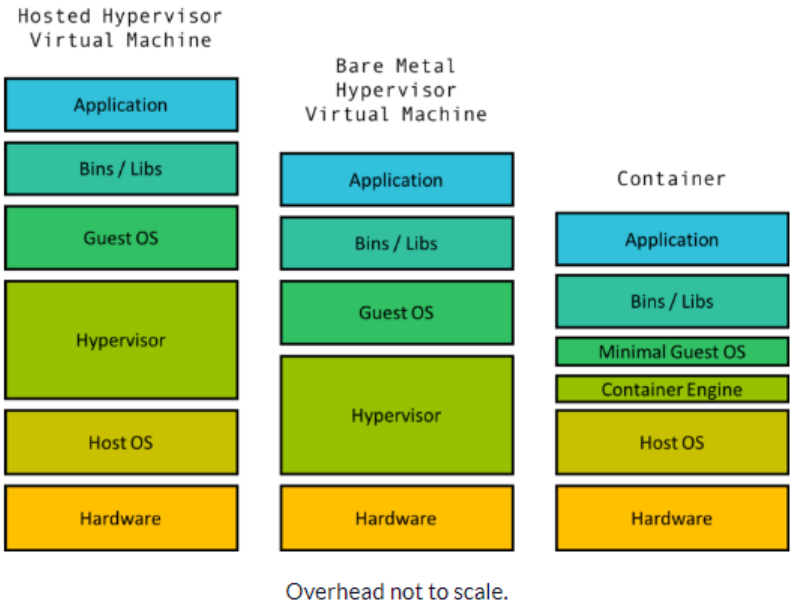


Figure 1: Technology Stack for virtual machines and Docker container on UNIX Systems. Source: freecodecamp.org [10]

Linux Container on Windows host

In our case, the host is given as a Windows machine but a container running Linux is required. This isn't what containers were originally intended for, but Microsoft has put in a lot of effort to support this use-case with their WSL(Windows Subsystem for Linux) technology. Particularly the upgrade to WSL2, developed in collaboration with Docker,

[1] Being able to use all available CPU cores when compiling large libraries such as OpenCV comes in very handy.

has made using Docker containers in this fashion much more interesting[8]. The main enabler is the switch to a full Linux Kernel running alongside Windows with full system call capacity. This way, a Linux based Docker container can use a native(rather than virtual) Linux Kernel, just like it would on a Linux host. At the same time, the deep integration into the Windows environment allows the Linux and Windows systems to work with the same file system.

WSL2 Linux Kernel

The Linux Kernel is hosted using Windows' Hyper-V type 1 hypervisor. A type 1 hypervisor, also known as a bare-metal hypervisor, interacts with the hardware directly. As such, WSL2 can offer near-native performance. NVIDIA reports a minimal overhead performance loss of less than 1% in conjunction with their CUDA GPU acceleration technology[1]. So while the container's advantage over VMs of not needing a hypervisor is lost when running non-native operating systems, WSL2 is so well integrated, that basically no overhead or performance loss results from its reintroduction into the tech stack.

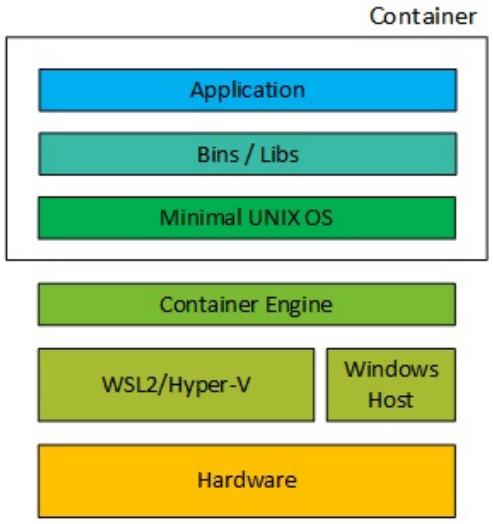


Figure 2: Technology Stack for our use-case: Linux Container on WSL2.

remaining advantages: No configuration required, images still much smaller, CUDA, docker ecosystem.

2.2 Architecture

right

The Docker platform consists of three core components:

- The **Docker daemon** does the heavy lifting of building and managing images as well as managing the container lifecycle
- The **client** takes the form of a command line interface(CLI) used to instruct the daemon to execute the various tasks mentioned in the previous bullet [2].
- The **registy** is where pre-built images can be obtained that are tailored for different purposes, with software like Python or SQL already installed.

Docker uses a client-server architecture, where the daemon acts as the server. In our case Docker Desktop is used, which runs both client and server on the same machine. Although interestingly, the daemon actually runs on the WSL2 system, and the CLI from which we access it on Windows.

The daemon accepts commands given by the client to execute tasks such as running images, building new images from a Dockerfile, pulling images from the remote registry (Docker Hub repository) or pushing local images to your own repository.

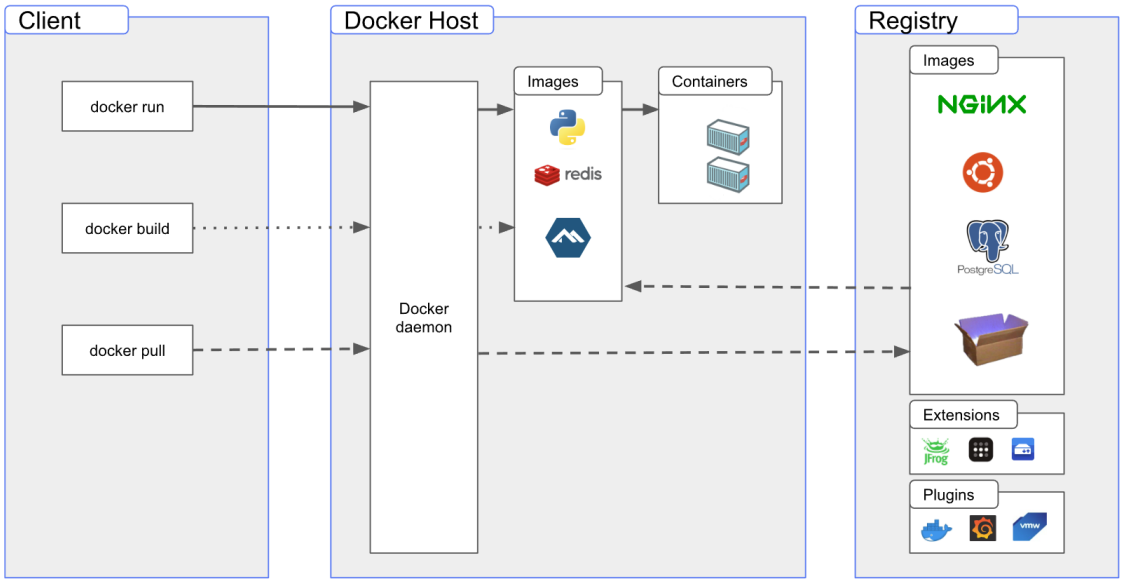


Figure 3: Architecture and common tasks illustrated. Solid: Creating a container from a locally available image; Dotted: building a new image from a Dockerfile; Dashed: pulling a new image from the registry to make it available locally. Source: Docker documentation [2]

Images and Containers

Similarly to a regular VM image, Docker images encapsulate the initial state of the virtual environment when first booted up. Unlike traditional VM images though, Docker images don't get altered by changes done inside the running instance. In Docker, running an image using `Docker run` creates a new object called the *container*. Changes *do* save to the container, but not the image.

The altered container can also be turned back into an image which will include the

[2]The later discussed feature 'docker-compose' actually also acts as a client.

changes made using the `Docker commit` command. However if the image needs to be rebuilt to change something else, the changes from the commit are lost, since they are not reflected in the Dockerfile. Hence this isn't part of the main Docker workflow, but sometimes useful while developing.

Docker commit

Images can be pulled from remote repositories using `Docker pull` or built from a *Dockerfile* using `Docker build`. The Dockerfile contains an ordered list of instructions executed when building the image starting with a base image in the `FROM` clause. If it is not available locally from a previous pull, this image will be pulled from the Docker Hub repositories based on the tag given. The `FROM` line alone will already yield an image that can be run and used.

Building Docker images

```
Dockerfile
1   FROM python:3.10
```

Figure 4: Absolute minimum example Dockerfile for building the Python base image available on Docker Hub with the *tag* '3.10', which denotes the Python version to be installed.

The lines following the initial `FROM` clause will build on this base image by installing additional software, copying files and configuring the environment further.

Each line in the Dockerfile typically corresponds to a *layer* of the built image, which associates all the memory affected in the image to the Dockerfile line. The order of the Dockerfile commands and in turn the order of the image layers can become quite important, if there are commands which take a long time to execute[3] because of *build caching*.

Image layers

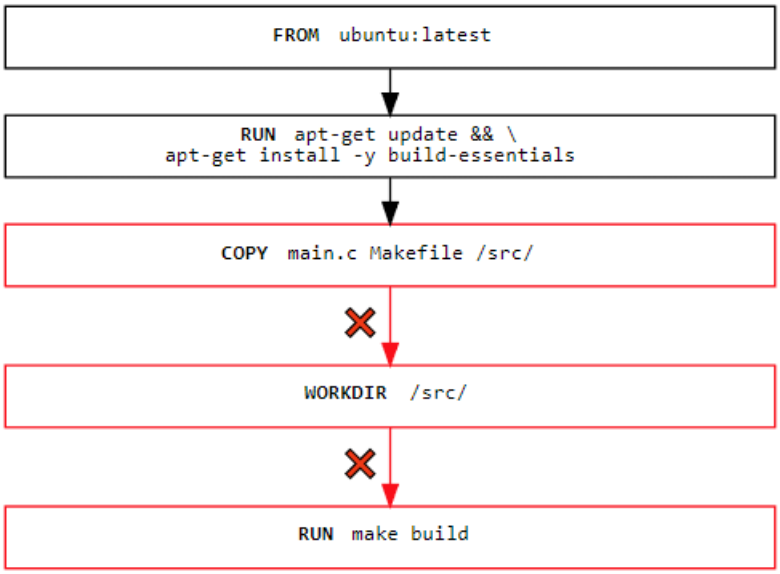


Figure 5: `COPY` command invalidating following layers, forcing 'make build' to be executed, even if nothing in `src` has changed. Source: Docker Documentation [3]

After building an image, the docker daemon caches each layer locally. When instructed to build another image, it checks for each layer, if it corresponds to one of the cached

Build caching

[3]For example building OpenCV from source.

layers and if so, reuses it instead of building it again to speed up the process of rebuilding images. However, once there is a layer that can't be loaded from cache, all subsequent layers will also be rebuilt no matter what, because the daemon can't be certain that the changes in the first uncached layer won't affect the later ones, since the layers are build in order. The `COPY` command, for example, will always invalidate all following layers as shown in figure 5, because it is not known at build time, what effect the copied files will have on following layers. Hence, `COPY` instructions should always be as far down the Dockerfile as possible.

