Tampere University

Jyry Uitto

OFFLOADING COMPUTATION WITH A MINIMIZED OPENCL RUNTIME FROM A NANO DRONE

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
October 2022

Abstract

Jyry Uitto: OFFLOADING COMPUTATION WITH A MINIMIZED OPENCL RUNTIME FROM A NANO DRONE
Master's thesis
Tampere University
Master's Degree Programme in Software Development
October 2022

The amount of resource-restricted devices is increasing rapidly with the common adoption of mobile phones and other small microcontrollers residing on battery-powered platforms. This number is only going to increase in the future with more devices being created and connected to the internet of things. Devices with limited resources cause energy efficiency to become a key limiting factor for their computational capabilities. This creates the incentive to save resources on computation that has to be done for applications running on resource-restricted devices. In this thesis, computation offloading is researched as a solution for preserving resources on a resource-restricted device by computing resource-intensive computation on a remote server. Being able to offload computation can aid the resource-restricted platform by saving resources locally and it can also enable more complex computation tasks to be run with the remote server hardware. Furthermore, offloading computation can enable running more complex algorithms in real-time that the resource-restricted would not be capable of. The key challenges in offloading computation over a wireless network are the transmission cost of the data to and from the edge server and the latency that is caused by offloading.

This thesis provides a proof of concept for wireless computation offloading on a nano drone with a minimized OpenCL runtime. The proof of concept attempts to improve the energy available for other functions on the drone and to enable more complex computation by offloading computation onto heterogeneous servers. In this thesis, the Portable Computing Language implementation of OpenCL with Remote extension capable of offloading computation was ported onto the AI-deck extension of the Crazyflie 2.1 nano drone. Then the platform was used to offload computing onto an edge server capable of running OpenCL kernels. This solution was then benchmarked to search for the minimum memory requirements to enable the platform. Furthermore, the proof of concept implementation for offloading computation through OpenCL API provides a latency overhead caused by offloading computation.

Keywords: Distributed systems, Resource restricted devices, compute roaming,

Edge devices, Edge computing, OpenCL, PoCL-R.

Contents

1 Introduction

The amount of resource-restricted devices is increasing steadily with the adoption of mobile phones and other battery-powered devices such as electric cars and drones. Resource-restricted devices often have some computation tasks that are required for their operation and they must use their limited resources to compute those tasks.

What if this was not the case? With the current state of affairs where almost every device is connected to the internet, there is an opportunity for new solutions to increase the capabilities of resource-restricted devices. In this thesis, offloading computation is the method explored as there are many types of resource-restricted devices that could benefit from offloading computation such as mobile phones or nano drones. In the offloading computation scenario, the resource-restricted device would search for a server and then transmit the data that has to be processed onto that server which will compute the data and then send the result back to the resource-restricted device. Being able to showcase the minimum hardware requirements that are needed to enable offloading computing would create a baseline for hardware requirements. With this baseline, a device would be able to compute any sort of computation-intensive task with the cost of transmitting the data required for the computation.

Two key benefits that offloading computing across a wireless network can provide are decreased energy consumption as the computation is done on the server instead of the device. Furthermore, if the computation required is parallel, then it can theoretically utilize all of the hardware that the edge server contains providing increased computation speed. If the increase in computation speed is more than the transmission time of the data, then this would also be a faster way to get the result onto the host device. Increased computation speed of the edge server can also be used to compute more complex algorithms that could not be computed on the drone in real-time. On the other hand, the latency of the connection to the server might cause problems for real-time computing and the transmission costs of data over a wireless network can cause the computation offloading to consume more energy on the resource-restricted device than is saved by computing the task on the edge server.

In this thesis, a proof of concept was created for showcasing wireless computation offloading on a nano drone that has very limited memory available. The Open Computing Language (OpenCL) (Khronos® 2009) was used to enable heterogeneous computing in the proof of concept. The Portable Computing Language (PoCL) with Remote extension (PoCL-R) (Solanti 2020a) was used as an OpenCL implementation that was ported onto the AI-deck (Bitcraze 2022b) expansion deck of Crazyflie 2.1

(Bitcraze 2022e). This PoCL-R implementation was then used to offload OpenCL kernels over a wireless network onto a remote server to be computed. The offloading runtime was then compared to running a local version of the kernel on the processor of the AI-deck itself followed by a discussion of the metrics of latency, memory usage, and energy consumption of the proof of concept.

The structure of this thesis is as follows. In the next Chapter 2 the theory related to this proof of concept is introduced. The following Chapter 3 explains the layers of standards that the proof of concept will use and then the drone development framework is discussed in Chapter 4. Chapter 5 describes technical details of the implementation and in Chapter 6 the solution is benchmarked. In Section 6.4 the evaluations of the runtimes offloading computing are discussed and compared to the locally computed kernel. Section 6.5 presents the next development steps for the runtime and then conclusions are drawn in Chapter 7.

2 Distributed Systems

The computing power of a single processor core has not increased with Moore's law (Moore 1998) as in the past, yet the need for more computing power has not decreased with the stagnation of performance improvements on a single core. To circumvent the performance limitations of a single core, multicore systems were introduced. Multiple cores allow for more performance to be achieved from a single processor. Yet as there are more cores in close proximity to each other the heat generation increases, which will limit the available processing power yet again (Esmaeilzadeh et al. 2012).

As a next step computation can be dispersed onto a distributed system that contains multiple processors. This is made more feasible by the fact that efficient usage of multicore processors requires parallelism in applications and this parallelism is required to run the application on multiple processors. Then to further increase the available computing power of the distributed system more processors can be provided to the system.

The scale of size for distributed systems is huge. On the other end, we have the internet which contains billions of devices, yet on the opposite end, the smallest unit that can be counted as a distributed system is a chip that has two distinct microcontrollers communicating with each other (Steen and Tanenbaum 2020).

Another defining attribute for distributed systems is that the microcontrollers have their own memory (Tanenbaum 2015). Thus true monolithic systems are more of a rarity since the hardware of devices usually contains specialized processors for specific tasks. It is possible however that a closed system on a microchip would be monolithic when all the microcontrollers on it share the same memory. Furthermore, it appears to be monolithic to the outside of the chip if it provides only a single communication channel outwards.

Utilizing many devices can be more energy consuming as transferring data around has a cost (Aslan et al. 2018). However, there are technologies such as Dynamic Frequency and Voltage Scaling (DFVS) that allow for more energy-efficient computation of the problem. Through the usage of multiple devices on optimal frequency instead of maximum frequency on a single device energy saving can be achieved (Rabaey 1996). Furthermore, another DFVS technique called race to idle can be used. Computing the task quickly and using the sleep state of the device can provide energy savings (Albers and Antoniadis 2014). Thus even though distributed systems cause a cost in transferring the data around, it is not necessary that the whole process is more costly as energy can be saved on the computation itself.

The use case of the distributed system has an effect on what kind of proximity

of computation resources best fits their purpose. The two opposite ends of the spectrum are Cloud computing and edge computing.

Edge computing is a paradigm where the computation happening on the distributed system is designed to occur as close to the sensor device as possible. This provides certain benefits in the form of faster response time from the server, since it is closer to the device. The fact that the message travels as few steps as possible has the effect of reducing communication bandwidth consumed from the internet (Marcham 2019). Edge computing is usually used as a paradigm when latency is the key metric in the distributed system. Distributed systems for edge computing tend to be more heterogeneous since the closest available servers are taken into use.

Cloud computing stands at the other end of the spectrum. In a cloud computing environment, there is a data center ready to compute on the data. However, communication delays are greater than in edge computing since the message has to travel all the way to the cloud server. This causes increases in the amount of bandwidth consumed as well since the number of messages traveling across the internet grows (Ruparelia 2008).

2.1 Properties of distributed systems

There are more plausible benefits to distributed systems than only being able to scale more computing power towards a problem (Steen and Tanenbaum 2020). On a device such as a mobile phone, you have only a single processor to compute the result of a computation, but if the problem is distributable you can have multiple computing clusters computing the problem. In addition, availability and resource sharing are benefits of distributed systems. The fact that a high-performance computing cluster can be used for more than one device at a time allows far greater utilization of available resources. Having higher utilization of available hardware allows less hardware to perform more tasks.

The geographical distribution of systems allows for good accessibility of systems. The possibility of connecting to a server from a mobile phone and taking the results where you need them instead of going to the server hall to retrieve the calculated result saves time and effort. Having this kind of high availability and accessibility of efficient resources is good for many use cases. When computation can be offloaded to these servers from anywhere and at any point allows many services, such as banking to exist. In addition, the amount of hardware is reduced since everyone does not need their own server hall.

One more beneficial thing for distributed systems is the possibility of redundancy. Being able to offload the same computation to multiple servers at once gives a safeguard against hazards. If some of the servers were to crash it would be tolerable as long as at least one result would arrive back.

The downsides of distributed systems are the fact that introducing more hardware and moving parts to a problem will create more surface for problems and errors to occur. For example in the use case of the internet, a single message sent from one place to another crosses multiple access points forwarding the message. The amount of nodes increases with the distance of the target of the message and all of these nodes have the possibility of an error occurring so that the message is either corrupt or lost. Furthermore sending messages from one microcontroller to another always incurs a cost in latency. Sending data over the internet and possibly lots of it can create a latency bottleneck for the distributed system.

2.2 Coupling of distributed systems

Distributed systems have two separate design categories in use (Osrael et al. 2006). These two are tightly coupled systems and loosely coupled systems. The key difference is that a tightly coupled system is not functional if a part of it is missing, whereas a loosely coupled system can still operate. A system in the bigger picture can be both tightly and loosely coupled at the same time. Coupling is usually private between individual pieces of the system.

Tightly coupled systems are usually created when the operation of the rest of the system does not make sense if a part of crashes due to an error. For example, a banking application on a phone requires that the bank's cloud service is properly running. This is a great example of a tightly coupled distributed system since the application is not able to work without the cloud service.

Loosely coupled systems usually come into existence when parts of the system can be replaced at a whim. While offloading some computation onto a server it does not really matter which server it ends up on. Another variation of this is a system that is able to compute the data locally in a case where the server is lost or does not exist.

Furthermore, a mixture of both types in a single system is not uncommon. Having one part of the system tightly coupled and another loosely coupled is normal. In a personal computer, the processor is tightly coupled to the motherboard but the keyboard that can be changed at a whim is loosely coupled.

2.3 Fault tolerance

As the distributed systems introduce more components to a system the need for fault tolerance arises. Fault tolerance is the capability of the system to recover from a crash or an error that occurs over runtime.

Fault tolerance can be done in many different ways but when communication over to other devices timeouts are quite natural (Steen and Tanenbaum 2020). If a

device does not respond over a period of time then the message is resent or another resolution to the issue is taken. This can happen naturally when communicating over the internet since there is a chance that a node connecting the devices could crash.

Different kinds of distributed systems tolerate faults differently. A tightly coupled system tolerates faults rather badly since the continuation of the work does not necessarily make any sense if a critical part of the system is malfunctioning. In this case, a gracious shutdown is the best course of action to take.

In the case of a loosely coupled system, the error handling can be quite different. If for example, a computation server crashes during operation, then a new one can be searched for when the realization occurs that the first server is down, and afterward the system can maintain operation as it were.

2.4 Resource restricted devices

Resource-restricted devices are devices that are somehow limited in their operating capacity. The usual scenario is that the devices are battery powered which brings limitations to how long they can operate. With only a single limitation of battery power, a device could operate indefinitely if the battery would be big enough. However, with multiple limitations occurring at the same time the resources available can be very limited. For example, creating a nano drone would impose strict limitations on the weight and size of the device and thus the amount of resources available can be very limited.

There are more and more resource-restricted devices emerging with the internet of things. Sensors and small microchips required for the internet of things have strict weight limitations and this restricts the computation power available. Sensors and small microchips required for the internet of things ideally consume as little resources as possible to not consume any more of the battery than necessary. This can impose serious limits on the computation power of sensors and microchips.

2.4.1 Overcoming resource restrictions

There are many ways to lift the limitations imposed by resource-restricted devices. Until now the shrinking size of the transistors has been the standard way of reducing the energy consumption of devices. Smaller transistors provide less energy-consuming microcontrollers and thus they alleviate the restrictions imposed on resources. However, Moore's law is slowing down and other solutions have to be searched for.

The design of all microcontrollers is strolling forward towards ever more energy-efficient designs. The emergence of new ultra-low power microchips that require less

than a milliwatt per billion operations (GreenWaves Technologies 2022). These are not as fast as server microchips, but they do provide energy-efficient computing and very low power consumption for computation on edge devices.

The method that is researched in this thesis for overcoming the limitations of resource-restricted devices is offloading computing. Being able to transfer the computation onto a device with less or no restrictions to its power usage or computation capability provides a way to save energy on the resource-restricted device itself. As long as the transfer of data consumes less energy than processing the data locally on the resource-restricted device it can be deemed worthwhile.