2.3 Basic Workflow, Containerizing an Application

Here, as a first hands-on example, the workflow for containerizing an application is demonstrated for a minimal Hello World Python application. Although we aren't really interested in using Docker to deploy and application in this project, this is the easier first introduction to the Docker concepts and will serve as a stepping stone towards the end goal of building a development Container.

HelloWorld.py

```
1 print("Hello World, from inside the container!")
2
```

Figure 6: Python test "application" to containerize.

Containerizing Hello World

Containerizing simply means, building a container capable of running the application by copying the application files onto the container. For this simple example the only tool necessary is Python, which will be taken care of by using a base image from Python's own Docker Hub repository. The Dockerfile always lives at the root of the project directory to ensure that the entire project directory is inside the build-time context[4]for copying files etc.

Host Directory Tree

```
1 project_directory
2 ├── HelloWorld.py
3 ├── requirements.txt
4 └── Dockerfile
5
```

Dockerfile

```
1 FROM python:3.10
2
3 COPY . /app
4 WORKDIR /app
5
6 CMD ["python", "HelloWorld.py"]
7
```

Figure 7: Project directory structure and Dockerfile for containerizing the HelloWorld.py Python application.

Dockerfile

The Dockerfile in figure 7 instructs the daemon to build an image with the Python3.10 image as a base, then copy the contents of the working directory (`.`) to a new folder `/app` at the root of the docker container's file system, set the working directory to that folder and finally the `CMD` line sets a command to run whenever the built image is run. This minimal Dockerfile already almost sports all the 5 main Dockerfile commands, the last one being the `RUN` command, which simply interprets the following string as a bash

[4]See `https://docs.docker.com/engine/reference/commandline/build/`

command to execute on the container, for example running `apt-get` commands to install additional software packages

Building the image

The image can be built by opening a terminal, navigating to the project directory and entering `Docker build . -t helloimage`, where the dot indicates the current working directory and `-t` tags the image with the following string[5]. When the build is finished, the image can be seen in the images tab on Docker Desktop or by using the `Docker ... Images` CLI Command.

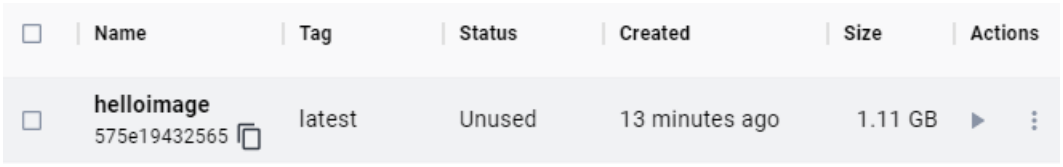


Figure 8: Docker image in Docker Desktop after building the image with the "helloimage" tag.

Running the image

Now a new container can be started up using `Docker run`. Note that each `Docker run` command creates a new container. To avoid containers piling up, the `--rm` flag can be added to remove the container after disconnecting from it. To be able to see the output of HelloWorld.py, the image needs to be run in interactive mode (`-i`) and allocate a pseudo-tty (`-t`). To keep the container from closing immediately and to be able to explore the container, Linux `sh` can be added to the end of the `Docker run` command as a Linux command.

Useful flags

2.4 Docker-compose

Docker-compose is an alternative service to the Docker CLI for issuing instructions to the Docker daemon. The main intended use-case and advantage over the Docker CLI is the ability to start multiple containers - or services, as they are referred to in this context - and specify how they should interact to create multi-container applications or testing networking based applications. In this work, it is simply used to store container startup configurations, as listing out the

```
>Docker run -it --rm helloimage
Hello World, from inside the container!

>Docker run -it --rm helloimage sh
# ls
Dockerfile  HelloWorld.py  requirements.txt
#
```

```yaml
services:
  app:
    image: helloimage
    container_name: devcontainer
    command: sh

    # equiv to -it
    stdin_open: true
    tty: true
```

Figure 9: Running image with `docker run` and `docker-compose` with similar configurations.

[5] Without a tag, the image will actually not show up in Docker Desktop, only in the CLI.

3 Container Development Environment

In this section the Docker concept is adapted to the use as a development environment. Going from application *deployment* as shown in the previous section to application *development* poses some new challenges, which will be adressed in the following chapters where the solutions implemented in the main devcontainer (`main` branch of the GitHub repository) will be shown as well.

For an overview of how the following solutions interact with one another, refer to figure 10.

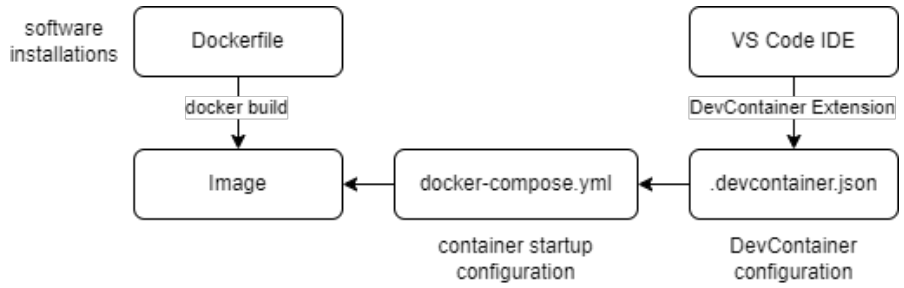


Figure 10: Overview of components of the final devcontainer setup and correspondences to the challenges discussed above. TODO!

Overview solutions

Briefly summarized: In the `Dockerfile`, the toolchain and its dependencies are installed. `Docker-compose` is a Docker feature, which allows for neat storage of container startup configurations in a yaml format. Here, project files are mounted into the container and a solution for display transfer is set up. The docker-compose configuration is in turn used by the VS Code Extension *Dev Containers* through the `.devcontainer.json` file to start a container with VS Code integration so that no IDE has to be installed on the container.

3.1 Setting up development Tools, Dockerfile

Firstly, if the application doesn't only need to run on the container but also be built on it, the toolchain and dependencies required to do so need to be installed.

Tools to be included on the devcontainer image are:

- Python
- C++ tools (gcc compiler, CMake)
- OpenCV for C++ as well as Python

Note: The tool for creating bindings between C++ and Python code will end up not having to be set up here, because a header-only library will be used for this (See section 4).

Dockerfile

Installing these tools is best done at image build time i.e. in the Dockerfile, so that the container's setup is transparent and easily adjustable later on. For the base image, a lightweight Debian-9 based image provided by Python was chosen which comes with Python3.10 preinstalled, taking care of the first point on the list right away.

Installing C++ tools

Setting up the C++ side of the environment is very easy on Linux (line 9 in the Dockerfile 3.1), which is a big advantage of being able to switch to a Linux OS using Docker and WSL. Simply running `apt-get build-essential` will install gcc, g++ and make along with some dependencies.

Building OpenCV from source

Next on the list is OpenCV. While on the Python side OpenCV can be made available with a simple pip install, the C++ library is often recommended to be built from source, even though there are pre-built packages available for download. The OpenCV documentation on installing the library on Linux[6], for example, doesn't even mention that this is an option and goes straight into the process for building from source. Some of the advantages of building OpenCV from source are:

- Ability to customize which modules are built for smaller final image
- Library will be built for specific platform used, leading to better performace
- Ensures correct linker setup, since same linker is used to build the library as the application that's linking it to the application later on

Since all these points are relevant for the devcontainer, we opted to build OpenCV from source as well.

CMake build configuration

OpenCV is built using CMake, which we're already installing as part of the C++ toolchain anyways. To customize the build, OpenCV built various CMake variables into their build configuration, which enable or disable different modules and features. These variables can be set using the `-D` flag in the `cmake` call. After building the library and testing correct functionality, the build time and image size were reduced by disabling the build of examples and extra apps as well as automated tests. Especially the build tests and performance tests add a significant amount of time. All the modules listed under 'main modules' on `https://docs.opencv.org/4.7.0/` are installed with this configuration. For a full list of available build options, use the CMake GUI or list them in the terminal using `cmake -LAH`. Some of the most important build options are also documented in the OpenCV docs [7].

[6] See `https://docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html`
[7] See `https://docs.opencv.org/4.x/db/d05/tutorial_config_reference.html`

```
1  FROM python:3.10
2  WORKDIR /app
3
4  ARG OPENCV_VERSION=4.7.0
5
6  # Install basic utility, C++ tools and OpenCV dependencies
7  RUN apt-get update && apt-get install -y \
8    wget pkg-config vim \                                  # basic utility
9    build-essential cmake --no-install-recommends \        # C++ tools
10   libgtk2.0-dev libncurses5-dev libcanberra-gtk-module   # OpenCV
11                                                          # dependencies
12
13 # Numpy needs to be installed before OpenCV so Python bindings can be ...
       generated
14 RUN pip install numpy
15
16 # Get, build and install OpenCV main modules (may take some time)
17 RUN mkdir -p /opencv && cd /opencv && \
18   wget -O opencv.zip ...
       https://github.com/opencv/opencv/archive/${OPENCV_VERSION}.zip && \
19   unzip opencv.zip && \
20   mkdir -p build && cd build && \
21   cmake \                    # configure build using CMake
22     -D CMAKE_BUILD_TYPE=Release \
23     -D CMAKE_INSTALL_PREFIX=/usr/local \
24     -D BUILD_TESTS=OFF\
25     -D BUILD_PERF_TESTS=OFF \
26     -D BUILD_EXAMPLES=OFF \
27     -D BUILD_opencv_apps=OFF \
28     ../opencv-${OPENCV_VERSION} \
29     | tee cmake.log && \     # create log file from cmake output
30   make -j"$(nproc)" && \     # build on all available CPU cores
31   make install && ldconfig  # install & configure the built library
32
33 # Install additional Python packages
34 COPY ./requirements.txt /app
35 RUN pip install -r ./requirements.txt
```

Figure 11: Dockerfile describing toolchain setup for the main devcontainer image.

Note: Commenting single lines of a multi-line command ('\') will comment the following lines as well and was only done here in the listing for illstrative purposes.

As discussed in chapter 2.2, changes to a line in the dockerfile will invalidate all layer caches associated with the following lines, causing them to be rebuilt instead of loaded from cache. Hence lines, which are least likely to change are placed near the top of the dockerfile and those which are more likely to change are placed far down. If for example instead of installing Numpy on its own prior to installing OpenCV, Numpy was simply included in `requirements.txt` and installed with the other packages on line 14, the `COPY` command would invalidate the OpenCV installation layer and the long process of building OpenCV will be done on every image build, even if nothing at all changed in the dockerfile. And if one was to avoid the `COPY` command by listing the packages directly in the dockerfile, OpenCV would be rebuilt, when that list of packages is changed.

The built image comes in at just 2.38GB uncompressed size and with a build time

of 5 minutes locally (8 CPU cores) and roughly 25 minutes on the GitHub automated test runner. The image is also available to download on docker hub as `ayomeer/cv-devcontainer-image:latest` (812MB download size).

3.1.1 CUDA Support

Separate Container

At around 2GB, the CUDA Toolkit is quite a sizable piece of software, requiring almost as much disk space alone as the entire container built so far. Adding OpenCV's CUDA modules, which were excluded before, adds another 2GB. Also, when working with such advanced tools, there is much better support for a fully featured OS such as Ubuntu, as opposed to the lightweight Debian-9 that was used so far. For these reasons, we opted to create a separate container for developing CUDA applications so that the main container, which likely will see much more use, can remain lightweight. The CUDA devcontainer image is also availabe on Docker Hub under the tag `cv-devcontainer-image:cvcuda`.

Switching base image

Thanks to Docker and the Linux software package concept, this doesn't mean starting from scratch again though. The container's base image can simply be switched out to another by changing the `FROM` line in the Dockerfile. Since in Linux software installation instructions are independent of the distribution, the rest of the Dockerfile remains valid. CUDA will be the most complex piece of the new development environment, so we chose to use leverage the Docker Hub image repository and use a CUDA container image provided by Nvidia themselves as a base for the new devcontainer, to save time on setup. Nvidia provides these images with a variety Linux distributions. Ubuntu 22.04[8] was chosen, as this distribution supports installing Python 3.10 through the standard package managager, keeping the Python version consistent with the previous container.

Installing OpenCV CUDA modules

OpenCV's CUDA modules are added to help with development of our own CUDA code, as well as enabling use of the existing CUDA modules in the OpenCV library. In OpenCV 4.x, the CUDA modules are not part of the main repository, but the opencv_contrib repository. So to build OpenCV with CUDA modules, the contrib files need to be downloaded separately and in the CMake configuration, the location of the contrib files needs to be set using the `OPENCV_EXTRA_MODULES_PATH` build variable. To enable building of the CUDA modules, we set `WITH_CUDA=ON`.

```
cmake \
-D CMAKE_BUILD_TYPE=Release \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D PYTHON_EXECUTABLE=$(which python3) \
-D PYTHON3_PACKAGES_PATH=/usr/lib/python3/dist-packages \
-D OPENCV_EXTRA_MODULES_PATH=../opencv_contrib-${OPENCV_VERSION}/modules\
-D WITH_CUDA=ON \
-D BUILD_TESTS=OFF\
-D BUILD_PERF_TESTS=OFF \
-D BUILD_EXAMPLES=OFF \
-D BUILD_opencv_apps=OFF \
../opencv-${OPENCV_VERSION} | tee cmake.log
```

Figure 12: Full CMake configuration for building OpenCV 4.x from source with CUDA modules from the `cvcuda` image's Dockerfile

Note that the CMake configuration output is saved to `cmake.log` which can be found

[8]The full image tag of the base image used is `nvidia/cuda:12.0.1-devel-ubuntu22.04`

under `/opt/opencv/build` in the built container's file system. Most notably, it shows:

```
--   GUI:                           GTK2
--     GTK+:                        YES (ver 2.24.33)
--       GThread :                  YES (ver 2.72.4)
--       GtkGlExt:                  NO
--     VTK support:                 NO

...

--   NVIDIA CUDA:                   YES (ver 12.0, CUFFT CUBLAS)
--     NVIDIA GPU arch:             50 52 60 61 70 75 80 86 89 90
--     NVIDIA PTX archs:
--
--   cuDNN:                         NO
--
--   OpenCL:                        YES (no extra features)
--     Include path:                /opt/opencv/opencv-4.7.0/3rdparty/include/opencl/1.2
--     Link libraries:              Dynamic load
```

Figure 13: Excerpt from `cmake.log` for OpenCV build configuration.

3.2 Project files and window display, docker-compose

In containerization, the copied files are the final application binaries and as such never change. The binaries can simply be copied in the Dockerfile to make them a permanent part of the container. On a devcontainer however, the project files will change throughout development and you'll want to manage them in a version control system like git.

In principle, git could be installed on the container to pull and push changes directly to and from the container environment. Since the devcontainer is specifically built for the project and visa versa, it makes sense though, to manage the container (i.e. the Dockerfile) under the same repository as the project files. So instead, version control is left to the host and the project directory is *mounted* from the host onto the container's filesystem to make them available to work on within the container. Hereby the container accesses the very same files present on the Windows host's filesystem, through the magic of WSL. This also has the advantage, that additional files can be added to the mounted directory at any time and they will immediately be available on the container. The volume mount is set up in the container's startup configuration (lines 8 and 9 in figure 14).

Mounting project directory

Next, we'll adress displaying GUIs ran on the container on the host system. The system Linux uses to draw GUIs is X11, which conveniently provides a way to export the display output to a network socket, instead of displaying on a screen as usual. To do this the `DISPLAY` environment variable is set to the IP is to accept the X11 connection. So with some changes to the Docker container's startup configuration, any window that opens on the container can sent the host using the X11 protocol. By default, Docker container's networking is isolated from the host. This can again be changed in the container's startup configuration by changing the network mode to `host`. Now the `DISPLAY` environment variable can be set to the host's IP using the DNS name `host.docker.internal` created by Docker.

Displaying windows on host

```yaml
services:
  app:
    image: ayomeer/cv-devcontainer-image:latest
    container_name: devcontainer
    network_mode: host  # Needed, so host.docker.internal points to
                        # actual host IP and not Docker subnet
    volumes:
    - .:/app            # Mount project directory

    environment:
    - DISPLAY=host.docker.internal:0.0 # Set display environment to local host IP
    - MPLBACKEND=TkAgg                 # Set matplotlib backend to one that is..
                                       # ..capable of drawing GUI windows
    # Make container's shell accessible through terminal
    stdin_open: true
    tty: true
```

Figure 14: `docker-compose.yml` file defining container startup configuration described in this section.

On the windows host, an X11 server program is needed to accept the X11 connection. 'VcXsrv' was used throughout development. Set up is described in section 6.1. We also set the matplotlib environment variable `MPLBACKED=TkAgg`, since the default backend `Agg` can only write to files rather than drawing windows.

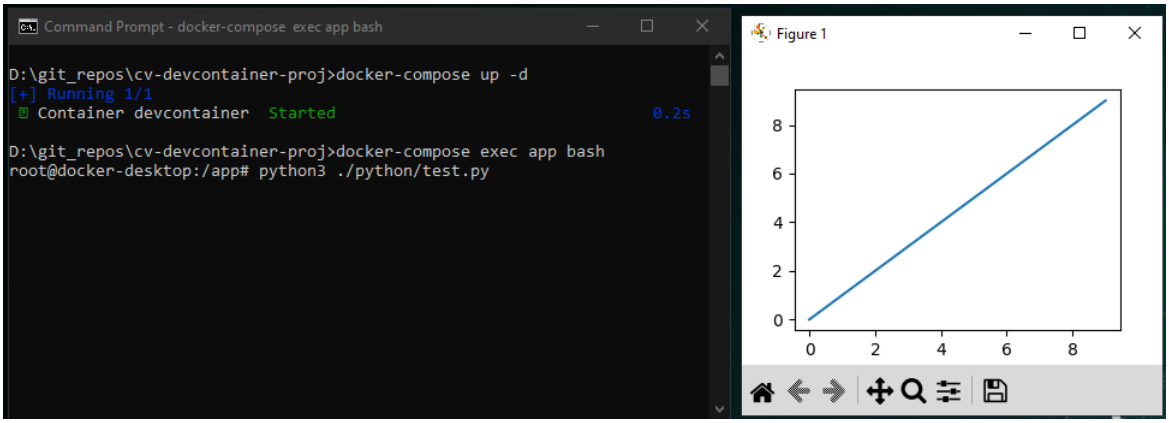


Figure 15: Starting a container using the docker-compose.yaml file from figure 14 and generating a Python plot on the container to test X11 setup and volume mounting.

3.3 Remote IDE, VS Code Dev Containers

Unlike deployment containers which are typically not accessed directly at all after they're spun up, development containers are interacted with constantly. A simple command line interface will no longer do: The use of an IDE within the container needs to enabled. Luckily, the VS Code IDE offers an elegant solution through the *Dev Containers* Extension.

Rather than installing a full IDE on the container and streaming the GUI back to the host, the Dev Containers extension starts up the container with *VS Code Server* added to it, which effectively acts as a remote backend to the IDE. This allows us to run VS Code locally on the host and only send information about file changes, commands given,

etc. back and forth through the network port. This means we get to have the best of both worlds: Interaction with the IDE is just as responsive as using it locally but the entire rest of the development environment is neatly packaged in the container.

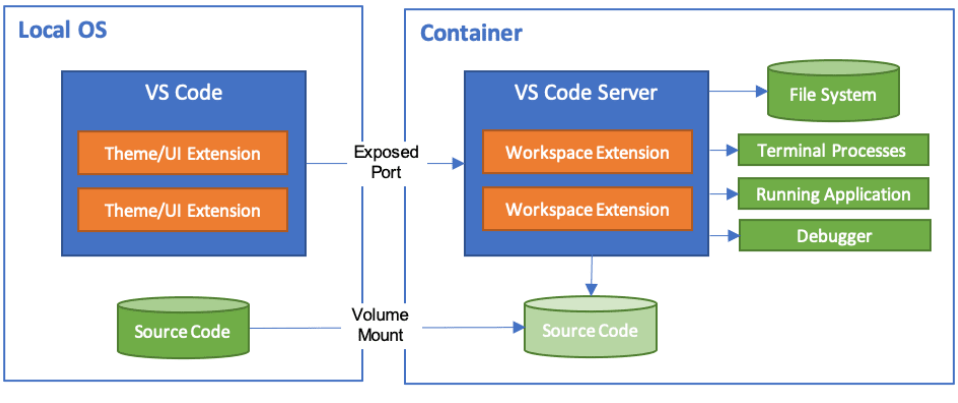


Figure 16: Container access using VS Code Dev Containers extension. Graphic courtesy of Microsoft [7].

.devcontainer.json

To set up a project to use a container environment, a json file with the exact name `.devcontainer.json` has to be added to the root of the project directory. If such a file is present, VS Code will ask to reopen the workspace on the container when opening the project workspace. In this file the container to be used and the startup configuration is set. The remote VS Code environment can also be customized here, by listing other extensions to be installed on the remote instance.

```
{
    "name": "cv-devcontainer",
    "dockerComposeFile": "docker-compose.yml",
    "service": "app",
    "workspaceFolder": "/app",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-python.python",
                "ms-vscode.cpptools",
                "ms-vscode.cmake-tools",
                "076923.python-image-preview"
            ]
        }
    },
    "shutdownAction": "stopCompose"
}
```

Figure 17: `.devcontainer.json` file.