2.4.2 Offloading computation

Offloading computation means transferring tasks that have to be computed away from the processor or device that requires the result. In practice, the main processor can offload a task onto another processor that handles that task while the main processor can focus on other tasks. This can have a multitude of benefits. As the main processor can focus on the remaining tasks the performance will be increased as there are fewer tasks to be done by the main processor. Furthermore, there are more resources available to operate on the problems overall which can increase performance as well. Additionally, energy can be saved from the main processor as the processors handling the offloaded data will pay for the processing of the data. The processing cost can also be lower for the processor the task is offloaded to since offloading can be done onto specialized heterogeneous hardware that is built for specific tasks. Heterogenous devices such as hardware accelerators can provide substantial performance and energy benefits. Saving energy on the main processor has an even more profound effect when the offloading occurs from a resource-restricted device onto another device with a different power supply.

The only requirement for computation offloading is a communication medium that can be used to transfer the data required for the computation. Using the communication medium requires a language both ends of that medium can understand. It can be an internet communication protocol if the communication is happening over the internet or a communication method for communication over a wire that connects microcontrollers.

The first issue with computation offloading is the fact that transferring data is never free. Transferring data over a communication channel incurs a cost of energy and time. This cost is further increased by the fact that sending data to be computed elsewhere needs to have the result sent back. In very latency-intolerant tasks this is more of a problem since having critical data arrive too late for a device in real-time flight might be disastrous. However, in this thesis, a look is taken at low latency implementation for computation offloading and the baseline metrics for the latency

cost incurred.

The second problem arises with communication as well. What if the computation on the device that we offloaded the computation to fails and the information never reaches the main processor? In essence, a system that does computation offloading requires some form of fault tolerance, otherwise, it will be very likely to crash when a message is lost or some part of the system crashes. Error tolerance can be brought to a computation offloading system by To avoid total failures when an error occurs the system offloading computation can be created to be less tightly coupled. More loosely coupled offloading of computation can be achieved by giving the program the ability to realize that a computation task is lost and then handle the problem by either sending the task to be computed again or dropping that task and continuing with processing.

However, fault tolerance is not required for computation offloading, and tolerating faults on a tightly coupled system might not be necessary. If a part of the system has malfunctioned then the rest of the system does not necessarily make any sense anymore.

2.4.3 Compute roaming

Compute roaming is the next step from offloading computation that builds on the wireless network roaming. In compute roaming the device that is roaming is actively searching for access points for an internet connection. Through the internet connection, it seeks for known computation resources to offload to or to computation resources broadcasting their availability for example through service location protocol (Day et al. 1999). The essential thing of roaming networks is the capability to reconnect to another access point when a better connection is available or the first access point drops the connection (Goransson 2007). After reconnecting to the access point the device can then re-establish the session it had or create a new one using the new connection as can be seen in Figure 2.1.

There are multiple issues the runtime on the device has to account for to make compute roaming possible. First, the device compute roaming has to regularly ping for new access points and the connection quality of the current connection. Connecting to an access point with a good quality connection is not the only thing required for compute roaming as the computation resources have to be located through the connection.

Additionally, for the device to be able to connect to new access points some form of fault tolerance is required. If the runtime on the device does not know if it is connected or not then the operation is not deterministic. Furthermore, the runtime has to be able to resume operation at any point of the communication cycle occurring with the server as the connection can change at any point. Thus if a connection

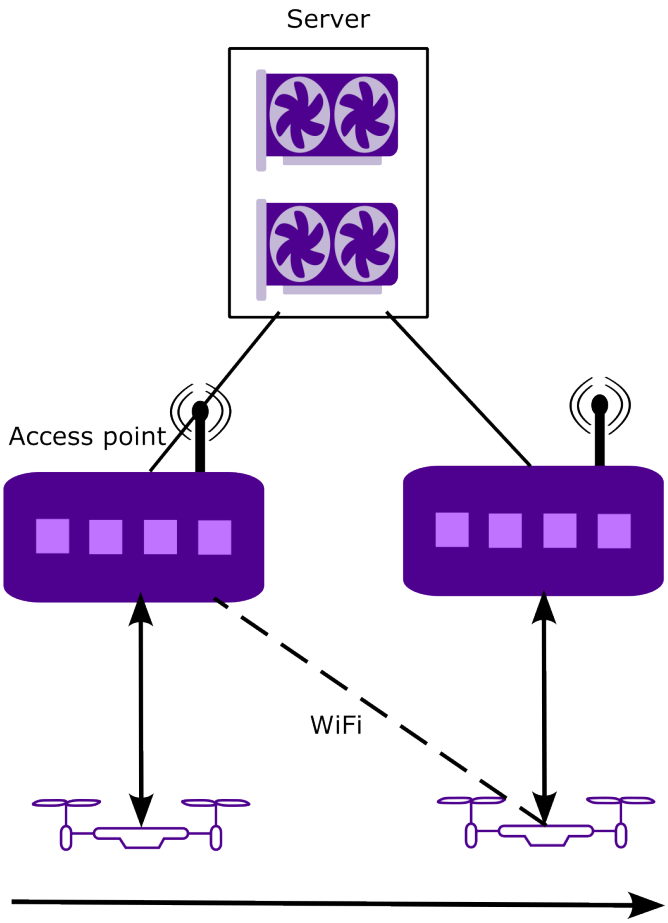

Figure 2.1 *Compute roaming*

is dropped then the first step of fault tolerance is to reset the runtime to a state where it can resume functionality when the connection is re-established. Another issue is the fact that it can not be known if the connection will be established back to the same server as previously. However, in an ideal scenario, the connection is re-established into the same server and the same session through any access point available. In the ideal case, no computation is necessarily wasted and the program can continue on the exact step the connection was dropped at.

2.4.4 Cyber foraging

Cyber foraging is similar to compute roaming. The key difference is the fact that a cyber foraging system is adjusting the computational requirements of the programs run on a device according to the resources available to it through remote resources. These remote resources are made available as surrogates which are remote computers capable of temporarily assisting the cyber-foraging device. A cyber foraging system can then connect to these surrogates to use their resources for computation or be forwarded to further resources through the surrogate. (Flinn 2012)

The ability to use the microcontrollers and chips lying around to create surrogates could provide a substantial amount of free computation power. With the emergence of more and more internet of things devices, the amount of idle processing power everywhere is increasing and in a current household, there might be a smart fridge, smart oven, smart washing machine, smart robot vacuum cleaner, or smart anything that most likely holds some computation power that is just idle. Being able to harness this idle computing power as surrogates for cyber foraging would be ideal (Satyanarayanan 2001).

2.4.5 Energy efficiency of computation offloading

All of the aforementioned methods of offloading computation will increase the total amount of processing required to compute data since they require the data to be transferred to the device that will process it. However, in the case of resource-restricted devices, the additional cost can be transferred to the device that will do that computation. Thus if the cost of transferring the data when offloading computation is lower than computing it locally on the device, then in theory the resources of the resource-restricted device are saved.

A further factor to the feasibility of offloading is the question of what can be done with the time and energy saved by transferring the computation. If the device doing offloading stands idle and that idle state consumes more energy than is saved with offloading, then the offloading is again not worthwhile. On the other hand, if the idle time between sending the computation task to be computed and receiving the answer can be used to compute other tasks the cost of offloading can be reduced.

There are more methods that could be implemented to save energy on devices offloading computation. As the energy consumption of microcontrollers increases with their frequency, a form of Dynamic Frequency and Voltage Scaling (DVFS) could be implemented to lower the operating frequency of the offloading device. The DVFS could be taken even further with the implementation of a race to idle functionality and thus when complex tasks are offloaded and computation is happening elsewhere the offloading device could go into the idle state to save energy.

The breaking point for beneficial offloading can be calculated using Equation 2.1. In this Equation, LC stands for Local Computation and S stands for Sending data, R for Receiving results, and IT for Idle Time. As can be seen from the equation, offloading computation is beneficial for local resources while the cost of local computing is greater than the cost of sending, and receiving data, and the energy consumed while waiting idly for the result to be computed and transferred across the network nodes. As the primary goal is to reduce the energy consumption of the local device itself the possible additional energy cost paid by other devices is of no consequence.

$$LC > S + R + IT \tag{2.1}$$

2.4.6 Latency of computation offloading

One deterrent for computation offloading is the fact that it causes latency and in many environments increased latency is intolerable. For example, using a mobile phone and having everything occurring a second later would lead to a great usability drop rendering offloading unfeasible.

In the case of latency, sending data to be processed on a remote device can be beneficial to the time required to get the result and it can be calculated with the Equation 2.2. In this Equation LCT stands for Local Computing Time, ST is Sending Time, RCT is Remote Computation Time, and RT is Response Time. As can be seen, offloading can be beneficial for latency as well if the local time it takes to compute the result is longer than the combined time it takes to send the data to the server that computed the data and sends the data back. Reducing the cost of sending and receiving data in offloading computing is the key driver of edge computing (Marcham 2019).

$$LCT > ST + RCT + RT \tag{2.2}$$

3 Standards enabling portability

If the development of software would have to start from scratch every time a new microcontroller is introduced to the world, creating a new working device would take far longer than today. Furthermore, interoperability between devices would be nonexistent since there would be no common language between any devices. The solution provided for speeding up the development and enabling interoperability of new devices and microcontrollers is standards.

There are standards for practically every aspect of the development of a microcontroller. There are standards on how to implement communication between devices, how to run a specific processor, and in what format one should write application code. Standards often have an official maintainer that reviews implementations created of their standard. If an implementation fulfills the standard it can be given a stamp of legitimacy. The end result of implementing a standard usually is to provide an Application Programming Interface (API) that can be used to run the functionality provided by that standard. This is beneficial because then another layer of software can be built that relies on the functionality provided by that API without knowledge of the implementation details of the API call itself. Overall this allows the porting of new functionality to new devices to be rather parallel as development can be done on the program itself and enabling the API calls that the program needs at the same time as it is most likely known what specific standards and API calls the program requires.

In this chapter, the standards implemented and available on the Crazyflie 2.1 (Bitcraze 2022e) nano drone are viewed. In the first section, the communication standards are explored and in the second section, the stack of standards that are required for the proof of concept to work are reviewed from low level to high level.

3.1 Communication standards

There are many different communication protocols available on the Crazyflie 2.1 nano drone and further methods can be made available with expansion decks. There are different communication standards for the internal and external communication of the drone combined with additional communication protocols created by Bitcraze that allows communication between microcontrollers using available communication standards.

3.1.1 Internal communication

Microcontrollers in embedded devices are connected through wires on the board. To enable the microcontrollers to communicate with each other through those wires the microcontrollers have to understand the signals sent and received on those wires. To help with communication between devices there are many standards made for over-the-wire communications.

The microcontrollers on Crazyflie 2.1 and AI-deck communicate with each other through the Serial Peripheral Interface (SPI). The SPI is a synchronous serial communication standard used as a common communication standard in embedded devices. The advantages that have caused it to emerge as a standard for communication are simplicity and small size in hardware design, simple software implementation, and high throughput. (Motorola, Inc. 2022)

Another standard that is used by the Crazyflie 2.1 and AI-deck to communicate between the drone and the expansion decks is the Universal Asynchronous Receiver-Transmitter (UART). UART is an energy-efficient and fast communication method that requires only a single wire between communication targets but has to be properly configured on both ends of the communication channel to function (Nanda and Pattnaik 2016; Gupta et al. 2020).

On top of these Bitcraze (Bitcraze 2022c) has developed two different communication methods for communicating with the drone's microcontrollers. The original communication method is the Crazy RealTime Protocol (CRTP) (Bitcraze 2022d) which is used by the cfclient in the development environment to communicate with the Crazyflie 2.1 drone. This protocol enables fast real-time communication that is used to control the flight on the Crazyflie.

The Crazyflie Packet Exchange (CPX) (Bitcraze 2022f) is a newer addition on top of the CRTP protocol with the ability to talk to any microcontroller on the Crazyflie drone, the AI-deck expansion deck, or with the host client. The CPX provides an API with which one can create and define CPX packets and then use a function to send the CPX packet according to the metadata supplied to it. The CPX API hides the details of the underlying functions doing the transmission and routing of the data between microcontrollers. Behind the API the CPX is using the SPI and UART communication protocols as required to transfer the data between microcontrollers on the drone or WiFi and radio to communicate outside the drone.

This hiding of implementation details eases the usage of communication between the microcontrollers and the host client as the user can use the API provided and the communication is handled for the user. The CPX protocol creates a similar environment to the internet stack where one can send a packet and the underlying protocols handle the forwarding of the packet.

3.1.2 External communication

The external communication on the Crazyflie 2.1 itself is done over the radio and this is the default communication that is intended to be used by developers. The radio is able to program the drone and send data back to the console running on the development environment. In addition to radio communication, the radio is capable of Low Energy Bluetooth communication.