The `.devcontainer.json` file in figure 17 uses the `docker-compose.yml` file in the same directory shown in the previous section in figure 14 to start the service[9] `app` with the same configuration as before. In the `customizations>vscode>extensions` section, the VS Code extensions to be installed on the remote container are listed. To add extensions to the list, simply find the extension as usual in VS Code (locally or on the

remote instance), right click the extension and choose 'add to devcontainer.json'.

For more information on working with dev containers in VS Code, refer to the Microsoft documentation [6, 7].

[9] A service is just a container as configured in a docker-compose file. See chapter 2.4.

4 Python Bindings

In the previous chapter, a development environment was set up that works well for working with Python and C++ seperately. It has not been adressed yet, how using C++ objects and functions from Python is enabled. To create *Python bindings*, libraries (Python or C++) are used.

General considerations

The core problem binding libraries aim to solve is translating data types between the two languages. For this project, the main objects that will be passed between Python and C++ are images with the datatypes of Numpy arrays on the Python side and OpenCV matrix objects on the C++ side. So when choosing a binding library, this specific interaction should be solved well and easy to implement. Passing the arrays should also be able to be done without copying the arrays each time they are passed, since images are large arrays.

4.1 Choosing a binding Method

There are many approaches one can take to achive this. Some of the most notable are:

Overview of binding Options

- Boost.Python
- Pybind11
- cppyy
- Cython

Boost.Python

Boost.Python is part of the large family of Boost libraries created to extend C++ with a wide variety of functionalities. Boost.Python is widely used and considered somewhat of a gold standard for its simple interface and support for almost any C-standard and compiler. This wide support of now mostly obsolote C-compilers however, has increased the complexity of the library over the years. As we don't need to work with legacy code written for an obscure C-compiler, the next option is more interesting.

Pybind11

Pybind11 is a header-only library on Boost.Python, which positions itself as 'Boost.Python light', doing away with all the legacy support and focusing on the currently most relevant C++11 standard. This allowed them to build the library around modern C++ features and paradigms, simplify usage further. Pybind11 is well documented for a young library and also sports specific support for Numpy arrays through the use of the Python C-API's buffer protocol.

cppyy

The cppyy Python library is an automated approach to creating Python bindings for C++ objects, which can be defined directly on the Python file. This doesn't fit the project all that well. Just small amount of C++ objects need to be bound but bound well for good performance. On the C++ side we need to non-standard OpenCV datatypes, so the control of writing manual bindings might be needed. Integrating C++ code directly into Python isn't quite the goal either. The prototyping use-case we are trying to support with the development environment, is to write a C++ module that can be used in the final, fully C++ based application, as well.

Cython

Cython is even more Python oriented, defining a custom Python-like language to *generate* C++ implementations. This strays far from this project's use-case.

Choice

Pybind11 was chosen to go forward with, as it fits the project's needs the best. The header-only implementation makes it lightweight and quick to integrate. The binding

interface is by far the simplest while also offering the control needed to implement a performant solution for non-standard datatypes. Microsoft also recommends Pybind11 over the other options for use with C++ specifically[5].

The header-only nature of Pybind11 means, that the Pybind11 header files just need to be included in as a subdirectory in the project files. No installation is needed. Hence, with this solution, binding functionality isn't built into the container image itself but simply part of the project files. These files are however part of the devcontainer repository template as well as a CMake configuration that links it to the project.

4.2 Creating bindings with Pybind11

Basic structure

Exposing C++ functions using Pybind11 could hardly be simpler, as demonstrated by the example function found in the `main` branch project template:

```
1  void sayHello(){
2      printf("Hello from C++! \n");
3  }
4
5  PYBIND11_MODULE(cppmodule, m){
6      m.def("sayHello", &sayHello);
7  }
```

Figure 18: Exposing a function with no arguments or return type.

The binding code can written be directly in the `.cpp` file where the functions and classes to bind are defined using the macro `PYBIND11_MODULE(module_name, handle)`. The `handle.def()` method tags the C++ function to be wrapped under the name given by the string in the first argument. Adding arguments and returns of standard datatypes like int or float actually doesn't need any additional binding code over what's shown in figure 18, as Pybind can handle those automatically. If datatypes are involved that are either exclusive to Python or C++, like Numpy arrays for Python and custom datatypes like OpenCVs `cv::Mat` one has to start putting some thought into it.

Datatype conversions

The documentation lists three ways to handle type conversions using Pybind11 out of the box:

1. Using the datatype native to C++ and wrapping it for use in Python
2. Using the datatype native to Python and wrapping it for the use in C++
3. Converting the datatype when crossing between C++ and Python (must pass by value)

Working with wrappers on either side is undesirable, but as one of the objects to pass between Python and C++ will inevitably end up being images in the form of large arrays, passing by value is rather unattractive as well. One thing we have working for us, is that Numpy arrays are really the only Python type we care about sending back and forth efficiently, which also happens to have another very general way of accessing its data in the *buffer protocol*. The solution implemented in the example application could be summarized as

4. Use 2. for passing arguments (Python $\rightarrow$ C++), 1. for returning Python $\leftarrow$ C++ and use buffer protocol to convert "by reference"

This both sides can work with, and send their native types to each other without having to copy data.

Buffer protocol

The buffer protocol is part of Python's C-API, which can be used to access the underlying memory of Python objects such as Numpy arrays directly. It specifies all the parameters needed to correctly interpret the underlying data as a matrix object, such as data pointer, shape, element datatype and stride, regardless of what class implements it or even which language that class is written in.

Numpy array to cv::Mat

To be able to accept Numpy arrays as arguments in C++ functions, Pybind11 provides the datatype `py::array_t<T>`, which generates the appropriate wrapper in C++ to be able to access the array's buffer. To use the array as a `cv::Mat` object in C++, we create one such object with and copy the structure of the incoming array as well as the data pointer. For the duration of the function call, there are two matrix objects of different classes linked to the same underlying data. After the function completes, the `cv::Mat` object leaves scope and gets deleted.

```
1  void acceptImage(py::array_t<std::uint8_t>& pyImg){
2      // Link pyImg data to cv::Mat object img
3      cv::Mat img(
4          pyImg_d.shape(0),           // rows
5          pyImg_d.shape(1),           // cols
6          CV_8UC3,                    // data type
7          (uint8_t*)pyImg.data());    // data pointer
8
9      ... // use img in C++ function
10
11     return;
12 }
13
14 PYBIND11_MODULE(cppmodule, m){
15     m.def("acceptImage", &acceptImage);
16 }
```

Figure 19: Using Pybind11's `py::array` datatype to accept Numpy arrays in C++.

A nice way to separate the C++ code from the binding code is using C++11's lambda functions to create another wrapper directly in the Pybind11 module definition. With this separation, the C++ function can be called from other C++ functions as well. This comes in handy if during development, it's beneficial to build a pure C++ executable instead of a shared library to be used in conjunction with Python, for example to test the C++ functions in isolation. Even for C++ functions, which are only ever meant to be called by other C++ functions, it can be neat to have this kind of Python interface to quickly test the logic using a Python skript.

```cpp
void acceptImage(cv::Mat& img){
    ... // use img in C++ function
    return;
}

PYBIND11_MODULE(cppmodule, m){
    m.def("acceptImage", [](py::array_t<uint8_t>& pyImg)
    {
        // Link pyImg data to cv::Mat object img
        Mat img(
            pyImg_d.shape(0),           // rows
            pyImg_d.shape(1),           // cols
            CV_8UC3,                    // data type
            (uint8_t*)pyImg.data());    // data pointer

        // -- Call C++ function with C-types
        return acceptImage(img);
    });
}
```

Figure 20: Alternate binding implementation, separating C++ function from Python binding code.

cv::Mat to Numpy array

To be able to return the result of the computations performed on `cv::Mat` objects in the C++ code, we go the other way and wrap the `cv::Mat` class, exposing it as a buffer object, which can be cast to a Numpy array by Python.

```cpp
cv::Mat returnImage(){
    ... // Create cv::Mat image to return
    return cvMat;
}
PYBIND11_MODULE(cppmodule, m){
    py::class_<cv::Mat>(m, "Mat", py::buffer_protocol())
        .def_buffer([](cv::Mat &im) -> py::buffer_info {
            return py::buffer_info(
                im.data,                                        // pointer to data
                sizeof(unsigned char),                          // item size
                py::format_descriptor<unsigned char>::format(), // item descriptor
                3,                                              // matrix dimensionality
                {                                               // buffer dimensions
                    im.rows,
                    im.cols,
                    im.channels()
                },
                {                                                // strides in bytes
                    sizeof(unsigned char) * im.channels() * im.cols,
                    sizeof(unsigned char) * im.channels(),
                    sizeof(unsigned char)
                }
            );
        })
    ;
}
```

Figure 21: Binding code for returning `cv::Mat` objects as buffer objects.

Here, Pybind11's inbuilt Numpy support really comes into full effect. `py::class_<T>()`

exposes the class, similar to how `.def()` did for functions in figure 18. Now an additional third argument can optionally be passed though, indicating a class to inherit from. Here we use the `py::buffer_protocol()` tag to give the wrapper class buffer properties. Now a buffer can be defined using `.def_buffer()`, the input argument of which is again constructed using a lambda function, where a `buffer_info` object is created from the `cv::Mat` objects properties. Now, whenever a `cv::Mat` object is returned to Python anywhere in the code, this wrapper will be applied to return a buffer object instead.

5 CUDA

The Nvidia CUDA framework allows for large scale parallelization by using the many processors available on the GPU. While a comprehensive introduction to the programming model is out of the scope of this work, a couple of relevant concepts and implementation details used in later examples shall be illustrated here. Especially, how OpenCV's CUDA core module can massively simplify the memory management for image processing applications shall be shown. For further reference, please refer to the example implementations in chapter 6.3.4 and the excellent CUDA programming guide by Nvidia [9].

GPU acceleration considerations

Not all algorithms benefit from GPU parallelization. In contrast to CPU parallelization, GPU parallelization comes with a potentially significant computational overhead as the data to work on first has to be copied onto the separate GPU *device memory* and back to *host memory*, once the computation has concluded. For GPU acceleration to be beneficial, the performance gained from computing on the GPU must overcompensate for the added overhead. If the data set is of moderate size and the computational complexity low, the CUDA implementation likely is actually slower that a simple CPU based one. However as datasets get bigger and the sheer computational bandwidth of a GPU gets fully utelized, a CUDA implementation is likely to outperform a CPU based one, even for relatively low complexity algorithms.