The AI-deck expands the external communication capabilities of Crazyflie 2.1 by providing a microcontroller capable of WiFi. WiFi is the method of communication that is required in the proof of concept work and in order to establish WiFi communication for the proof of concept, only the API provided by the microcontroller for connecting to a WiFi access point has to be used. The internet communication is then handled by the Berkeley Socket API (The Open Group 2022) calls on the microcontroller.

The microcontroller firmware implements the lower levels of the Open Systems Interconnect (OSI) model (Zimmermann 1980) and the developer only has to use the socket API to control the TCP/IP (Lammle 2018) communication which is regarded as layer 4 in Figure 3.1. The hiding of implementation details of the lower levels of the OSI layer through providing interfaces has benefits. This allows the developers to use higher-level API calls allowing for faster development, since then the developer does not have to know the more precise details of how the communication is truly happening on the hardware.

3.2 Layers of functionality to enable offloading computing

In this section, the standards and their implementations that are in use for the proof of concept are walked through. The standards follow the same principle as the OSI model and are built on top of each other in layers. Lower-level layers provide functionality that enables the higher-level layers of Figure 3.2.

Splitting the implementation into layers of standards and functionality aids with portability and hastens development. API provided by lower layers can be used by the higher layers without knowing the details of how they are implemented.

In the following sections, the standards in Figure 3.2 are explained in detail starting from the application layer in the figure and proceeding to lower-level layers.

3.2.1 Application

The first layer in the model is the application. In the application layer, the functionality required is coded into the program through available API calls. One can use known available API to achieve the functionality desired without knowing the details behind the API calls.

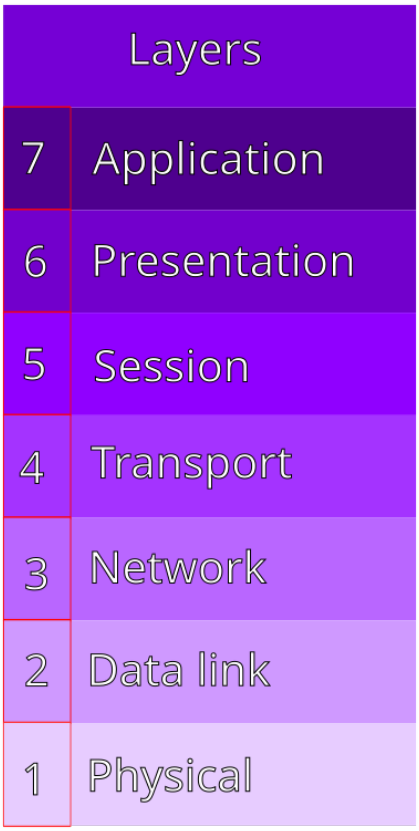

Figure 3.1 *OSI model*

In the thesis, an application was written using high-level API calls that use standards implemented on the platform to achieve the required functionality. The application written according to those standards would work on any platform that is capable of running the same standards because it relies on the fact that the API calls exist and not how they are implemented in practice. This allows for great portability since if the required standard is fulfilled on a device, then the program should run on that device without problems.

3.2.2 Open Computing Language

In this thesis, the application that was built relies on OpenCL (Khronos 2022). OpenCL is an open standard that specifies an API through which users can take advantage of computation capacity residing on different types of hardware. The computation done on systems utilizing various types of hardware is known as heterogeneous computing.

OpenCL computes data through small programs called kernels. These kernels have to be compiled according to the hardware of the device that they are meant to be run on. The kernels can be compiled before running the program which is called offline compiling or during the program runtime which is called online compiling.

Figure 3.2 *Functional layers*

There are many implementations of the OpenCL standard that provide the API described in the OpenCL specification (Khronos 2022). To fulfill the OpenCL standard according to the specifications the implementation has to provide the API therein. The OpenCL specification does not care how the API calls are implemented, but only that they provide the functionality that they should. Implementations of OpenCL do not need to have the capability for online compiling of kernels but a method of compiling kernels is required.

OpenCL can be run on any device that provides a driver for OpenCL. The driver provides the OpenCL implementation information about how to run that specific device. That is the only requirement implementations have for achieving functionality on a device and this makes OpenCL very portable.

3.2.3 Portable Computing Language

PoCL is an OpenCL implementation that aims to be easy to port onto new platforms (Jääskeläinen et al. 2015). PoCL is created to be modular which enables users to turn off and remove parts of the implementation that they do not need. If there would be no modularity then it would be impossible to fit the implementation onto the nano drone. LLVM (LLVM 2009) is used as a kernel compiler on PoCL and

the size of LLVM as a module is in gigabytes. Not being able to turn the kernel compiler off would make the porting task onto a nano drone impossible.

Furthermore, the PoCL is extendable and in this thesis, the PoCL-R (Solanti et al. 2022; Solanti 2020b) extension was used to enable remote server functionality. The PoCL-R extension enables a driver for communication with a server in the backend. On the application side, the OpenCL API calls are exactly the same as they would be without the extension, meanwhile, the PoCL-R is forwarding the OpenCL commands in the background onto a server.

3.2.4 Portable Operating System Interface

Portable Operating System Interface (POSIX) is an operating system standard (I. The Open Group 2018). It is made to enable compatibility between different operating systems. In essence, the code compiled that runs on a POSIX interface should run without problems with any POSIX-compliant system. Furthermore operating systems can be made POSIX-conformant which entails that they have POSIX API calls available.

POSIX also provides POSIX threads (Pthreads) that are used for multithreading and running parallel applications. It is a fairly common standard that is provided by many operating systems, especially those that are POSIX-conformant.

To provide portability across operating systems, programs can be built to use POSIX API calls. Creating a program that uses POSIX API calls is easy to port onto any platform that has POSIX conformance. This is the scenario with PoCL-R as it uses Pthreads to run, thus making it portable onto operating systems that have Pthread API calls that function.

3.2.5 Free Real-time Operating System

Free Real Time Operating System (FreeRTOS) is a real-time operating system that is commonly used to run embedded devices. It is a lightweight operating system that has modularity and extendability. The operating system kernel can be built to be as small as 9 kilobytes which allows it to be run on very resource-restricted devices (FreeRTOS 2022a).

The FreeRTOS is not a POSIX-conformant operating system and thus the core implementation is not able to run Pthreads. However, the FreeRTOS does have a FreeRTOS-Plus-POSIX (FreeRTOS 2022b) extension that provides a subset of POSIX threading API. The extension redirects the Pthread API calls to use the internal API calls of FreeRTOS. This allows running programs that make use of the Pthread API and that helps with porting new programs onto the operating system.

3.2.6 Board Support Package

To enable FreeRTOS to run the Board Support Package (BSP) is required. BSP is the first software layer that is built on top of the hardware. The purpose of BSP is to provide access to all available functionality on the hardware Through API calls. The FreeRTOS runtime is made available by linking the necessary FreeRTOS functionalities to use the BSP API calls.

3.2.7 Hardware

Hardware is the layer on which software is built upon. Even the hardware has to fulfill standards as it is built according to schematics to be the hardware that it is. Properties built onto the hardware are built according to standards as well to provide capabilities such as memory that work as intended. The hardware defines what functionality can exist in the first place and the BSP is programmed according to the properties available.

Embedded devices can contain many different components such as the processor. These components inside embedded devices are rarely made from scratch and as such there are ready-made implementations that follow defined standards. For example, many processors on embedded devices are built upon implementation of the RISC-V instruction set architecture (RISC-V International 2022; Waterman 2016). This helps with creating new programs for the devices since the compiler for RISC-V instruction set architecture already exists. Furthermore, the BSP functionality is portable since similar components can exist on different devices.

4 Crazyflie 2.1 development framework

In this chapter, the development environment that Bitcraze has created for developers working with the Crazyflie 2.1 (Bitcraze 2022e) nano drone is gone through. Following the development environment, the structure of the Crazyflie 2.1 drone used in the proof of concept for offloading computing is viewed along with the expansion decks attached to it. After that, the communication methods used by the drone to communicate with the client, expansion decks, and the server are glanced into. Finally, a peek is taken into FreeRTOS operating system that is run on every microcontroller used by the Crazyflie 2.1 and AI-deck.

Bitcraze has created a development ecosystem for the Crazyflie 2.1 drones in which they have software frameworks that provide tools to ease the development for the Crazyflie 2.1. For example, with the cfclient (Bitcraze 2022g) Bitcraze enables users to scan the radio frequencies for active Crazyflie 2.1 drones and then flash their builds through the radio and also try and recover failed builds on the microcontrollers on the drone. In addition through this radio connection that can be established to the drone, the drone can be controlled by the cfclient directly or users can attach any controller onto the host running the cfclient and that can be used to control the drone through the cfclient. Furthermore, the cfclient brings a custom console that can get output from the drone through the STM32 (STMicroelectronics 2022) microcontroller and the radio connection. The cfclient has a built-in logging framework that allows for easy logging of data and actions taking place on the drone. The logging framework was used in this proof of concept to log the data required for evaluations. The cfclient uses cflib (Bitcraze 2022h) which provides API for all functions related to controlling the drone. This API is brought in as an import to allow users to take full advantage of the library provided for the Crazyflie 2.1 drone.

The software stack does not come without its own problems though. All of the microcontrollers have their own GitHub repository for programming. This inherently brings in so many libraries and versions of those libraries that the environment is cumbersome and difficult to install on the user's computer. To tackle this issue Bitcraze provides Docker containers (Docker 2022) that have all the libraries installed on them to run the development environments for each of the microcontrollers. The development code is mounted to the container through a folder that allows the developer to have a standard project locally and environment details are hidden inside the container. This solution is fairly ideal to get started with developing the drone firmware to suit one's needs, yet containers cause issues when users try to change details in the implementations of the environment that are hidden

inside the container such as FreeRTOS parameters.

For the hardware, Bitcraze has provided a good development framework for the Crazyflie 2.1 drone. The drone hull has a pin grid built into it which can be used to attach any of the expansion decks that is available for the Crazyflie 2.1 drone. In addition, the pin grid connectors do not require soldering to use which makes changing the expansion decks effortless. There is a wide variety of expansion decks available for the drone from the attachable Micro SD card deck to positioning systems such as the Lighthouse positioning deck that uses HTC Vive base stations or other Lighthouse implementations for high-precision positioning. Users could also build their own expansion decks that attach to the pin grid if the need for such specialized hardware were to arrive. Conveniently there is also a deck that enables WiFi connection and provides additional computing resources called AI-deck (Bitcraze 2022b) that was used to host the software stack required for the proof of concept. All of these expansion decks are changeable in case they break in use and furthermore the motors, propellers, and engine frames of the Crazyflie 2.1 can also be changed by the user.

4.1 Drone setup for offloading computation

For the offloading computation in the proof of concept, the AI-deck was required to provide the ESP32 WiFi controller chip that could handle the wireless connection and internet communication. In addition to that, the AI-deck provided the drone with more computation resources and a GAP8 ultra-low power processor. In practice, the Crazyflie 2.1 also required some form of flight stabilization since the flight of the drone was unstable. First, the Lighthouse deck was tried to be used as the stabilizer but the expansion pins had overlapping communication pins which caused the drone to malfunction. In the end, the Flow deck v2 was used because it did not have overlapping communication pins with the AI-deck. The Flow deck v2 enabled truly autonomous flight and program runtime without interference from users. The Flow deck v2 communicates directly with STM32 microcontroller on the Crazyflie 2.1 to keep the drone hovering in a certain location at a certain altitude.

In the next sections, the properties of the base Crazyflie 2.1 drone and the AI-deck expansion deck are going to be discussed. The Flow deck v2 will be omitted since it required no further programming to functionally stabilize the drone flight and the deck only communicates with the Crazyflie 2.1 drone without interfering with other microcontrollers. The communication connections between the microcontrollers of the Crazyflie 2.1 and AI-deck are shown in Figure 4.1.