5.1 Thread Blocks and Grids

As a consequence of GPU architecture, thread ressources are segmented into *blocks* of 1024 threads each. For larger structures, these blocks can be arranged in a *grid*. To match the structure of the data that's being worked on, both blocks and grid can be arranged in up to three dimensions. For processing images for example, it makes sense to arrange blocks into two dimensions, such that the thread indexes represent the subregion the thread block works on. Similarly, the grid is also arranged as a 2D array of blocks, to arrange the subregions into the same shape of the processed image. The arrangement in figure 22, for example could represent a 64x128 image, processed by a grid of 2x4 block of 32x32 threads.[10]

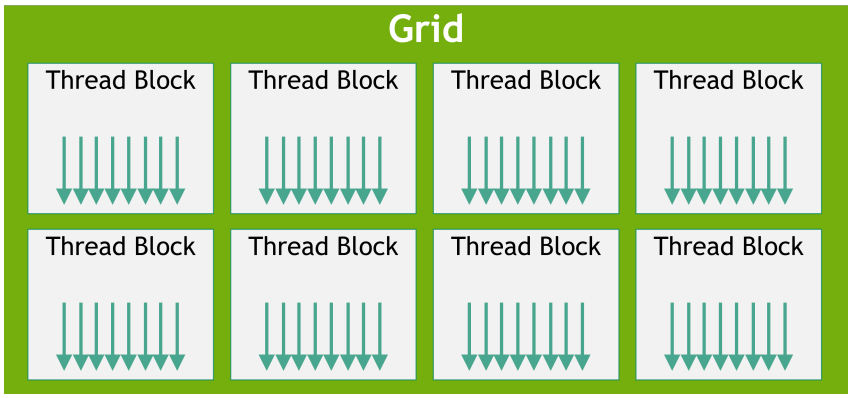


Figure 22: Thread hirarchy of blocks and grid. Source: CUDA Programming Guide[9]

Each thread can access its index within the block as well as the block's index within

[10] Usually, not all 1024 threads are explicitly used to allow the others to be used for memory management. A common choice is a block size of 16x16

the grid. This information is provided directly by the CUDA framework through the built-in variables `threadIdx` and `blockIdx`. These variables mirror the dimensionality of the block or grid they're indexing. Together with a third built-in variable `blockDim`, the position of the thread's pixel can be reconstructed as follows:

```
const int i = blockIdx.x * blockDim.x + threadIdx.x;
const int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Figure 23: Using Pybind11's `py::array` datatype to accept Numpy arrays in C++.

5.2 Writing CUDA Kernels for Image Processing

The data structures of images are quite challenging to work with when writing low-level CUDA code. A naïve implementation of even simple function like transposing an image can lead to abysmal use of the potential of the GPU, using less than 5% of the available bandwidth [4]. Thankfully, OpenCV's CUDA core module offers a framework purpose built for handling the access of image data on the GPU efficiently.

The main enabler is the `cv::cuda::GpuMat` class, which is a matrix class designed for managing image data in device memory. Once instantiated, these objects automatically allocate space in device memory and provide methods for uploading and downloading data to and from device memory. More importantly though, the class defines an interface for efficient access of the elements, once passed to the kernel. This is important because kernels, being low-level constructs, do not allow usage of classes, unless they are built specifically for this purpose. Figure 24 illustrates the usage of this interface by the simple example of transposing an image. In the host code, the image can be passed to the kernel as a `GpuMat` object, while the kernel function receives it as a `PtrStepSz` object. The mechanism behind this automatic conversion is an overloaded typecast operator declared as a member of `GpuMat`. `PtrStepSz` only exposes element access through the `.ptr()[]` method and basic information about the structure of the matrix such as rows and columns.

cv::cuda::GpuMat

```
// Device code
__global__ void transposeKernel
(
const cv::cuda::PtrStepSz<uchar3> src,
cv::cuda::PtrStepSz<uchar3> dst,
)
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    const int j = blockIdx.y * blockDim.y + threadIdx.y;

    dst.ptr(i)[j] = src.ptr(j)[i];
}

// Host code
int main(){
    ...
    // Kernel launch
    transposeKernel<<<gridSize, blockSize>>>(src, dst);
    ...
}
```

Figure 24: Working with cv::cuda::GpuMat in the cuda kernel.

6 Developing on the Containers

This section should be the entry point for anyone wanting to make use of the devcontainers that were build up throughout the previous chapters. The requirements and setup to make use of the devcontainers will be shown. Then, the process of developing a computer vision application on the devcontainers will be illustrated by example of a sample application for each container, making use of the complete toolchain on each one.

6.1 Windows Host Setup

It was the goal to keep dependencies on the host at a minimum. While a handful of programs are required to make use of the devcontainers, the installation process is very straightforward as none of them require any further configuration.

These are the setup steps:

1. Upgrade WSL to WSL2
   - Start a terminal and run `wsl --install`.
   - Check WSL version: Run `wsl -l -v`.
2. Install Docker Desktop
   - Download from `https://www.docker.com/products/docker-desktop/`
3. Install VS Code
   - Download from `https://code.visualstudio.com/download`
   - Install 'Dev Containers' Extension
4. Install 'VcXsrv' X11 Server Software
   - Download from `https://sourceforge.net/projects/vcxsrv/`

Note: Since it was not directly relevant to the goal of this project, setup on Linux based hosts was not tested. While Docker containers are host OS independent, the setup for Docker itself and container startup configuration for exporting the display in the docker-compose.yml file may differ.

To start working with the devcontainers, start Docker Desktop and VcXsrv's XLaunch. Nothing has to be done in Docker Desktop, it just needs to run so the Docker daemon is active. It can be useful to monitor the status of the container, once it runs though.

When starting VcXsrv, all but the last dialogue 'Extra settings' can be clicked through unchanged. Here the checkbox 'Disable access control' needs to be set, so the container is allowed to connect.

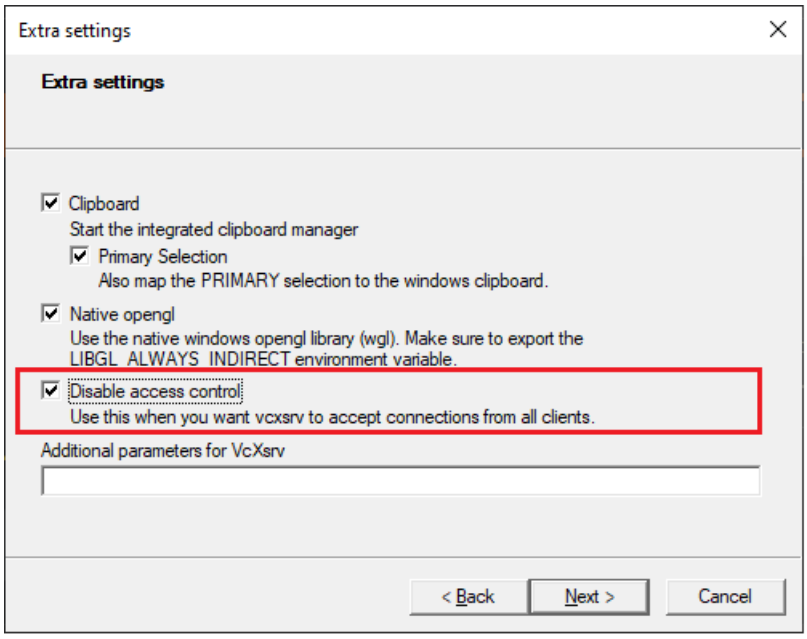


Figure 25: VcXsrv startup configuration.

Beyond this chapter, there won't be any more mention of the development container, as once the setup is complete, it really becomes a "local-quality environment" and one has to pay no mind to the fact that the back end actually runs on a container.

6.2 Starting a new Project

The main branch of the GitHub repository has been set up to serve as a starting template for new projects. To start a new project, simply clone or download the main branch files. The folders `_img`, `cpp` and `python` are where your new project files go. In `cpp`, a template project is set up including Pybind11 files and a `CMakelists.txt` to link them to the project using CMake. The `rebuild.sh` shell script is provided to rebuild the C++ module after changes have been made to `cppmodule.cpp`.

6.3 Example Application: Homography Undistortion

In this example, usage of the development environment is illustrated for the intended use case of advanced computer vision prototyping. First, a Python script is written to implement and test the algorithm logic. With the motivation of moving towards a real-time capable implementation, we then begin optimizing performance by first moving the section most critical to the algorithms performance over to a Python callable C++ module. A further optimization iteration makes use of the pthreads standard library to split the workload across multiple CPU cores. Finally, we take parallelization to the next level by making use of the CUDA toolkit to write a custom CUDA kernel and GPU accelerating the algorithm.

6.3.1 Homography

Homography Concept

Abstractly speaking, Homography is a transformation, which maps points from one plane to another. More concretely, Homography is a geometric coordinate transform which can be understood as an extension of the more commonly known 2D affine transform. By generalizing affine space to *projective space* two new degrees of freedom describing *perspective* are added. While for affine transforms, parallel lines will always remain parallel, projective transforms can transform parallel lines to lines which meet. A unit-square for example, can be transformed into something, which looks more like a square object lying in a 3D scene.

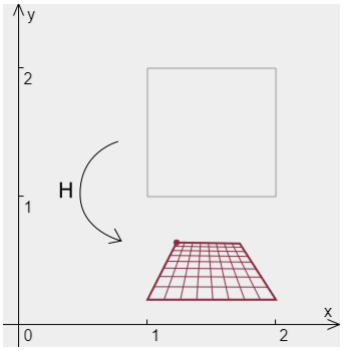


Figure 26: Projective Transform

Application

Inversely, we can also *undistort* the perspectively distorted plane to get back to the unit-square. This is the idea behind this first example application. A planar surface is chosen in the perspective input image by manually picking four points $\tilde{x}_{d,i}$. In the undistorted output image, we expect the corresponding points to be arranged in a rectangle. These points become the corners of the undistorted output image $\tilde{x}_{u,i}$.

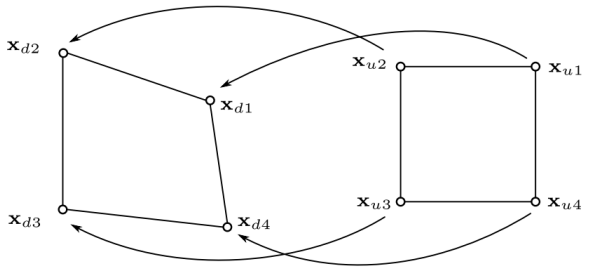


Figure 27: Point correspondences between perspectively distorted input image and the undistorted output image.

Point correspondences

So after choosing the dimensions of the output image, we have four point correspondences between the input image and the undistorted output image, which is exactly what is needed to determine the 8 degrees of freedom of a homography transform using a system of equations. The system of equations can be derived from the homography transform statement for each point $i \in 0, 1, 2, 3$:

$$\tilde{\mathbf{x}}_{d,i} \cdot \lambda = \mathbf{H}_u^d \cdot \tilde{\mathbf{x}}_{u,i} \qquad \Rightarrow \qquad \begin{bmatrix} x_{d,i} \\ y_{d,i} \\ 1 \end{bmatrix} \cdot \lambda = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{u,i} \\ y_{u,i} \\ 1 \end{bmatrix} \tag{1}$$

which encapsulates the system of equations:

$$\begin{cases} x_{d,i} \cdot \lambda = a \cdot x_{u,i} + b \cdot y_{u,i} + c \\ y_{d,i} \cdot \lambda = d \cdot x_{u,i} + e \cdot y_{u,i} + f \\ \lambda = g \cdot x_{u,i} + h \cdot y_{u,i} + 1 \end{cases} \tag{2}$$

By substituting the last equation into the previous two, the number of equations is reduced to two equation for each point:

$$\begin{cases} x_{d,i} \cdot (g \cdot x_{u,i} + h \cdot y_{u,i} + 1) = a \cdot x_{u,i} + b \cdot y_{u,i} + c \\ y_{d,i} \cdot (g \cdot x_{u,i} + h \cdot y_{u,i} + 1) = d \cdot x_{u,i} + e \cdot y_{u,i} + f \end{cases} \tag{3}$$

Gathering the homography matrix coefficients into a vector, the system of equations can once again be expressed in matrix form as follows:

$$\begin{bmatrix} x_{u,i} & y_{u,i} & 1 & 0 & 0 & 0 & (-x_{d,i} \cdot x_{u,i}) & (-x_{d,i} \cdot y_{u,i}) \\ 0 & 0 & 0 & x_{u,i} & y_{u,i} & 1 & (-y_{d,i} \cdot x_{u,i}) & (-y_{d,i} \cdot y_{u,i}) \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_{d,i} \\ y_{d,i} \end{bmatrix} \tag{4}$$

Bringing all four equations into the same system and thus resolving the index i, we get the full matrix equation in the form $\mathbf{Ax} = \mathbf{y}$, where A is a square (8×8) Matrix that can be inverted to solve for the coefficient vector.

$$\begin{bmatrix} x_{u,0} & y_{u,0} & 1 & 0 & 0 & 0 & (-x_{d,0} \cdot x_{u,0}) & (-x_{d,0} \cdot y_{u,0}) \\ 0 & 0 & 0 & x_{u,0} & y_{u,0} & 1 & (-y_{d,0} \cdot x_{u,0}) & (-y_{d,0} \cdot y_{u,0}) \\ x_{u,1} & y_{u,1} & 1 & 0 & 0 & 0 & (-x_{d,1} \cdot x_{u,1}) & (-x_{d,1} \cdot y_{u,1}) \\ 0 & 0 & 0 & x_{u,1} & y_{u,1} & 1 & (-y_{d,1} \cdot x_{u,1}) & (-y_{d,1} \cdot y_{u,1}) \\ x_{u,2} & y_{u,2} & 1 & 0 & 0 & 0 & (-x_{d,2} \cdot x_{u,2}) & (-x_{d,2} \cdot y_{u,2}) \\ 0 & 0 & 0 & x_{u,2} & y_{u,2} & 1 & (-y_{d,2} \cdot x_{u,2}) & (-y_{d,2} \cdot y_{u,2}) \\ x_{u,3} & y_{u,3} & 1 & 0 & 0 & 0 & (-x_{d,3} \cdot x_{u,3}) & (-x_{d,3} \cdot y_{u,3}) \\ 0 & 0 & 0 & x_{u,3} & y_{u,3} & 1 & (-y_{d,3} \cdot x_{u,3}) & (-y_{d,3} \cdot y_{u,3}) \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_{d,0} \\ y_{d,0} \\ x_{d,1} \\ y_{d,1} \\ x_{d,2} \\ y_{d,2} \\ x_{d,3} \\ y_{d,3} \end{bmatrix} \tag{5}$$

Undistortion

Once the homography which transforms the four points in the input image to the four points in the output image is found, the output image can be generated by iterating through each of its pixels, applying the homography transform to the coordinate vector according to equation 1 to get the coordinates of the corresponding pixel in the input image and writing the RGB value found at that location in the input image into the current output image pixel. Implementing this algorithm in Python is very straight forward. One such implementation is shown in figure 28.