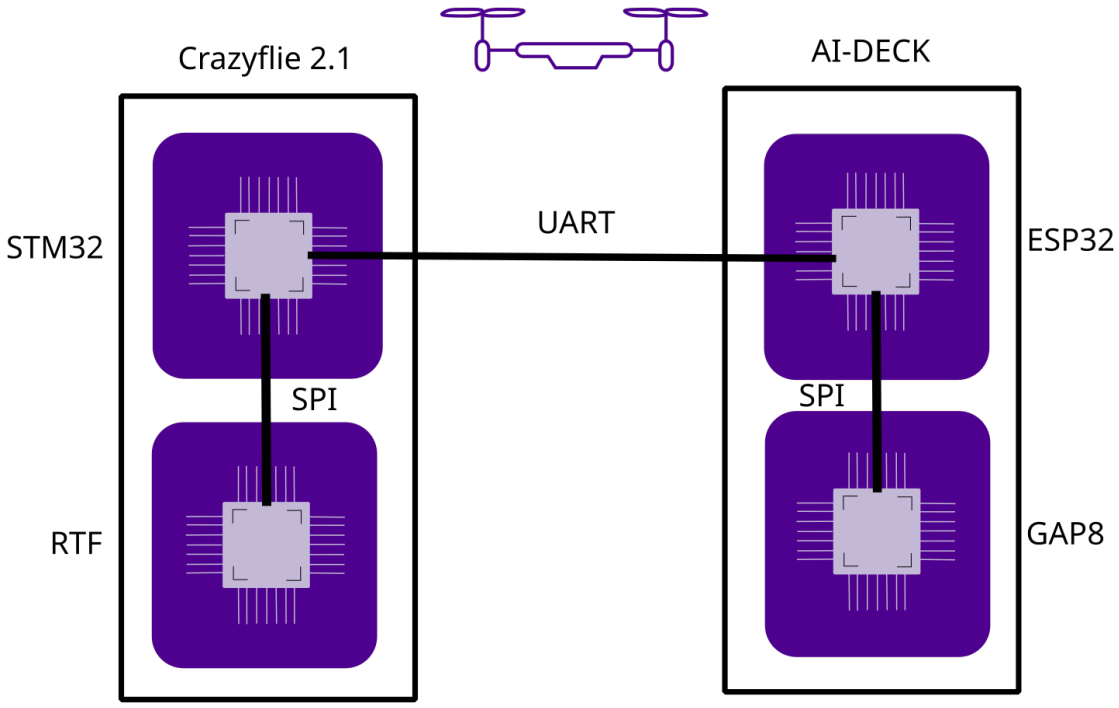

Figure 4.1 *Microcontrollers on Crazyflie 2.1 and AI-deck*

4.2 Crazyflie 2.1

The Crazyflie 2.1 is a 27-gram quadcopter that has a flight time of 7 minutes that the battery on the drone allows with a recommended additional payload of 15 grams. There are many peripherals put into Crazyflie 2.1 as well. There are three different accelerometers available combined with high-precision pressure sensors for evaluating flight state. The drone also has a radio that has the possibility of low-energy Bluetooth. There is support for multiple expansion decks through pin grid controllers that do not require any soldering. Furthermore, the drone has two different methods that can be used to program it. First, there is a USB port that can be used to program the microcontrollers on the drone, although the main function of the USB port is to charge the battery of the drone while it is docked. The method suggested by Bitcraze for programming the microcontrollers on the drone is to use the radio that is embedded into the drone.

The microcontrollers packed into the Crazyflie 2.1 are the NRF and STM32. The NRF microcontroller handles radio communication, Low energy Bluetooth, and power management such as booting up the STM32. The STM32 is the main microcontroller that has the most firmware running on it.

The STM32 microcontroller specifications essential to the proof of concept are listed in Table 4.1. It has a frequency of 168 MHZ which provides quite a sufficient amount of computing power. The bottleneck with running PoCL-R on the STM32

is the total ram that is available on the microcontroller and most of which is already used by the firmware created by Bitcraze. The STM32 also has flash memory to store the application when the drone is offline.

STM32	
Frequency	168 MHz
Ram	192 kb
Ram available	14600 bytes
Flash	1 Mb

Table 4.1 *STM32 specifications*

The STM32 is also responsible for the operation of the drone and the expansion decks while it also runs the code that maintains flight and engine control. All of the peripherals except the radio are connected to the STM32 such as gyroscopes and the expansion pin grid for UART communication.

STM32 has the hardware controls for the motors and it has fine-grained flight control software inside the firmware. This flight control could be preprogrammed to do what the user wants but it can also receive commands during the flight from the communication channels in use. The Crazyflie 2.1 is able to receive communication packets through the radio and it also has the possibility to forward them to different microcontrollers on the drone and to the expansion decks. The STM32 microcontroller is also capable of sending radio messages through NRF's radio and it is sending console messages to the cfclient console on the host computer. Furthermore, STM32 also communicates with all the other expansion decks through the pin grid UART and is able to detect and run them automatically.

4.3 AI-deck 1.1

In order to get the PoCL-R to fit the memory of the drone, the AI-deck was required. The AI-deck also provides the drone platform with WiFi which was required to enable offloading computation on the drone. The AI-deck expansion deck for the Crazyflie 2.1 provides more than enough memory and computing power to the setup.

The AI-deck has JTAG (JTAG Technologies 2022; IEEE 2013) pins for each of the microcontrollers on the board, and programming through them is a faster way to flash binaries onto the HyperFlash. The flashing over radio provided by Bitcraze is a magnitude slower way of flashing the drone albeit the radio flashing is a simple method.

The deck contains 512 Mb of HyperFlash (Infineon 2022) to house the binary of the proof of concept while the drone is offline. The HyperFlash should be able to contain a filesystem on the drone but firmware issues caused it to be unusable and

the support for filesystem and caching binaries were removed from PoCL-R for the time being.

The deck also contains 64 Mb of HyperRAM (Winbond 2022) which is not in use because the memory is not directly addressable from the GAP8 processor. The GAP8 ultra-low power processor provides more than enough computing power and the essential detail is the fact that the GAP8 processor contains enough ram to fit the entire runtime memory usage of our runtime library and demo application.

The AI-deck contains two different microcontrollers, the ESP32 (ESPRESSIF 2022) which is inside the NINA-W102 system on a chip, and the GAP8 (GreenWaves Technologies 2022) ultra-low-power processor which is connected through SPI to the ESP32.

The last important component on the AI-deck is the Himax HM01B0 320x320 pixel camera which is used in the proof of concept to capture images to compute upon.

4.3.1 GAP8

GAP8 is a nine-core processor that is designed to run ultra-low-power computing on edge devices. This accounts for an ideal processor to have on the Crazyflie 2.1 since it offers a substantial amount of processing power with low energy consumption.

The GAP8 offers easy programmability through all the cores being identical RISC-V cores. However, there is a core designated as the Fabric controller that runs on a higher frequency compared to the cluster cores as can be seen in Table 4.2. In addition, the fabric controller has its own L1 cache memory. The fabric controller is an always-enabled core that handles all operations on the peripherals with the capacity to enable or disable the cluster as well. In the example of offloading computing, the cluster provided by the GAP8 is not in use, and instead, a remote server is used for the computation.

All of the RISC-V cores have direct memory addressing access to the ram inside the GAP8. The processor has 512 kB of ram that is enough to contain the runtime library with PoCL-R. Another factor that affects the PoCL-R binary is the fact that the RISC-V cores in the processors do not support floats, thus the float support of PoCL-R is removed to accommodate this.

The GAP8 represents state-of-the-art ultra-low-power processing and acts as an excellent benchmarking comparison for the energy efficiency of offloading computation. However, there will be a new benchmarking target in the future with the forthcoming GAP9 (GreenWaves Technologies 2022) processor which provides all the strong points of GAP8 while boosting energy efficiency to a new level. Furthermore, it brings way more processing power into the mix while adding support for floats and increasing memory capacity and clock frequencies. The GAP9 specifications

showcase a very energy-efficient processor which will be an interesting processor to benchmark against.

GAP8	
Fabric controller Frequency	250 MHz
Cluster Frequency	175 MHz
L1	80 kb
Ram	512 kb
Flash	-

Table 4.2 *GAP8 specifications*

4.3.2 NINA-W102

The second system on a chip on the AI-deck is the NINA-W102 which contains the ESP32 processor for ultra-low-power WiFi. In this thesis, the ESP32 is used to connect the drone to the internet and handle socket API calls between the drone and the server. The ESP32 handles forwarding the internet packets to the rest of the drone through different peripherals. SPI is used to communicate with the GAP8 processor on the AI-deck and UART to communicate with the STM32 on the Crazyflie 2.1 itself.

The NINA-W102 chip supports WiFi 4 with IEEE 802.11n standard with a speed of 25 Mb per second. Using a 2.4 GHz frequency this WiFi connection should be quite stable and the specification states that the range should be up to 400 meters. In reality, the connection has troubles with a connection that has a range of a single meter, and communicating with the access point through a wall and with a distance of 3 meters proved an impossibility.

4.4 Operating systems on the microcontrollers

The microcontrollers available for this demo can not handle anything the size of Linux or Windows which run with the size in megabytes. Thus all of the microcontrollers on the Crazyflie 2.1 and AI-deck have separate instances of the FreeRTOS operating system running on them which has the memory usage in order of kilobytes.

In addition, standard operating systems are not as suitable for embedded devices as real-time operating systems. Real-time operating systems have the ability to prioritize tasks so that critical tasks are always run over less meaningful tasks. Another advantage of real-time operating systems is the fact that they are small so they can be fitted easily to embedded devices as an operating system.

The small kernel provided by FreeRTOS does not have support for many of the properties that standard operating systems have. For example, FreeRTOS is made

aware of the available heap through the linker file during compilation, whereas a standard operating system could query it during installation. Furthermore, memory protection is missing from the FreeRTOS operating system so if bad memory accesses happen then the runtime is likely to crash. Memory leaks overall in FreeRTOS systems are fatal over a longer period of time since there are no protective subsystems available and nothing is going to clean the memory like in a standard operating system.

FreeRTOS uses tasks to run multiple operations concurrently. This is not true parallelism in the case of the GAP8 though and the tasks are only given processing time slices by the FreeRTOS scheduler. Parallelism is plausible in the scenario where the platform running the FreeRTOS would have identical cores that share the same memory. The ESP32 is using true parallelism as the cores on it are identical but this is not possible on the GAP8 as the fabric controller core has its own cache memory. Hardware parallelism is on the other hand achievable by offloading computation onto the cluster.

Memory allocation for tasks used by FreeRTOS is manually defined. Tasks have their stacks allocated from the heap and thus giving the tasks too little memory will cause overwriting of memory to occur killing the runtime. To counteract this issue the tasks were given extra memory in the implementation so that they do not overwrite memory. While it is ideal that a flat amount of memory is given to the FreeRTOS tasks that should exactly fit their needs, the fact that when tasks are swapped out of execution the context of the task is put into the stack as well which consumes additional memory. This caused a lot of hard-to-define bugs to occur when swapping the tasks out of execution caused memory to be written onto the heap on an area which was then overwritten by another task.

The FreeRTOS tasks communicate with each other through direct-to-task messages or queues that can be read by other tasks. On Crazyflie 2.1, the FreeRTOS queues are in use and the communication between chips is using a separate FreeRTOS task and queue on each chip to read and redirect messages around the FreeRTOS tasks of that specific microcontroller.

The FreeRTOS kernel is also extendable with many additional libraries that provide additional functionality for the operating system. Depending on the microcontroller one can add support for example for internet connectivity, input-output enhancements, or a filesystem. In this thesis, the interest lies in the FreeRTOS-Plus-POSIX library, since the plain FreeRTOS kernel does not support Pthread API.

5 Porting the runtime library onto the drone

This chapter will explain what steps were taken in order to make the PoCL-R runtime library fit and work on the nano drone. First, the discussion starts with what was done to enable the PoCL-R binary to fit in the memory of the drone. Then the extensions built for the GAP8 firmware to support the PoCL-R functionality are discussed. The chapter continues with an explanation of how the communication between the drone and the server is enabled and concludes with a discussion on how flight control is enabled.

5.1 Creating minimal PoCL-R library

In order to fit the runtime onto the limited memory of the nano drone the memory footprint of the runtime had to be minimized. To reduce the size of the final binary the PoCL-R was compiled with size optimization (-Os). Furthermore, the -ffunction-sections -fdata-sections were used to reduce the memory footprint of the final binary with the cost that the intermediate binary was larger and slower to compile.

In addition to flags to reduce the size of the binary, some of the properties that existed were removed from the runtime library. Removed features included floats that were not supported by the GAP8 processor. Furthermore, the support for a filesystem was removed because the version of the firmware of the GAP8 used for the runtime on the AI-deck did not have functioning filesystem support.

In this proof of concept, the remote server where the computation was offloaded was also used to compile and run the kernels. This allows turning off modules such as the LLVM, PoCLCC, ICD, HSA, and Loadable drivers which are not required by the proof of concept. This minimizes the runtime as only the online compilation provided by the LLVM module would consume gigabytes of memory which the drone does not have available. Instead, the kernel is stored in a string buffer on the drone that is sent to the remote server to be compiled when required.

The only optimization that was not used to reduce the binary size was the MinSizeRel flag on the compiler. Enabling that could have improved the memory footprint but all the requirements to do so were not available on the current toolchain used to build the binary, thus the binary was built with the release flag instead.

On top of minimizing the binary size, the runtime memory consumption was reduced as well. PoCL has structs that are consuming memory with statically allocated arrays according to the worst-case requirement of dynamically consumed memory. Reducing the size of these arrays to only fit the practical needs of the minimized runtime was a method to reduce the memory footprint of the runtime.