```python
def hom2inhom(xhom):
    return xhom[0:2]/xhom[2]

def inhom2hom(x):
    xhom = np.ones(3)
    xhom[0:2] = x
    return xhom

def pointwiseUndistort(H_d_u, img_d, M, N):
    img_u = np.empty((M,N,3), np.uint8)
    for m in range(M):
        for n in range(N):
            # Change to hom. coords, do transform, go back to inhom coords
            xu = np.array([m, n])
            xu_hom = inhom2hom(xu)

            xd_hom = H_d_u @ xu_hom # hom. transform

            xd = hom2inhom(xd_hom)
            xd = np.round(xd).astype(int) # get integer coords

            # Use transformed coords to get pixel value
            img_u[m][n] = img_d[xd[0], xd[1], :] # last dimensions: rgb channels
    return img_u
```

Figure 28: Python implementation of the undistortion transform algorithm.

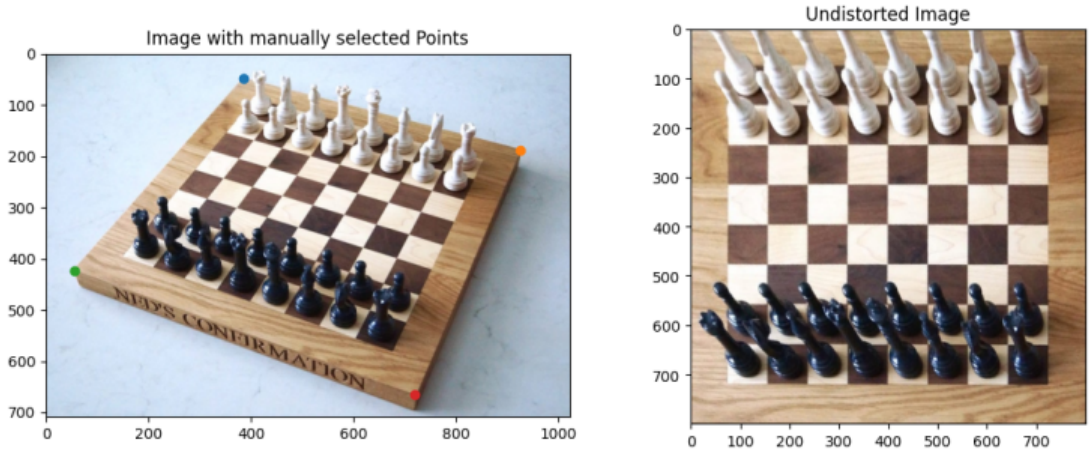


Figure 29: Image containing planar surface to undistort and result of the homography transform.

Critical section

For an image with the modest dimensions of 1000x1000 pixels, this sequence will already need to be executed a million times. It is safe to say, that this pointwise undistortion is the critical section, when it comes to performance. Matrix multiplication in Python using Numpy is fast, as Numpy uses a highly optimized implementation in C under the hood. Loops however, are notoriously slow in Python. In the following sections, different methods for improving performance using C++ are explored.

This Python-only prototype sets the performance baseline at **6.27 seconds**.

6.3.2 C++ Implementation, Pybind11

Reimplemented in C++, the `pointwiseUndistort` function looks much the same, other than that we now use the OpenCV class `cv::Mat` as our matrix class instead of Numpy arrays. Of course, this being C++, now data types have to be payed close attention to as well.

```
Mat pointwiseUndistort(cv::Mat& img_d, cv::Mat& H, int M, int N){
    Mat img_u(M, N, CV_8UC3); // prepare return image

    for (int m=0; m<M; m++){ // m = {0, 1, 2, ... , M-1}
        for (int n=0; n<N; n++){
             // Create hom. vector from coords (m, n)
            Mat xu = (Mat_<int>(3, 1) << m, n, 1);
            xu.convertTo(xu, CV_64F); // convert xu to double for matprod

            Mat xd = H*xu; // hom. transform

            xd = xd / xd.at<double>(2,0); // convert back to inhom coords
            xd.convertTo(xd, CV_32S); // get integer coords

            // Use transformed coords to get pixel value from distorted image
            int x = xd.at<int>(0,0);
            int y = xd.at<int>(1,0);

            img_u.at<Vec3b>(m, n) = img_d.at<Vec3b>(Point(y, x));
        }
    }
    return img_u;
}
```

Figure 30: Python implementation of the undistortion transform algorithm.

How a function such as this is exposed to Python is described in chapter 4.2. The examples shown there are based directly on the binding code of this very function, so the code will not be repeated here. The binding code will also stay the same for the other implementation variations shown in the following chapters.

This relatively painless procedure of porting the critical section to C++ yields roughly a ten-fold increase in performance with a new execution time averaging around **0.6 seconds**.

6.3.3 C++/pthreads Implementation

Now that the transition to C++ has been accomplished in way that allows us to use the same datatypes, as if we were building a pure C++ application, the full toolset available in C++ can be leveraged. One rather low hanging fruit is making use of more than one CPU core, i.e. *multithreading*. This is well supported in C++, for example through the standard library 'pthread'. Instead of one process looping through the entire index range of the output image, the image is sliced up into equal strips to be processed by different threads concurrently. To do this, the algorithm is encapsulated into a *thread function*, which is used by all threads but with different arguments.

```
1  ... // Other includes
2  #include <thread>
3  using namespace std;
4
5  void undistortTh(Mat& img_u, Mat& img_d, Mat& H,
6                   size_t xStart, size_t xEnd){
7      double N = img_u.rows;
8      for (double m=xStart; m<xEnd; m++){ // double for compatibility for
9          for (double n=0; n<N; n++){         // for matmul later in body
10             ... // same body as before (line 9 through 19 previous listing)
11         }
12     }
13 }
14 Mat pointwiseUndistort(Mat& img_d, Mat& H, int M, int N ){
15     Mat img_u(M, N, CV_8UC3); // prepare return image
16
17     size_t range = M/4;
18     thread th1(undistortTh, ref(img_u), ref(img_d), ref(H), 0*range, 1*range);
19     thread th2(undistortTh, ref(img_u), ref(img_d), ref(H), 1*range, 2*range);
20     thread th3(undistortTh, ref(img_u), ref(img_d), ref(H), 2*range, 3*range);
21     thread th4(undistortTh, ref(img_u), ref(img_d), ref(H), 3*range, M);
22
23     th1.join(); th2.join(); th3.join(); th4.join(); // Forgive me for I have
24                                                     // sinned (for brevity)
25     return img_u;
26 }
```

Figure 31: Using the pthread standard library to implement a multithreaded solution.

Note: `std::thread` can't accept references, as it must be able to copy the arguments into the function it calls. To avoid copying the image data, the wrapper `std::ref()` can be used.

Again, with little extra effort over the previous implementation a manifold improvement in performance can be gained. This implementation reduces the algorithms execution time to **0.15 seconds** on four cores.

Of course this is a rather naive implementation, spelling out each thread explicitly. A more elegant one would be to use a thread pool approach. Here, we won't dwell on CPU multithreading very long though, as in the next section, we make another leap to a fundamentally different architecture in CUDA and taking parallelization to the extreme by involving a GPU.

6.3.4 C++/CUDA Implementation

Switching Containers

To start using CUDA, without having to clone another repository and copy our files over, we can simply switch out the `docker-compose.yml` defining the container environment for VS Code to attach to to the one included in the `cv-devcontainer-proj/cuda` branch. If this is the first time using the CUDA container, the image will automatically be downloaded when starting up VS Code in the project directory and choosing "rebuild and reopen in container".

CUDA Project setup, CMake

To convert the C++ project into a CUDA-Project we need to change the CMake configuration slightly. CUDA is not a library, but an extension to the C++ language, so what we need to do is add the CUDA language to the project and change a couple of other setting so the `CMakeLists.txt` reads as follows:

```
cmake_minimum_required(VERSION 2.8)
project(cppmodule LANGUAGES CXX CUDA) # [Changed] add CUDA language

# Set C++11 as minimum required standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CSS_STANDARD_REQUIRED ON)

add_subdirectory(pybind11)
pybind11_add_module(cppmodule cppmodule.cu) # [Changed] .cpp -> .cu

# [NEW] Set compute capability to 6.1 for Pascal architecture (Quadro)
set_target_properties(cppmodule PROPERTIES CUDA_ARCHITECTURES "61")

# Find and link OpenCV library
find_package(OpenCV REQUIRED)
target_link_libraries(cppmodule PRIVATE ${OpenCV_LIBS})
```

Figure 32: CMakeLists.txt for CUDA project

Now we can change our `cppmodule.cpp` to a CUDA file `cppmodule.cu` and we can start making use of the CUDA Toolkit.

Reimplemenation in CUDA

Reimplementing the undistortion function once more for CUDA is not entirely dissimilar to the previous implementation using multithreading. In the same way as in multithreading, a function is defined, which is used by all threads. In the context of CUDA, the counterpart of the threading function is the *kernel*. As discussed in chapter 5.2, CUDA kernels are quite restrictive when it comes to high level structures, which is why the matrix multiplication has been realized using primitive operations only.

```cpp
// Device code
__global__ void undistortKernel
(
    const cv::cuda::PtrStepSz<uchar3> src,
    cv::cuda::PtrStepSz<uchar3> dst,
    double* H
)
{
    // Get dst pixel indexes for this thread from CUDA framework
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    const int j = blockIdx.y * blockDim.y + threadIdx.y;

    // H*xu_hom
    double xd_hom_0 = H[0]*i + H[1]*j + H[2];
    double xd_hom_1 = H[3]*i + H[4]*j + H[5];
    double xd_hom_2 = H[6]*i + H[7]*j + H[8];

    // Convert to inhom and round to int for use as indexes
    int xd_0 = (int)(xd_hom_0 / xd_hom_2); // x
    int xd_1 = (int)(xd_hom_1 / xd_hom_2); // y

    // Get rgb value from src image
    dst.ptr(i)[j] = src.ptr(xd_0)[xd_1];
    return;
}
```

Figure 33: CUDA Kernel implementing undistortion.

Host code

In the host code, the fundamental difference to CPU multithreading becomes apparent. The host memory available to the main program and the CPU is completely separate from GPU memory. Any data to be used in the kernel needs to be allocated and copied to device memory. For images, OpenCV's `cv::cuda::GpuMat` is used to manage its memory. The homography matrix on the other hand, shows how regular datatypes are managed using CUDA functions `cudaMalloc()` and `cudaMemcpy`.

The `cudev::divUp()` function seen on line 20 and 21 is a very simple helper function that essentially just divides the first argument by the second and rounds the result up to determine the number of blocks needed in each dimension.

On line 24, the special CUDA syntax for launching a GPU a set of `gridSize·blockSize` kernels using tripple angle brackets. `cudaDeviceSynchronize()` ensures all threads are done before we retrieve the image from device memory.

```cpp
// Host code
void pointwiseUndistort(Mat& img_d, double* H, int M, int N){

    Mat img_u(M, N, CV_8UC3); // prepare return image
    cv::cuda::GpuMat img_u_gpu(img_u); // create GpuMat from regular Mat

    // Allocate space on device for H-array
    double* dPtr_H = 0;
    cudaMalloc(&dPtr_H, (3*3)*sizeof(double));
    cudaMemcpy(dPtr_H, H, (3*3)*sizeof(double), cudaMemcpyHostToDevice);

    // prep input image and return image
    cv::cuda::GpuMat src;
    src.upload(img_d);

    Mat ret;
    cv::cuda::GpuMat dst(M, N, CV_8UC3); // allocate space

    // Kernel launch
    const dim3 blockSize(16,16); // rule of thumb
    const dim3 gridSize(cv::cudev::divUp(dst.cols, blockSize.x),
                        cv::cudev::divUp(dst.rows, blockSize.y));

    undistortKernel<<<gridSize, blockSize>>>(src, dst, dPtr_H);
    cudaDeviceSynchronize();

    // Get and show result
    dst.download(ret);
    imshow("gpu image", ret);
    waitKey(0);

    return;
}
```

Figure 34: CUDA host code launching a kernel for each pixel in the output image.

Note: The first time a set of CUDA kernels is launched is always significantly slower that consequent launches due to the GPU being initialized on the first run. To assess performance, always do multiple runs and discard the first one.

Performance

Leveraging large scale parallelization and assigning a thread to each pixel in the output image like this cuts the transformation's execution time by magnitudes once more. This implementation completes in around **0.0009 seconds or 900us**. At around 300us, uploading and downloading the data to and from device memory actually makes up two thirds of the execution time, however it's still absolutely worth it in this case.

6.4 Example Application: Real-time Homography Reconstruction

Having proven with the experiment in the previous chapter, that using CUDA brings homography transformations well within the realms of real-time viablity, a more ambitious project was started to really demonstrate the performance we're able to achieve while still being in a prototyping environment on a pseudo-remote development platform.

Application Pipeline

The manual selection of four points and determining the homography mapping is to be replaced by automatic feature detection using the SIFT algorithm and RANSAC to fit a homography between the perspective input image of a box and a template image of the box's top face. Then, a new element in pose estimation is introduced and used to draw a wireframe around the boxes edges. Once found, the pose can be altered by interacting with the wireframe plot. The altered wireframe then serves to provide new correspondence points to find the homography to, which can in turn be used to transform the box faces from the perspective image correctly unto the altered wireframe, effectively rendering a 3D reconstruction of the box.

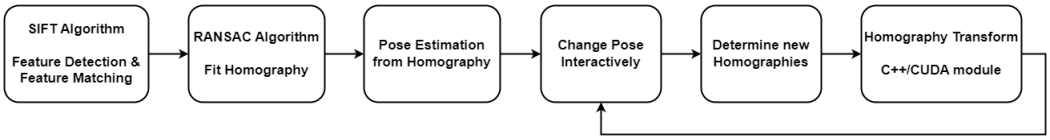


Figure 35: Pipeline for the real-time homography reconstruction application.

Real-time setup

While there is a lot of interesting maths going on, on the Python side of things, time has run too short at this point to go into any significant amount of detail. The most relavant thing to prototyping with CUDA and the development environment itself than I can outline quickly is how the CUDA module from the last chapter was adapted to real-time usage. The distinction is significant because memory is now to be allocated once, and reused for each kernel launch. A class implementation to manage data allocation and pointers makes sense.

```
1  class HomographyReconstruction {
2      public:
3      HomographyReconstruction(py::array_t<imgScalar>& py_queryImage); // <-,
4      ~HomographyReconstruction(); // frees GPU memory  |  allocates GPU memory
5
6      // kernel launch function
7      cv::Mat pointwiseTransform(
8          py::array_t<matScalar>& pyH,
9          const py::array_t<int>& py_polyPts,
10         const py::array_t<int>& py_polyNrm);
11
12     private:
13     cv::Mat queryImage;
14     // device memory pointers
15     double* d_ptr_H;
16     int* d_ptr_polyPts;
17     int* d_ptr_polyNrm;
18 };
```

Figure 36: Device memory and kernel launch class.

In the final application, the CUDA kernels are tasked with performing a point-in-polygon check for each of the three rendered sides to determine, on which face the thread's point resides (if any) and thus which homography matrix to use. Afterwards, the homography transform is performed as before. The execution time of the entire re-rendering loop from GUI-event to updated canvas has been measured to **5ms**.

Figure 37: Input image with wireframe from pose estimation and re-rendered output image in interactive window.

7 CI/CD Pipeline for Docker Images

CI/CD Platforms

Using Docker containers as test environments is well supported by the prevalent CI/CD platforms like GitHub Actions and GitLab CI/CD. The Docker documentation heavily favors GitHub though, so that is the platform we build our CI/CD pipeline on. Much of this pipeline's design and implementation comes directly from the ressources[11] provided by Docker DevOps engineer Bert Fischer. On Docker's own GitHub repository, one can find many useful workflow actions for working with Docker in GitHub workflows. These can directly be used in your own workflow by using the `uses: ... <repo>/<action>@<version>` syntax.

CI/CD Pipeline

The implemented CI/CD pipeline is a classic pull request based workflow: When work starts on a new feature, a new branch is created and a pull request is opened, so that pushing to the new branch triggers the CI part of the workflow, where automated tests are conducted. Once the feature is done, the branch is merged back into the original branch, which now also triggers the CD part. In this case, deployment means pushing the image to the Docker Hub repository, so the changes are reflected in the image users use.

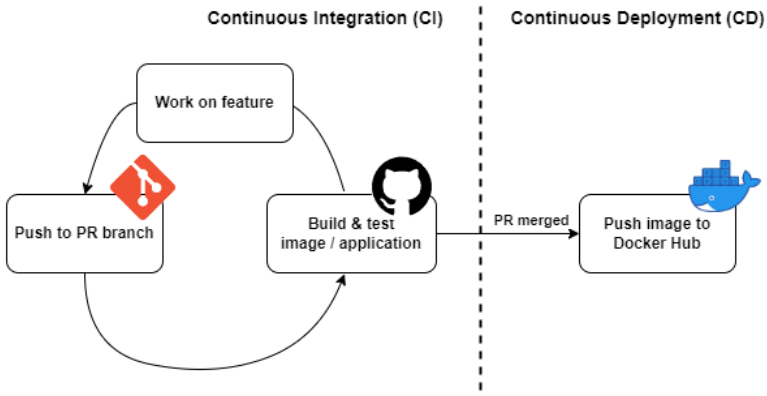


Figure 38: CI Workflow using

Building Images

The primary test for Docker images is wether they can be built or not. On Ubuntu based GitHub runners, Docker is actually available by default. However, to be able to make use of build caching like in Docker Desktop, we are going to set up the BuildKit build system using the Docker action `docker/setup-buildx-action@v2`. BuildKit is indeed the same build system that Docker Desktop uses under the hood. BuildKit can interface with GitHub's caching system to cache previously build layers for reuse. This is even more beneficial here in CI/CD, since image builds often get triggered without any changes to the image's Dockerfile. This is quite easily implemented using the `cache-to` and `cache-from` options with the `docker/build-push-action`. For the exact syntax to connect to GitHub's caching system, see figure 39. Incidentally, there are no separate build and push actions because of consistency and maintainablity concerns. So if you want to build the image, test it and then only push it, if it passed the test, build caching is very nice to have as the second build will just load the image from cache again.

Build Caching on GitHub

[11] See `https://github.com/BretFisher/docker-ci-automation/tree/main/.github/workflows`