5.2 Extending GAP8 firmware to port PoCL-R

Once the PoCL-R is compiled into a library, the functionality required from it has to be linked into the firmware of GAP8 to form a single binary. There are some requirements on the PoCL-R that could not be removed and had to be implemented instead as extensions to the GAP8 firmware. This chapter will discuss how the requirement for Pthreads was handled along with the function calls that were required but missing.

The PoCL-R requires Pthreads to function. As Pthread is quite common POSIX property there is already built-in support for it in the FreeRTOS ecosystem. To enable Pthreads the FreeRTOS-Plus-POSIX extension is added on top of the basic FreeRTOS kernel. This extension provides the POSIX threading API that is used to redirect API calls that use Threading API to use the tasks provided by the FreeRTOS kernel. In addition, the extension provides mutexes and conditional locks that are also used by the PoCL-R runtime. The Pthread requirements of PoCL-R are fulfilled after adding the Pthread extension and enabling static allocation of mutexes and locks in the extension's configuration files.

Another missing feature that the PoCL-R requires to function was environmental variables. The environment variables were built into the current implementation of the program as a static string array. This array is a mock for the real environment variables and it only contains a couple of variables that were necessary to run the PoCL-R. Through these variables, the IP of the server is hardcoded into the runtime, and some other configuration values for the runtime are defined such as the fact that remote devices are the ones that should be used combined with the fact that local compiling is not supported.

There were many other functions that were missing, such as the functions for printing and atomic operations. The functionality that they implemented was redirected to the functionality provided by the firmware of GAP8 such as the BSP and FreeRTOS kernel. In the scenario of the atomic operations, there were atomic operations available on the firmware and thus redirecting was all that was required to achieve functionality. The timestamps functionality was also missing from the firmware yet it was required by PoCL-R. However, the AI-deck has a real-time clock embedded in it and the BSP provides API for its functionality. Thus the real-time clock was initialized and used to create the timestamps. In the scenario where there was no direct support for functionality on the GAP8 firmware, the firmware was extended with the c functionality from Newlib (sourceware 2022).

5.3 Enabling communication

The PoCL-R uses the socket API to communicate with the server over the internet. However, the GAP8 has no means to access the internet and thus no support for the socket API. The ESP32 on the AI-deck does however have a WiFi transmitter and support for the socket API. Thus the issue was how to get the socket API calls made on the GAP8 processor to transfer over and use the functionality provided by the ESP32.

Enabling this functionality is made easier by the fact that the drone environment already has a communication infrastructure in place for communication between microcontrollers. In the proof of concept, the task that remained was to pack the socket API calls onto the CPX packets and forward them to the correct functions on the correct microcontrollers.

To redirect the socket API calls from the GAP8 to ESP32 a redirecting socket API was created on the GAP8 side. The function of this API is to pack the information of the socket API calls onto CPX packets and forward them to the correct function on the ESP32 side and once a response is coming from the ESP32 forward it back to the original caller function. To the GAP8 it appears that it is using the standard socket API while in truth the API calls are forwarded onto ESP32.

The CPX system is not without its own problems. The CPX packet has a maximum size of 1022 bytes which implies that larger buffers can not be transmitted with a single packet and have to be split into multiple packets to be forwarded. In this thesis, this limitation was avoided by limiting all of the packets that were sent over CPX to be smaller than 1022 bytes. Since the only API calls, that could be over the size limit were the read and write calls of the socket API, Limiting their size was trivial as they already had support for limiting the transfer sizes to the desired amount.

5.4 Flight control

After enabling the socket API the communication between the server and the drone functions. This allows the drone to run OpenCL kernels on the remote server and receive the responses back from it. However, there is no functionality in place to control the flight of the drone.

The CPX packets can already be sent from any microcontroller on the drone to any other microcontroller on the drone. However, these packets are not able to control the flight of the drone on the STM32 microcontroller. The CRTP packets are the ones used for flight control and the firmware for the drones has no way of sending them between microcontrollers.

The solution that was used was to wrap the CRTP packets inside CPX packets

on the GAP8. This allows the routing of the CRTP packets between GAP8 and the STM32 microcontrollers. Furthermore, a capability for the STM32 firmware was built that allows unwrapping the CRTP packet from the CPX packet. After unwrapping the CPX packet the STM32 redirects the data in the CRTP packet to the motion commander that resides on the STM32. The motion commander then handles the flight control according to the data inside the CRTP packet.

After providing support for all the functions required by the PoCL-R and enabling the communication between the flight control of STM32 and GAP8 a functioning runtime library has been created.

5.5 Walkthrough of the runtime

In this section, the details of the runtime that is executed on the nano drone in the offloading computation proof of concept are described. The explanation of the runtime offloading computing is followed by going through the locally computed runtime which is used as a comparison for the PoCL-R runtime. First, though a look is taken at the booting of the Crazyflie since it is identical between the runtimes. Starting with the Crazyflies microcontrollers a look will also be taken at the microcontrollers on the AI-deck and how the runtime executes there.

When the drone boots there is a lot of initialization that is done. The STM32 is booted by the NRF microcontroller and it then initializes itself for operation by enabling communication and flight control. Along the initialization the drone checks if there are expansion decks connected through the expansion pin grid.

5.5.1 ESP32

When the ESP32 on the AI-deck boots up, the first action it does is to enable the Com task that allows it to talk to the other microcontrollers. Com task is followed by the CPX task which is activated so that the packets can be sent and received.

After the internal communication is initialized the ESP32 starts to look for a WiFi connection. In the computation offloading example the WiFi access point that is searched for is static and the credentials for it are hardcoded into the binary of the runtime library.

Once the connection is established to the access point then a CPX packet containing an acknowledgment of the event is sent to the GAP8 microcontroller. After the packet is sent the ESP32 enters a loop where it listens to CPX packets arriving at the CPX function on the ESP32 and forwards them according to their metadata. Packets that are sent to the WiFi control loop by GAP8 are unwrapped and forwarded to their proper socket API calls that exist on the ESP32.

5.5.2 GAP8

The initialization of the GAP8 starts with the launching of the Com task as well as it enables the microcontroller to receive messages from the other microcontrollers. In the case of GAP8 however, The GAP8 enters an idle state after the Com and CPX tasks are initialized. The GAP8 remains in the idle state until it receives a packet telling it that the WiFi connection is established.

After the WiFi connection acknowledgment is received the GAP8 sends the motion commander task running on the STM32 microcontroller on the Crazyflie 2.1 a takeoff command through CPX. Afterward, the GAP8 starts the main PoCL-R runtime task. The task uses the standard OpenCL API to establish a standard OpenCL runtime while in the background the PoCL-R backend redirects the API calls to use the remote server through the ESP32.

Running the clGetPlatformIDs command causes the PoCL-R runtime pipeline to try and enable a socket connection through which available servers are scanned. In the proof of concept, the server IP was hard coded into the runtime of GAP8 and that server was then searched and connected to through the socket API. All the socket API calls used were redirected to the ESP32 through CPX packets that wrap the function call data. These packets are then sent to the ESP32 microcontroller that unwraps the packets and forwards the data to the socket API available on the ESP32.

Once the connection is established the PoCL-R establishes the backend functionality on the GAP8 that it requires to run the OpenCL on the remote server. This includes creating 2 writer threads, 2 reader threads, and a PoCL remote driver thread. As there is a lack of more fine-grained control in the GAP8 all of these threads are initialized with the same amount of stack which causes some memory to be wasted.

When the backend is initialized by PoCL-R, the standard OpenCL function calls are redirected to the server through the socket connection. Querying the devices with the clGetDeviceIDs function call causes the runtime to forward the query to the remote server whose reply is given as the answer. Then the runtime uses the clCreateContext function call and creates the context for the device that was queried and follows that by creating a command queue for that device with clCreateCommandQueue.

The kernel that is intended to be used by the runtime is stored as a string buffer in the binary on the drone. The string buffer is at this point sent to the server with the clCreateProgramWithSource function call. The server then compiles the string buffer into the kernel that can be run on the server using the clBuildProgram and clCreateKernel. The clBuildProgram also sends the binary back to the drone but in

the proof of concept, it is thrown away once it is received.

As the kernel is now compiled and ready to run on the server, the next step is to create the buffers that are required to provide the server with the data and to receive the resulting data back from the server. The last initialization step is to set the kernel arguments to be the buffers that were created.

The next step in the runtime is to enter the main computation loop. In this loop, the computation tasks are done according to the runtime requirements. In the PoCL-R runtime, there are three steps occurring. First, the data is sent to the server with the clEnqueueWriteBuffer call which is followed by a clEnequeueNDRangeKernel call to execute the kernels with the data. Finally, the results are queried with clEnqueueReadBuffer from the server back to the drone.

The result that came from the server is then used to calculate the amount of height and yaw the drone should adjust and the amount desired is then wrapped into an STM32's motion commander command. The motion commander command is then wrapped into a CRTP packet which is wrapped into a CPX packet and that is then sent to the STM32. The main computation loop is then repeated until a terminating condition occurs with 2 seconds of wait time between each iteration so that the drone has time to orient itself according to the command given to the motion commander on the STM32.

5.6 Locally computing runtime

The local comparison runtime for the PoCL-R runtime is very similar to it. The key difference is removing the OpenCL API calls and replacing them with a single function call that computes the same kernel on the GAP8 microcontroller.

In this scenario, the ESP32 has only a single task and that is to act as a communication medium between the STM32 motion commander and the GAP8 microcontroller. The GAP8 microcontroller on the other hand is able to boot directly without waiting for the WiFi connection acknowledgment from the ESP32 and it starts the runtime straight from bootup.

The GAP8 starts the execution by sending the liftoff command through CPX to the STM32 and then proceeding to the main computation loop. The first step in the loop is again sending the data. However, in the local runtime, the buffer containing the data is given as a parameter to the local function that computes the result. The corrections to the height and yaw of the drone are then calculated from the result of the local function and forwarded to the motion commander on the STM32.

This loop is then iterated until a terminating condition occurs with the 2-second delay and then the landing command is issued to the motion commander on STM32.

6 Evaluation

In this chapter, The details of the kernels used to measure the metrics of the proof of concept runtime are explained. This is followed by an explanation of how the measurements were taken in the main computation loop and what results were obtained. These results are then discussed in detail and the discussion is followed by chapters about the future and related work.

6.1 Kernels used to measure performance

The PoCL-R runtime was measured by placing different kernels into the main computation loop on the GAP8 microprocessor. For the computation offloading proof of concept, a kernel was run on the server that computed the brightest pixel on an image provided by the AI-deck camera. Another kernel run was the MinOverhead kernel which was used to measure the minimal latency overhead possible that is caused by PoCL-R. These kernels were then compared to running the kernel that computes the brightest pixel of an image as a local function. Next, the details of the kernels are described, starting with the SeekBrightestPixel kernel and followed by the MinOverhead kernel.

6.1.1 SeekBrightestPixel kernel

The SeekBrightestPixel kernel described in Program 6.1 computes the brightest pixel out of a buffer provided. The kernel is simple as it only goes through the given data once and returns the coordinates of the highest number in the buffer as a result. The kernel is run using only a single thread for easy comparison with the locally executed version of the kernel.

As is visible in the Program 6.1 the kernel is plain C language that is stored as a char buffer on the drone. This requires a macro to be defined that translates the variable values inside that string to proper values. The XSTR macro replaces the given parameter with the correct value associated with it in the code file. The resulting char buffer is then forwarded to the server using OpenCL API calls during runtime. The server that received the string buffer then compiles the kernel in the string buffer to be able to run that on the hardware of the server.

Before the kernel on the server is run, the server receives two buffers that are required as parameters for the kernel. The first data buffer is for the data that arrives on the server. The data buffer is of the constant size of 79 056 bytes which is derived from the dimensions of the camera on the drone. The second coords buffer is for sending back the resulting coordinates to the drone.

After receiving the buffers and execution order the kernel is run. The kernel initializes variables it requires for looping through the buffer and to calculate the correct coordinates. The buffer contains a concatenated two-dimensional buffer so the modulo operation derives the correct coordinates out of the buffer. These resulting coordinates are then put into the coords array to await the moment when the answer is requested to be sent back by the drone.

When evaluating the PoCL-R performance the kernel is compared to running the same kernel as a compiled local function on the drone.

```
1 static const char *kernelsrc = ""
2 "kernel void SeekBrightestPixel(global const uchar* data,"
3 "                               global int* coords) {\n"
4 "  int id = get_global_id(0);\n"
5 "  int max_x = 0; int max_y = 0; uchar max_val = 0;\n"
6 "  for (int i = 0; i < " XSTR(CAM_WIDTH) "*" XSTR(CAM_HEIGHT) ""
7 "  ; ++i) {\n"
8 "    uchar val = data[i];\n"
9 "    if (val > max_val) {\n"
10 "      max_x = i % " XSTR(CAM_WIDTH) ";\n"
11 "      max_y = i / " XSTR(CAM_WIDTH) ";\n"
12 "      max_val = val;\n"
13 "    }\n"
14 "  }\n"
15 "  coords[0] = max_x;\n"
16 "  coords[1] = max_y;\n"
17 "}";
```

Program 6.1 *SeekBrightestPixel kernel*

6.1.2 MinOverhead kernel

The MinOverhead kernel aims to provide as little work for the PoCL-R runtime while still using all the parts required for a proper runtime of the system. In the MinorOverhead kernel 6.2 the kernel only increments the data given by 1. The kernel exists to test out the minimum latency overhead caused by running the PoCL-R on the drone. The execution is similar to the execution of SeekBrightestPixel, only the amount of data sent and the complexity of the computation differ.

```
1 static const char *minkernelsrc = ""
2 "kernel void MinimalOverhead(global const uchar* data,"
3 "                            global int* coords) {\n"
4 "  coords[0] = data[0] + 1;\n"
5 "}";
```

Program 6.2 *MinOverhead kernel*

6.2 Measuring the PoCL-R metrics

The SeekBrightestPixel and MinOverhead kernels are used in the main computation loop on separate runtime iterations to measure the baseline performance of the PoCL-R runtime created for the Crazyflie 2.1 nano drone. The main computation loop was run with 100 iterations to measure the average performance of the runtime as well. In addition, the flight was disabled as it does not affect the memory consumption or latency of the implementation.

In the measured runtime of the program, the server was a computer that was connected to the same access point that the runtime tries to connect to at the initialization phase. The OpenCL device on the server was an Intel iRISx integrated graphics die (Intel 2022).

The performance metrics were measured in the main computation loop of the runtime. In the main computation loop of the SeekBrightestPixel, the latency metric measurement starts after the image is taken with the camera. At this point, the next command to occur is the clEnqueueWriteBuffer which sends the image to the server. Sending the buffer to the server is followed with the clEnqueueNDRangeKernel command that tells the server to compute the kernel and the result is queried back to the drone with clEnqeueueReadBuffer. After receiving the result from the server time measurement is concluded for that iteration.

The latency thus includes the overhead caused by tunneling the socket API calls to the ESP32 microcontroller through the SPI with CPX packets and receiving them back as well. The latency metric also includes the time the PoCL-R takes internally to process the commands. Latency also includes the time it takes for the data to be transferred to the server and back from the ESP32 microcontroller, and the time it takes to compute the kernel on the server.

The memory consumption was measured on the values of the SeekBrightestPixel kernel runtime and that should represent a realistic memory consumption as it uses the whole OpenCL program pipeline to run the computation. The key difference to the minimum possible memory consumption of the runtime library is the buffer allocated for the image as it is program-specific memory usage.

6.3 Metrics

In this section, a look is taken at the metrics of the PoCL-R binary and the runtime on the GAP8 microcontroller. The other microcontrollers are omitted in memory metrics since the GAP8 holds the runtime binaries and the functionality built to support the runtime library of the PoCL-R. The runtime library is required in other contexts where the PoCL-R could be used while the other microcontrollers can be changed. Also, the WiFi connection details vary depending on the microcontroller

that is used to achieve it. How the socket API is implemented in the background is not relevant to this thesis and thus the way it is handled on the GAP8 is the meaningful implementation. On top of that, the implementation could be running on any sort of other microcontrollers, and the STM32 and flight control work only as an example.

In this section, the memory usage metrics of the PoCL-R on the GAP8 are examined first followed by latency measurements and the last topic to be handled is energy consumption.

6.3.1 Memory consumption

The memory usage of the PoCL-R implementation can be divided into two categories. The static amount used by the binary and the dynamic memory usage of heap and stack by the program runtime. Furthermore, the memory consumption of the runtime library is similar to the SeekBrightestPixel example runtime of the PoCL-R.

The size of the binary that is able to run the PoCL-R on the drone is 184 352 bytes as can be seen in Table 6.1. With the drone having 512 000 bytes of ram available the binary with a size of 184 352 bytes can be put there easily.

Stack is used by the FreeRTOS tasks running on the drone. The 5 Pthreads that PoCL-R uses to run the communication with the server and to operate are implemented as FreeRTOS tasks. They use 3584 bytes of ram each totaling 17 920 bytes. Further stack is used by the task running PoCL-R on the drone which uses 12 800 bytes of memory and the Com and CPX tasks that are used to communicate with the other microcontrollers that end up using 1024 bytes of memory each. Thus the total stack consumption is 32 768.

The runtime allocates memory dynamically from the heap and the maximum amount of heap that the SeekBrightestPixel runtime uses at a single point in time is 103 636 bytes. However, the memory consumption of the image buffer is 79 056 bytes which is only required in the SeekBrightestPixel demo application and thus the runtime library itself only consumes 24 580 bytes of the heap.

6.3.2 Latency

For the latency metric, there are several different runtimes whose latencies were measured. The measurements taken start with the MinOverhead kernel and proceed to the program using the SeekBrightestPixel kernel and finish with the latency of the locally computed kernel. In addition, the results of pinging the drone over the internet are discussed for comparison with the latency of the runtime.

	Application	Runtime library
Component	memory usage in bytes	
Binary	184 352	
Stack	32 768	
Heap	103 636	24 580
Total ram consumption	320 756	241 700
Total Ram	512 000	

Table 6.1 *Memory consumption*

Most of the latency on the kernels run on the remote server was from the communication between the ESP32 and the server. The computation time required for the kernels was minimal as neither of the kernels is computationally intensive. The Intel iRIS integrated graphics (Intel 2022) used in the server has a 1.3 GHz frequency thus if the computation takes 21 ms with the GAP8 running at 250 MHz it should be less than 5 ms on the server. The MinOverhead kernel only increments a number on the server side and this should take less than 1ms on each iteration.

The MinOverhead kernel had an average runtime of 492 ms when running 100 iterations of the kernel. On top of that, the worst runtime from a single iteration was 10 357 ms while the fastest runtime was only 79 ms. The differences between the runtime speeds with the MinOverhead kernel have a huge variance as can be seen in Table 6.2.

The great discrepancy continued in the runtime latencies of the SeekBrightest-Pixel kernel. The average runtime of the kernel was 6691 ms while the best iteration had a runtime of only 608 ms and the worst iteration took 71 206 ms.

These latency figures are quite high compared to the runtime of the locally computed version of the SeekBrightestPixel kernel which took only 21 ms to compute the result.

Kernel	Average runtime	Fastest	Slowest	Total
MinOverhead	492 ms	79 ms	10357 ms	49277 ms
SeekBrightest-Pixel	6691 ms	608 ms	71206 ms	669087 ms
Local	21 ms	21 ms	21 ms	2100 ms

Table 6.2 *Table of runtime latency metrics*

In addition to the runtime latencies, the latency of the network was measured by pinging the ESP32 microcontroller on the drone. If a ping command was used with parameters that set the packet size to 1022 bytes, sending interval to one second,

and iterations to 100, then the measured latency of the ESP32 averaged to 67.9 ms with the best iteration having a latency of 4.2 ms, and the slowest 178.3 ms.

The surprising aspect in latency metrics appeared when the ping command was modified to be more frequent. As can be seen in Table 6.3 if a ping was sent every 2 ms, then the average latency was only 3.2 ms with the best iteration having 2.2 ms of latency and the slowest having 20.6 ms. The reason for this could be in the energy-saving states of the ESP32 microcontroller. If the WiFi goes to sleep and has to wake up for the communication to happen then the latency might be higher and it might be that as the ping frequency increases the ESP32 does not go to a sleep state and has a fast and stable response time.

Ping interval	Average	Fastest	Slowest
1s	67.9 ms	4.2 ms	178.3 ms
0.002s	3.2 ms	2.2 ms	20.6 ms

Table 6.3 *Ping latencies*

6.3.3 Battery voltage levels on remote versus local execution

The last metric that is discussed is the battery voltage levels of the drone when the SeekBrightestPixel kernel was run using PoCL-R and how the voltage level compares to using the local version of the SeekBrightestPixel kernel. The battery voltage level runtimes were identical to the demo program runtimes.

The PoCL-R starts from a battery voltage of ≈ 4.191 and over the runtime it drops by ≈ 0.269 volts leaving the battery with a voltage level of ≈ 3.921 over a runtime of 664 seconds. The local runtime starts from a voltage level of ≈ 4.185 and over the runtime it drops by ≈ 0.095 leaving the battery voltage level to ≈ 4.090 with a runtime of 209 seconds. This comparison can be seen in the Figure 6.1 and Table 6.4.

Implementation	Starting voltage	Ending voltage	Time	Voltage drop
PoCL-R	4.191 V	3.921 V	664 s	0.269 V
Local	4.185 V	4.090 V	209 s	0.095 V

Table 6.4 *Battery voltage levels*

6.4 Discussion

This section discusses the implications that the metrics have for the PoCL-R runtime and its reasonability as a framework for offloading computation. In the first subsection, the memory footprint of the runtime is discussed and that is followed

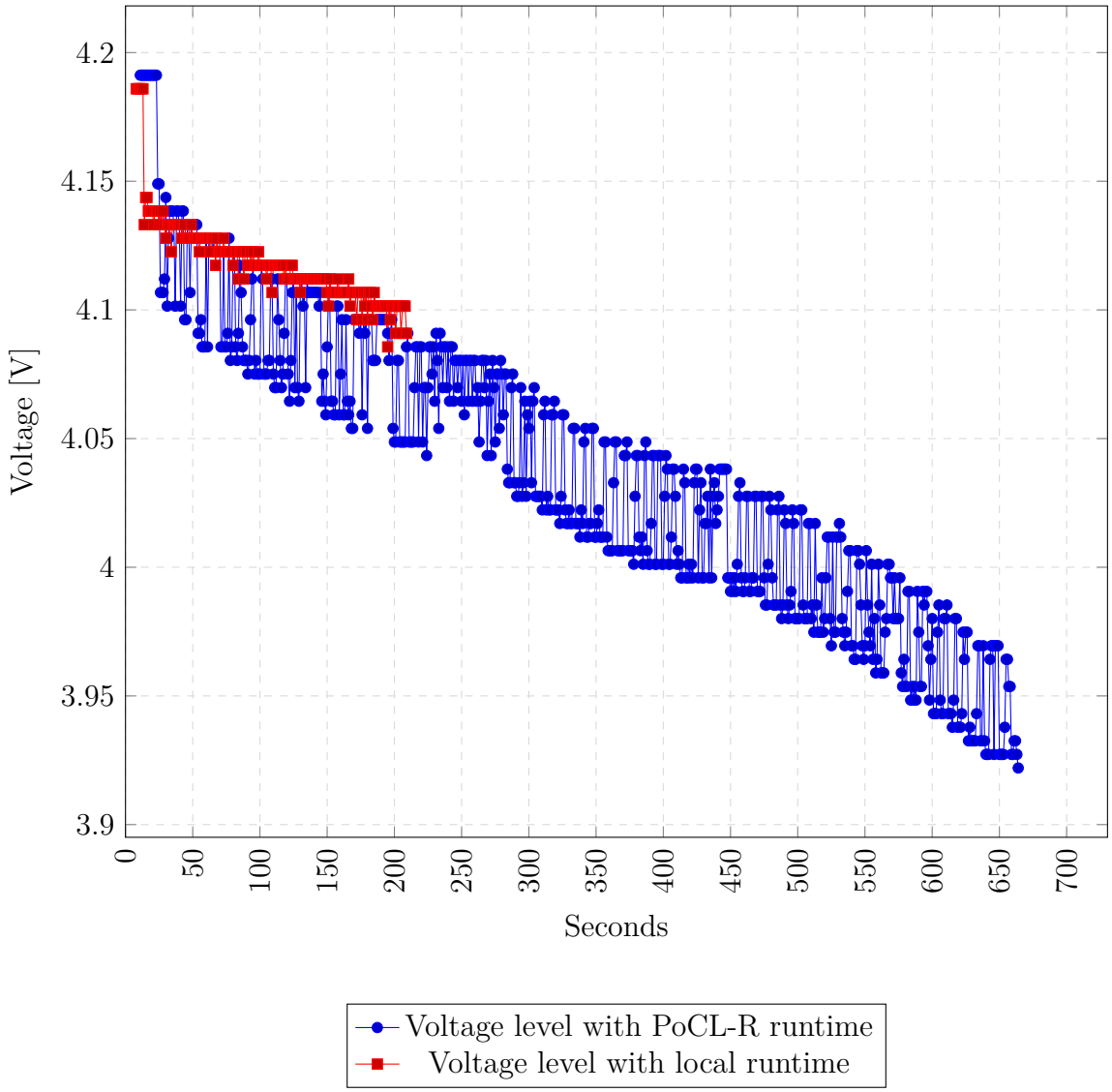

Figure 6.1 *Battery voltage levels when running applications on the drone*

by a subsection discussing the latency of the runtime and then the last subsection tackles the energy consumption metrics.