```yaml
name: DevContainer CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    # all branches
  workflow_dispatch:
    # enables manual dispatch from github actions tab
    # (will push image to Docker Hub if tests passed)

env:
  TAG: ayomeer/cv-devcontainer-image:main

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Set up Docker Buildx (BuildKit)
        uses: docker/setup-buildx-action@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build Docker Image
        uses: docker/build-push-action@v4
        with:
          context: .
          push: false # don't push to repo until unit tests passed
          load: true  # make available locally for testing
          tags: ${{ env.TAG }}
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Run Unit Tests
        run: docker-compose -f .github/docker-compose.ci.yml run |
             app python3 ./python/unitTests.py
        # failed test return will propagate through docker run command and fail CI

      - name: Build (again) and Push Docker Image
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: ${{ env.TAG }}
          cache-from: type=gha
          cache-to: type=gha,mode=max

```

Figure 39: Full GitHub Actions workflow with build caching and unit testing based on ressources by Bret Fischer of the Docker team.

After building the image, the image's setup as well as applications being developed on it

Unit tests

can also be tested with unit test frameworks. Since the tested Cpp module already has Python bindings, the Python package `unit_test` was used. Conveniently, unit tests can simply be run over the standard Docker CLI using `docker run` or `docker-compose ... run` and test fail returns will propagate back and get caught by the GitHub workflow. A separate container launch configuration `docker-compose.ci.yml` file was created for the GitHub runner environment in the `.github` directory.

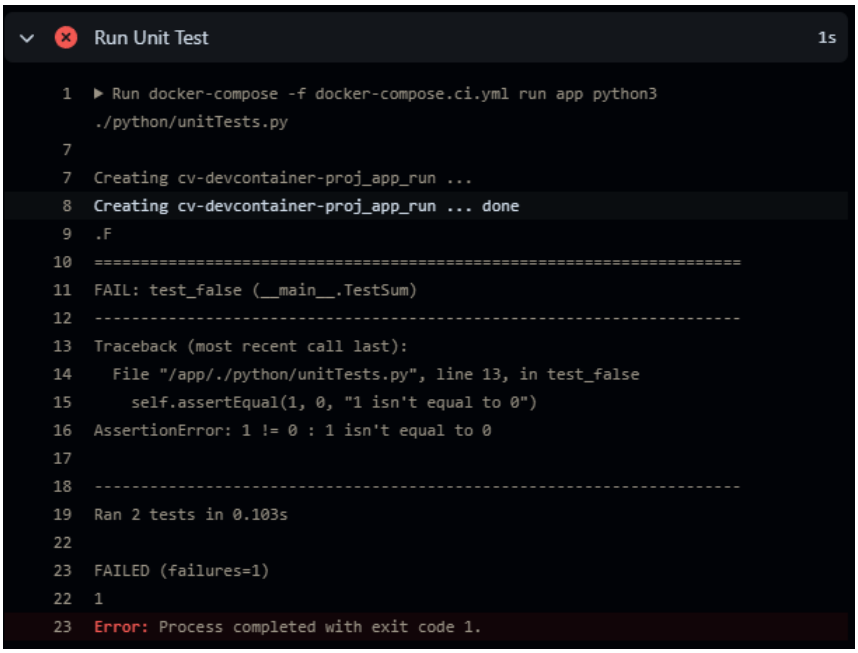


Figure 40: Output from failed unit test on GitHub Actions tab after being run through docker-compose CLI like shown in figure 39 (line 42).

8 Conclusions

The Docker platform has proven to be a valuable tool for streamlining computer vision algorithm prototyping. In most scenarios, the development environment being on a container has no negative effects for the developer. In fact the development experience is identical to working locally unless one deals with real time image input (cameras) or output (display transfer to host). While display transfer is simply limited by having to pass through a IP socket and acceptable in most circumstances, USB camera input is impossible as of the time of writing on WSL2, i.e. on Windows hosts. We were also able to confirm, that the CUDA Toolkit can be integrated into a container based development environment with good performance, usability and while maintaining portability.

9 Outlook

Providing a linux boot on lab PCs is currently in talks. For the main devcontainer this would have no consequence. It has already beed tested on Ubuntu GitHub runners throughout development and compatability is given. For the cuda devcontainer however, slight changes might be necessary to how the GPU is made accessible to the container which may involve switching from using Docker Desktop to docker-ce, as problems have been reported regarding Docker Desktop in conjunction with GPU-enabled containers on Linux. [12]

The CUDA devcontainer was built with *multi stage builds* in mind, down the line. The idea is to have one image with all tools one could possibly need, and from that create subset images, which copy dependencies for certain a certain subset of tools to cut down on the container size. If that is something that will become relevant, it is something to look into. The CUDA devcontainer is currently 11GB in size.

[12] See `https://github.com/NVIDIA/nvidia-docker/issues/1717`

10 Anti-Plagiarism Declaration

I hereby confirm that everything written is my own work and that all sources that contributed to this thesis are properly mentioned.

Date: June 2nd, 2023

Signature:

Andreas Kuhn

References

[1] Boissel, R., and Sundaram, A. Leveling up cuda performance on wsl2 with new enhancements. https://developer.nvidia.com/blog/leveling-up-cuda-performance-on-wsl2-with-new-enhancements, August 2021. Accessed: 2023-03-13.

[2] DockerDocs. Docker overview. https://docs.docker.com/get-started/overview/. Accessed: 2023-03-14.

[3] DockerDocs. Optimizing builds with cache management. https://docs.docker.com/build/cache/. Accessed: 2023-03-27.

[4] Eric Young, F. J. Image processing and video algorithms with cuda. https://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Image_Processing_and_Video_with_CUDA.pdf, 2008. Accessed: 2023-05-28.

[5] Microsoft. Create a c++ extension for python. https://learn.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2022, Octobre 2022. Accessed: 2023-05-28.

[6] Microsoft. Create a dev container. https://code.visualstudio.com/docs/devcontainers/create-dev-container, March 2023. Accessed: 2023-05-29.

[7] Microsoft. Developing inside a container. https://code.visualstudio.com/docs/devcontainers/containers, March 2023. Accessed: 2023-05-28.

[8] Murtaugh, B. Using dev containers in wsl2. https://code.visualstudio.com/blogs/2020/07/01/containers-wsl, July 2020. Accessed: 2023-03-13.

[9] Nvidia. Cuda programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide, April 2023. Accessed: 2023-04-13.

[10] Wang, W. Demystifying containers 101. https://www.freecodecamp.org/news/demystifying-containers-101-a-deep-dive-into-container-technology-for-beginners-d7b60d8511c1/, October 2018. Accessed: 2023-03-13.

A Original Problem Formulation

OST Ostschweizer Fachhochschule

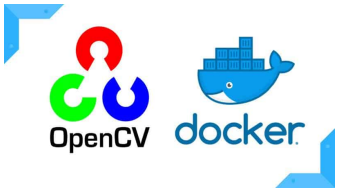
Studiengang Elektrotechnik
Fachbereich Computer Vision

A Toolchain for Computer Vision Algorithm Development

Bachelor Thesis for Andreas Kuhn

Problem Formulation

Students in the field of computer vision have limited time to explore and configure the required complex toolchains and to create a project setup needed to implement and test algorithms. In this work a toolchain and project shall be set up with examples to create python bindings for functions implemented in C++ with support of at least the C++ OpenCV library. As a means to store and reuse the toolchain and project easily, a Docker Container should be created.



Working Steps

- Getting familar with the docker concept
- Choosing the operating system within the container
- Choosing a suited method to create Python bindings
- Demonstrate the efficiency-gain of C++ code accessed via Python bindings with an attractive example
- Integration of the OpenCV C++ library
- Implementation of unit tests
- Documentation how to reuse the container and how to use the tool chain by teachers and students

Optional Steps

- Integration of the Dlib C++ library, with an example application
- Example application that maktes use of an available NVIDIA-GPU
- Example for the usage of an onnx CNN-model

Literature

Some hints to to start with:

[1] `https://learnopencv.com/install-opencv-docker-image-ubuntu-macos-windows`

[2] `https://www.sicara.fr/blog-technique/2017-11-28-set-tensorflow-docker-gpu`

Lecture materials, articles and books recommended by the supervisor, research in libraries and internet.

Report

The work carried out must be documented in a written report, which must be submitted in electronic form at the end of the work. The report must contain the unchanged assignment, a summary (1 page), as well as a concluding comment with the signature of the student must be included. Part of the comment must be the anti-plagiarism statement: „I hereby confirm, to the best of my knowledge, that everything I have written is my own work. All sources that contributed to this thesis are properly mentioned.“

Link to GitLab repository

The sources of hardware & software should be submitted digitally. The report must be enclosed at least as a pdf file. The sources of the report (text and images) are desirable. The rights to fork the GitLab repository wit the software and data created must be provided to the lecturer.

Timeframe

Beginning of the work: Montag 6.2.2023
Submission of the report: Donnerstag 10.06.2023

Other dates

Dates for short presentation and weekly meetings by verbal agreement.
Rapperswil, 3. Februar 2023 Prof. Dr. Martin Weisenhorn