6.4.1 Memory footprint

The runtime library that enables running the main computation loop has a memory footprint of 241 700 bytes. This is only 45.2% of the ram that is available on the AI-deck, leaving a decent chunk of memory available for the program.

The memory footprint of the entire runtime with SeekBrightestPixel stands at 320 756 bytes as the main computation loop with the SeekBrightestPixel kernel requires 79 056 bytes of memory for the buffer that contains the image taken by the camera. This accounts for 76.2% of the 103636 bytes of heap that is used.

Furthermore, the image buffer accounts for 24.6% of the total memory usage of the runtime, and even this amount could be greatly reduced by taking the image in smaller pieces and sending slices of it instead of the whole picture at once.

The stack memory footprint could be directly reduced by creating a slightly more fine-grained control over the stack sizes of the Pthreads created for PoCL-R. In the current implementation, the same amount of memory is allocated for every thread and the thread that requires the most memory is the memory amount that has to be allocated. This ends up consuming extra memory since the threads that require less memory end up with additional memory that is not used.

The binary has a chance that it could be made smaller by implementing the required functionality so that the MinSizeRel compiler flag would work. Nevertheless, the binary size of 184 352 bytes is reasonable as it takes less than half of the ram available. Furthermore, it would be possible to not have the entire binary drawn into the ram of the processor.

Combining the binary size with the additional memory consumed by the current runtime, the total memory size works as a nice baseline for the amount of memory required for enabling offloading computation on a device capable of WiFi connection.

6.4.2 Latency

The latencies of the main computation loops are most likely higher than they could be as the variance in the latencies measured is high. With the MinOverhead kernel, a tolerable fastest iteration of 79 ms was achieved, however with an average iteration time of 492 ms. 492 ms is an amount of time that makes real-time applications unreasonable and the fact that the slowest iteration took 10 357 ms causes the offloading to be unusable for any real-time application.

In the case of the SeekBrightestPixel kernel, the 79 056 byte image buffer has to be sent over 78 serial write commands that will take at least 5296 ms to complete if the latency average was 67.9 ms. The SeekBrightestPixel kernel achieves an average runtime of 6691 ms which contains clEnqueueWriteBuffer that sends the image buffer, the EnqueueNDRangeKernel message to compute the data, and the clEnqeueueReadBuffer to read the data totaling 80 commands to be sent to the server. Sending these 80 commands over 6691 ms is much over the timeframe that can be allowed for real-time flight control. However, the best iteration that the SeekBrightestPixel kernel had was only 608 ms which is already a far more reasonable number and averages to 7.6 ms of latency for each message. The likely cause for this variation is in the ESP32 WiFi controller chip energy saving states.

This issue can be seen when the drone is pinged from the host computer of the server. Pinging the drone with one-second intervals brings a long average latency of 67.9 ms yet the fastest iteration is only 4.2 ms. The improvement to the latency

when more load is put on the connection is intense. If the interval is brought down to 2 ms between pings the average latency drops to 3.2 ms which would make the offloading of computation feasible.

With the best iteration of 608 ms and 7.6 ms of average latency for each message, the SeekBrightestPixel kernel overhead with the CPX tunneling and the time it takes to compute the data on the server can be estimated to be 3.2 ms, if the time it takes the WiFi to send each message would be calculated with the fastest iteration of 4.2 ms. This is most likely an overestimation of the latency caused by the PoCL-R because it is very unlikely that all of the 80 messages transferred by the WiFi would have been done with the fastest iteration recorded from the ping command, especially with the average latency being that much higher.

6.4.3 Performance

From the battery voltage metrics shown in Figure 6.1, it can be seen that the remote execution of the SeekBrightestPixel kernel causes the battery voltage to lower more than the local execution. With the battery voltage levels, one can give a coarse approximation of the amount of energy used from the 250 mAh Lithium polymer battery on the Crazyflie drone's battery (Bitcraze 2022a). The table of capacity and voltage relationship provided by (AMPOW 2022; Mark B 2022) closely resembles the discharge chart of a similar 250 mAh battery (BPI 2022). Based on this it can be approximated that the local kernel battery voltage of 4.090 equals 85% of the maximum capacity of the battery and the remote battery voltage of 3.921 around 65%. With the battery size of 250 mAh, the local kernel has consumed approximately 37.5 mAh and the remote kernel 87.5 mAh and with this rough approximation, it can be said that the remote implementation consumes more energy.

The situation might not be as bad as it appears though. Running only the PoCL-R with the SeekBrightestPixel kernel for 100 iterations took 669 seconds. When the best iteration that occurred took only 608 ms and 100 iterations consuming that amount of time would take only 60.8 seconds. This means that at least 90.9% of the runtime is spent on waiting for the WiFi controller to manage to transfer the packets.

This amount of wait time causes the current version of the proof of concept to be inefficient. On the other hand, it also means that offloading computation with PoCL-R is not a hopeless endeavor if the WiFi controller could achieve average latencies closer to the best iterations. First, reducing the amount of time spent waiting for messages to be transferred will save energy, since keeping the hardware in a waiting state will consume energy. Secondly, the WiFi communication channel has to be open all the time when it is trying to communicate. Reducing the time spent waiting on the WiFi by 50% would cause the runtime to spend five times

the amount of time on running tasks of the total runtime compared to the current implementation.

Finally, the SeekBrightestPixel example is computationally simple. If the example would be more computationally intensive the increase in computation cost can only be seen on the locally executed version of the runtime as the server is doing the computation for the remote implementation and for that the drone does not have to pay any kind of energy cost. Thus even with the current implementation, a very computationally intensive kernel could be beneficial to offload to the server instead of executing it locally.

6.5 Future Work

The top priority for future work regarding the PoCL-R runtime example is to fix the WiFi controller's latency issues. Reducing the latency of the WiFi to even half of the current status could provide tremendous metrics uplift towards the benefits provided by offloading computing. If the delay could be brought to the level of the fastest computation iterations currently occurring, then the competition between energy efficiency could be in favor of the PoCL-R runtime instead of local computing.

The latency of the runtime could be further reduced by compressing the image and using the cl_pocl_content_size (Michal Babej 2022) extension to reduce the amount of data that is sent over the WiFi.

The next task to improve the drone runtime would be to make it tolerate errors. The possibility to reconnect to the server in case the connection is lost would be vital for using the PoCL-R runtime for compute roaming or anything that has to be successful at once. For example, computing the results for this thesis would have been far faster and easier if every disconnect from the server would not have caused a total loss of the result set.

To enable compute roaming the PoCL-R runtime could be made to also connect to multiple servers to improve the performance of the computing and provide fault tolerance on the server side. The servers should also be made to broadcast that they exist using the service location protocol (Day et al. 1999). Combining service broadcasting servers with a drone that searches for WiFi access points and for available servers in the network through service discovery such as the lightweight service discovery protocol (Lim et al. 2005), would create a true compute roaming scenario.

Furthermore implementing the possibility to run OpenCL kernels locally on the cluster of the GAP8 processor could enable the drone to perform computation more energy-efficiently in some cases. This combined with running the PoCL-R server on the drone and using the GAP8 cluster as the compute device would provide stepping stones for a drone swarm to collectively compute tasks.

6.6 Related work

The AI-deck is used in many other research topics that use computer vision and neural networks (Palossi, Zimmerman, et al. 2022; Palossi, Conti, et al. 2019; Navardi et al. 2022). The goal of these projects is to run a deep neural network or computer vision pipeline locally on the GAP8 processor available on the AI-deck. They aim to implement the vision pipeline on the drone to showcase that the algorithms can be fitted onto the limited memory available on the GAP8 and to achieve autonomous flight. This is similar to the proof of concept shown in this thesis as well, the difference between these works comes from the fact that the other projects aim to provide as energy-efficient algorithms as possible that can be run locally on the drone. Whereas in this thesis the computation is offloaded from the drone to be done on a server which eliminates the need for local energy-efficient algorithms if offloading computation is possible and energy efficient. In addition to the possibility of saving energy, offloading computation allows more resources to be used on a task. This enables more complex algorithms to be used by the resource-restricted device since complex algorithms can be computed faster on a remote device to fit a timeframe while consuming less energy as only the transmission fees have to be paid by the offloading device.

Further related work is occurring on the mobile device sector (Eshratifar and Pedram 2018; Tian et al. 2021). The problems presented are similar to the nano drone environment even though the available resources for computation and networking are on a different scale. Furthermore, the aim of these works is to offload a very specialized dynamic neural network task to the server whereas in this thesis the offloaded data is OpenCL kernels which enables more general task offloading as well.

Another topic where the similarity to the nano drones studied in this thesis can be seen is the topic of autonomous driving of smart cars. There are many different research projects studying the problem of how to create a dynamic mobile network of cars to enable autonomous driving with collaboration (Boukerche and Soto 2020; Jin et al. 2021; Wang et al. 2016). The computing ecosystem for smart autonomous cars is by nature very heterogenous and requires the ability to offload computation. Using PoCL-R in a similar manner as in this thesis many of these problems are already tackled since offloading computation from a nano drone is mobile edge computing.

7 Conclusions

In this thesis, the PoCL-R was ported to the GAP8 microcontroller on a Crazyflie 2.1 nano-drone as a baseline implementation for offloading computation from a nano drone. The motivation for porting PoCL-R came from the possibilities that offloading computation from microcontrollers to heterogeneous servers provides. The baseline implementation provides resource-restricted devices additional capabilities through remote resources and the research can continue to compute roaming on the nano drone.

To port the PoCL-R onto Crazyflie 2.1, the PoCL-R was minimized into the Nano-PoCL, a minimalistic implementation of PoCL-R. The Nano-PoCL implementation was then benchmarked to give the memory requirements for the Nano-PoCL runtime library and the overhead the current implementation causes on latency and energy consumption.

The Nano-PoCL runtime library provides baseline memory requirements for offloading computation that can be improved upon. The current implementation requires 241 700 bytes of ram to fit onto the Crazyflie 2.1 nano-drone. This amount could be finetuned to be even lower for example, by more precise stack allocation for tasks and Pthreads. Nevertheless, it works as an excellent baseline that can be used to run kernels remotely and upon which further improvements to memory footprint can be made.

The latency of Nano-PoCL is difficult to estimate since the WiFi controller has a great variance in latency. Pinging the ESP32 microcontroller with longer intervals has a latency of 67.9 ms which is high for computing kernels in real time. If the program runtime has to send multiple messages in serial the latency compounds quickly, for example, sending the image buffer in the SeekBrightestPixel kernel with 78 serial messages would take 5296 ms which cannot be considered real-time. However, the fastest iteration of running the SeekBrightestPixel only took 608 ms which is far faster than could be achieved with the average latency for 80 messages. Calculating the Nano-PoCL overhead to be 3.2 ms with the fastest iteration of SeekBrightestPixel kernel and 4.2 ms latency of the WiFi controller is possibly an underestimation of the performance of Nano-PoCL. The likelihood of sending all 80 messages with a latency of 4.2 ms is unlikely with the average latency of the WiFi controller with a 1-second interval ping being 67.9 ms.

If the interval of the ping is lowered to 2 ms then the average latency of the connection to the ESP32 is reduced to 3.2 ms which could provide a usable latency for real-time computation offloading. The difference in the average latency between different ping intervals on the ESP32 makes it look like a WiFi controller idle state

problem because a reasonable ping can be achieved from the drone's WiFi controller when the load is significant enough. Aside from fixing the idle state of the WiFi controller, Nano-PoCL could achieve more performance over the same network connection by using a communication protocol that does not require acknowledgment of received messages such as User Datagram Protocol (Postel 2022).

The high average latency of the WiFi connection causes issues with the energy consumption of Nano-PoCL as well. Computing SeekBrightestPixel remotely consumes more energy than computing the kernel locally. However, at least 90.9% of this time is spent waiting for the WiFi connection to manage the packets. This on the other hand sounds very promising regarding energy consumption since the drone consumes energy even when idle and the time that is spent waiting can be reduced by improvements to the WiFi connection. Furthermore managing to resend the lost packets and handling the communication on the WiFi chip has an energy cost as well. Overall, the energy consumption could be way smaller if all of that time waiting for the WiFi connection could be reduced close to the current best-case latency. This could reduce the energy consumption of the ESP32 greatly and the improvement could be seen in the energy consumption of the Nano-PoCL as well since the WiFi controller is likely the greatest energy drain on the entire implementation.

Even with the SeekBrightestPixel kernel using more energy when computed in the remote server than locally, the overall state of Nano-PoCL sounds promising. The SeekBrightestPixel kernel used in the example is really simple because the kernel needs only a single iteration over the buffer that contains the image. If the kernel would do facial recognition instead and thus require more computation compared to the buffer size, then transferring the buffer to the server would make more sense. In addition, making the kernel more computation-heavy would accrue no additional cost to latency or power usage of the drone when it is offloading the computation. However local execution of the kernel would see a penalty in latency and energy consumption since the computation would take more time. By any means at the point where computing the result locally costs more than transmitting the buffer to the server offloading computation would become a net positive for the energy consumption of the drone.

In the end, the Nano-PoCL as a proof of concept for offloading computing from the Crazyflie 2.1 shows great promise for more general adoption in the design of resource-restricted devices. The size of the implementation is compact enough that it can fit many edge devices with ease. Furthermore, the latency of the optimal case that the main computation loop has is acceptable while the average latency is still troublesome. The latency issues seem like a problem of the ESP32 WiFi microcontroller and once the microcontroller's plausible problems related to irregular latency and idle state are handled then the computation offloading might be more

generally beneficial on the Crazyflie 2.1 as well. Finally, offloading computing with Nano-PoCL could already be beneficial on different platforms and it could be a key technology in the design of future resource-restricted devices.

References

Albers, Susanne and Antonios Antoniadis (2014). "Race to idle: New algorithms for speed scaling with a sleep state". In: *ACM Transactions on Algorithms* 10.2. ISSN: 1549-6325.

AMPOW (2022). *Lipo Voltage Chart: Show the Relationship of Voltage and Capacity*. URL: https://blog.ampow.com/lipo-voltage-chart/ (visited on 10/18/2022).

Aslan, Joshua, Kieren Mayers, Jonathan G. Koomey, and Chris France (2018). "Electricity Intensity of Internet Data Transmission: Untangling the Estimates". In: *Journal of Industrial Ecology* 22.4. ISSN: 1088-1980.

Bitcraze, AB (2022a). *250mAh LiPo battery*. URL: https://store.bitcraze.io/collections/spare-parts/products/250mah-lipo-battery?variant=39540518748247 (visited on 10/18/2022).

— (2022b). *AI deck*. URL: https://www.bitcraze.io/products/ai-deck/ (visited on 09/28/2022).

— (2022c). *bitcraze*. URL: https://www.bitcraze.io/ (visited on 09/28/2022).

— (2022d). *Crazy RealTime Protocol*. URL: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/crtp/ (visited on 10/13/2022).

— (2022e). *Crazyflie 2.1*. URL: https://www.bitcraze.io/products/crazyflie-2-1/ (visited on 09/28/2022).

— (2022f). *Crazyflie Packet eXchange*. URL: https://www.winbond.com/hq/product/mobile-dram/hyperram/?__locale=en (visited on 10/13/2022).

— (2022g). *Crazyflie PC client*. URL: https://github.com/bitcraze/crazyflie-clients-python (visited on 10/25/2022).

— (2022h). *Crazyflie python library*. URL: https://github.com/bitcraze/crazyflie-lib-python (visited on 10/25/2022).

Boukerche, Azzedine and Victor Soto (Aug. 2020). "Computation Offloading and Retrieval for Vehicular Edge Computing: Algorithms, Models, and Classification". In: *ACM Comput. Surv.* 53.4. ISSN: 0360-0300. DOI: 10.1145/3392064. URL: https://doi-org.libproxy.tuni.fi/10.1145/3392064.

BPI (2022). *LP-502030+PC*. URL: http://www.bpi.com.es/pdf/Litio%20recargable/Li-Pol%C3%ADmero/Prism%C3%A1ticas/LP502030+PC%20-%20250%20-%20Tech%20Data%20sheet.pdf (visited on 10/18/2022).

Day, Michael D., Charles E. Perkins, John Veizades, and Erik Guttman (June 1999). *Service Location Protocol, Version 2*. RFC 2608. DOI: 10.17487/RFC2608. URL: https://www.rfc-editor.org/info/rfc2608.

Docker (2022). *Docker*. URL: https://www.docker.com/ (visited on 10/25/2022).

Eshratifar, Amir Erfan and Massoud Pedram (2018). "Energy and Performance Efficient Computation Offloading for Deep Neural Networks in a Mobile Cloud Computing Environment". In: *Proceedings of the 2018 Great Lakes Symposium on VLSI (GLSVLSI'18)*. GLSVLSI '18. ACM. New York: ACM. ISBN: 1450357245.

Esmaeilzadeh, Hadi, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger (2012). "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3. ISSN: 0272-1732.

ESPRESSIF (2022). *ESP32*. URL: https://www.espressif.com/en/products/socs/esp32 (visited on 10/25/2022).

Flinn, Jason (2012). *Cyber Foraging*. Vol. 10. Morgan & Claypool Publishers. ISBN: 1608458504. DOI: 10.2200/S00447ED1V01Y201209MPC010.

FreeRTOS (2022a). *FreeRTOS*. URL: https://www.freertos.org/ (visited on 10/08/2022).

— (2022b). *FreeRTOS-Plus-POSIX*. URL: https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_POSIX/index.html (visited on 10/08/2022).

Goransson, Paul (2007). *Secure roaming in 802.11 networks*. 1st edition. Communications engineering series. Amsterdam: Newnes/Elsevier. ISBN: 1-281-12031-6.

GreenWaves Technologies (2022). *Ultra low power GAP processors*. URL: https://greenwaves-technologies.com/low-power-processor/ (visited on 09/29/2022).

Gupta, Ashok Kumar, Ashish Raman, Naveen Kumar, and Ravi Ranjan (2020). "Design and Implementation of High-Speed Universal Asynchronous Receiver and Transmitter (UART)". In: *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*. DOI: 10.1109/SPIN48934.2020.9070856.

IEEE (2013). "IEEE Standard for Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*. DOI: 10.1109/IEEESTD.2013.6515989.

Infineon (2022). *HyperFlash*. URL: https://www.infineon.com/cms/en/product/memories/nor-flash/hyperflash/#!products (visited on 10/13/2022).

Intel (2022). *Intel iRIS Xe Graphics*. URL: https://ark.intel.com/content/www/us/en/ark/products/208660/intel-core-i51145g7-processor-8m-cache-up-to-4-40-ghz-with-ipu.html (visited on 10/18/2022).

Jääskeläinen, Pekka, Carlos Sánchez De La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg (2015). "pocl: A Performance-Portable OpenCL Implementation". In.

Jin, Zilong, Chengbo Zhang, Guanzhe Zhao, Yuanfeng Jin, and Lejun Zhang (2021). "A Context-aware Task Offloading Scheme in Collaborative Vehicular Edge Computing Systems". In: *KSII Transactions on Internet and Information Systems* 15.2. ISSN: 1976-7277.

JTAG Technologies (2022). *JTAG*. URL: https://www.jtag.com/ (visited on 10/13/2022).

Khronos (2022). *OpenCL*. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf (visited on 09/28/2022).

Khronos® (2009). *OpenCL*. The Khronos® Group. URL: https://www.khronos.org/api/opencl (visited on 10/07/2022).

Lammle, Todd (2018). *TCP/IP*. Indianapolis, Indiana: Sybex. ISBN: 1-119-47170-2.

Lim, Byong-In, Kee-Hyun Choi, and Dong-Ryeol Shin (2005). "An Architecture for Lightweight Service Discovery Protocol in MANET". In: *Computational Science – ICCS 2005*. Ed. by Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-32118-7.

LLVM (2009). *LLVM*. URL: https://llvm.org/ (visited on 10/07/2022).

Marcham, Alex (2019). *Fog and edge computing : principles and paradigms*. 1st edition. Wiley series on parallel and distributed computing. Hoboken, New Jersey: John Wiley & Sons, Inc. ISBN: 1-119-52506-3.

Mark B (2022). *Lipo Voltage Chart*. URL: https://hobbygraderc.com/lipo-voltage-chart/ (visited on 10/18/2022).

Michal Babej Pekka Jääskeläinen, Jan Solanti (2022). *cl-pocl-content-size*. URL: https://registry.khronos.org/OpenCL/extensions/pocl/cl_pocl_content_size.html (visited on 10/11/2022).

Moore, Gordon E. (1998). "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1. ISSN: 0018-9219.

Motorola, Inc. (2022). *SPI Block Guide*. URL: https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf (visited on 09/28/2022).

Nanda, Umakanta and Sushant Kumar Pattnaik (2016). "Universal Asynchronous Receiver and Transmitter (UART)". In: *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*. Vol. 1. IEEE. ISBN: 9781467392051.

Navardi, Mozhgan, Aidin Shiri, Edward Humes, Nicholas R. Waytowich, and Tinoosh Mohsenin (2022). "An Optimization Framework for Efficient Vision-Based Autonomous Drone Navigation". In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. DOI: 10.1109/AICAS54282.2022.9869975.

Osrael, Johannes, Lorenz Froihofer, and Karl M. Goeschka (2006). "A system architecture for enhanced availability of tightly coupled distributed systems". In: *First International Conference on Availability, Reliability and Security, Proceedings*. Vol. 2006. IEEE Comp Soc. Los Alamitos: IEEE. ISBN: 9780769525679.

Palossi, Daniele, Francesco Conti, and Luca Benini (2019). "An open source and open hardware deep learning-powered visual navigation engine for autonomous nano-UAVs". In: *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE International Conference on Distributed Computing in Sensor Systems. IEEE. New York: IEEE. ISBN: 9781728105703.

Palossi, Daniele, Nicky Zimmerman, Alessio Burrello, Francesco Conti, Hanna Muller, Luca Maria Gambardella, Luca Benini, Alessandro Giusti, and Jerome Guzzi (2022). "Fully Onboard AI-Powered Human-Drone Pose Estimation on Ultralow-Power Autonomous Flying Nano-UAVs". In: *IEEE internet of things journal* 9.3. ISSN: 2327-4662.

Postel, J (2022). *RFC 768 User Datagram Protocol*. DOI: 10.17487/RFC0768. URL: https://datatracker.ietf.org/doc/html/rfc768 (visited on 10/24/2022).

Rabaey, Jan M. (1996). *Digital integrated circuits : a design perspective*. Prentice Hall series in electronics and VSLI. Upper Saddle River (NJ): Prentice Hall. ISBN: 0-13-178609-1.

RISC-V International (2022). *RISC-V*. URL: https://riscv.org/technical/specifications/ (visited on 09/28/2022).

Ruparelia, Nayan (2008). *Cloud computing*. The MIT Press essential knowledge series. Cambridge, Massachusetts ; The MIT Press. ISBN: 0-262-33413-5.

Satyanarayanan, Mahadev (2001). "Pervasive computing: vision and challenges". In: *IEEE Personal Communications* 8.4. ISSN: 1070-9916.

Solanti, Jan (2020a). *Distributed Low Latency Computing With OpenCL: A Scalable Multi-Access Edge Computing Framework*.

— (2020b). *Distributed Low Latency Computing With OpenCL: A Scalable Multi-Access Edge Computing Framework*. Informaatioteknologian ja viestinnän tiedekunta - Faculty of Information Technology and Communication Sciences.

Solanti, Jan, Michal Babej, Julius Ikkala, Vinod Kumar Malamal Vadakital, and Pekka Jääskeläinen (2022). "PoCL-R : A Scalable Low Latency Distributed OpenCL Runtime". In: Orailoglu, Alex. Springer.

sourceware (2022). *Newlib*. URL: https://sourceware.org/newlib/ (visited on 10/09/2022).

Steen, Marten van and Andrew S. Tanenbaum (2020). *Distributed systems*. 3.03. Maarten van Steen. ISBN: 978-90-815406-2-9. URL: https://www.distributed-systems.net/index.php/books/ds3/.

STMicroelectronics (2022). *STM32*. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f405-415.html (visited on 10/17/2022).

Tanenbaum, Andrew S. (2015). *Modern operating systems*. Ed. by Herbert Bos. Fourth edition. Pearson. ISBN: 9781292061955.

The Open Group (Sept. 2022). *The Open Group Base Specifications Issue 7*. URL: https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10 (visited on 09/28/2022).

The Open Group, IEEE (2018). "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. DOI: 10.1109/IEEESTD.2018.8277153.

Tian, Xianzhong, Juan Zhu, Ting Xu, and Yanjun Li (2021). "Mobility-included DNN partition offloading from mobile devices to edge clouds". In: *Sensors (Basel, Switzerland)* 21.1. ISSN: 1424-8220.

Wang, Xiaofei, Zhengguo Sheng, Shusen Yang, and Victor C. M Leung (2016). "Tag-assisted social-aware opportunistic device-to-device sharing for traffic offloading in mobile social networks". In: *IEEE Wireless Communications* 23.4. ISSN: 1536-1284.

Waterman, Andrew Shell (2016). *Design of the RISC-V Instruction Set Architecture*.

Winbond (2022). *HyperRAM*. URL: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/cpx/ (visited on 10/13/2022).

Zimmermann, H. (1980). "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection". In: *IEEE Transactions on Communications* 28.4. DOI: 10.1109/TCOM.1980.1094702.