Implementation of FPGA-based star tracker pre-processing pipeline

Oscar Björkgren 38655

Supervisors, Åbo Akademi University: Annamari Soini, Johan Lilius
Advisors, Aboa Space Research Oy: Tero Säntti, Tuomo Komulainen
Master's thesis in Computer Engineering
Åbo Akademi University
2022

Abstract

The processing pipeline for a star tracker requires computation within various problem domains. The first stage of star detection and tracking is to extract star features from image data. The image processing, described as the pre-processing pipeline, is implemented and documented in this thesis. The goal is to provide a theoretical basis of hardware aspects, image processing, and star tracker systems to support the design choices made for the pre-processing pipeline.

The hardware design is built for a Xlinix 7-series FPGA, functioning as a testbed for the pre-processing pipeline. The Opal Kelly XEM7305 module provides the components, such as power, clock, SDRAM, and a communication channel to the FPGA.

The system requirements form the high level goals of the pre-processing pipeline. The requirements create a framework for the FPGA design to be developed with robust and user-firendly engineering principles. The requirements are implemented in a design consisting of a computation unit, communication unit, memory controll unit, and client-PC software.

Extracted stars are rendered on top of images provided to the pre-processing pipeline. The successfull extraction of stars can be verified through the client software. The processing time of an image is satisfactory, with a very low variation. This is a precondition for integration with other star tracker components, when following the principle of predictability. The performance of the most complex part of the computation unit, the CCL process, is compared to high performance software implementations. A satisfactory result is concluded when the software execution is put to scale with the lower clock speed of the hardware implementation.

The resource use of the FPGA, provided by the Xilinx Vivado design suite, is reviewed with the conclusion being that the FPGA part fits the design well with a high utilization rate of LUT's and BRAM. Parallelization could be increased to utilize more DSP blocks for faster results.

Keywords

Star tracker, image processing, CCL, box filter, FPGA design

Abbreviations

Contents

List of Figures

List of Tables

1. Introduction

A hardware-based solution for a software algorithm is sometimes needed within application areas where performance matters the most. One example of this is the aerospace industry where the most optimized solution is sometimes the only choice. In this thesis, I work together with Aboa Space Research Oy to create a hardware implementation of a pre-processor module for a star tracker system using a software component as a basis. A star tracker is commonly used as part of a larger system where the role of the tracker is to provide orientational data to other components. This information might be used for navigation and control or in combination with scientific instruments to help with further analysis. Hardware design can be inspired by software algorithms when designed for a specific purpose, as in this case. The software defines the functional part of the system, which leaves a big part of the supporting architecture to be designed. In this thesis, I will explain the process of implementing functionality to a digital circuit, by example.

To complete a set of tasks in any environment, a scientific project needs well-defined tools to reach its goals. In technologically restricted environments such as space, tools often need to be multipurpose for optimal use. An image sensor is in this case used to provide reference data for a star tracker. This image sensor can provide data to multiple systems by reusing data or capturing new images with other settings. The purpose of the system in this project is to refine data from predefined sensors to provide additional and reinforcing information about the environment for the use of other scientific instruments. The goal of this thesis is to create a roadmap of the system design process and include theory of relevant areas.

2. Custom hardware design in general

2.1 Single and general purpose processors

A general purpose processor refers to a hardware computing platform which is designed for universal use with broad benefits across different problem-solving domains. Computing platforms such as microprocessors and CPUs are examples of this type of processor, their hardware logic is implemented in such a way that it enables a large variety of computations to be performed. The characteristics of a processor for general problem solving are well suited to common tasks where there might be many hard- and software abstractions between the application interface and hardware logic. This is a necessity in, for example, PCs.

When hardware is required for a single purpose with a finite set of tasks to be performed, there will be drawbacks with using hardware designed with flexibility in mind. This could be compared to the use of a multitool for driving a screw into a piece of wood when, in fact, only a screwdriver is needed. It is not the wrong way to do it, but there is a more optimized way of achieving the result. Any CPU could be used for computer graphics calculations but since GPUs are designed for the single purpose of this type of processing, they are a better tool for the job.

Single purpose processors have a rich history with roots in the early ages of computers. Vector processors were, for example, used in supercomputers. This was before computers were seen as general purpose technology and high performance computing was about as common as ordinary PCs. In recent years, the utilization of specialized hardware has been rising, with new application areas evolving. Single purpose processors such as GPUs and field programmable gate arrays (FPGAs) have found themselves into areas such as machine learning, cryptocurrency mining, and other high performance computing applications. As the

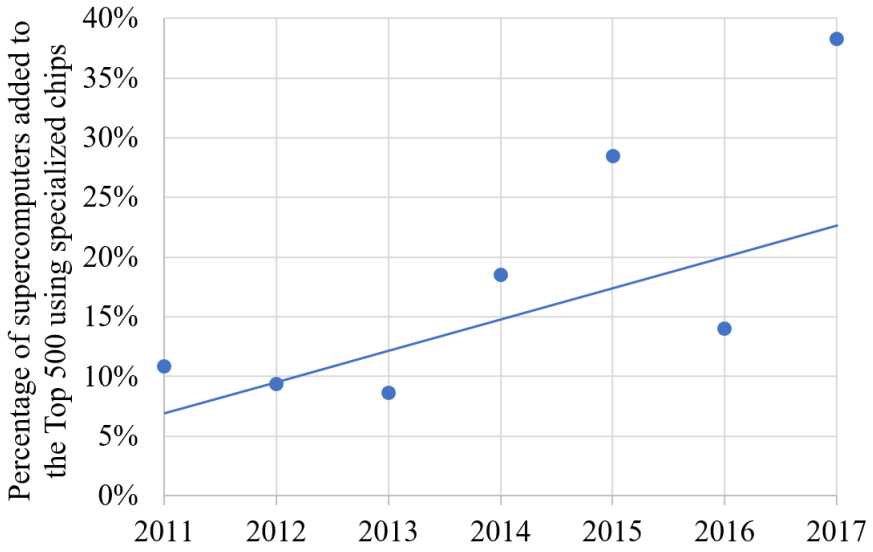

Figure 2.1: Graph showing increasing popularity of specialized hardware in high performance computing [1].

hardware used in aerospace applications generally is specialized according to strict requirements, single purpose computing platforms are heavily used. For example, the Nasa's Perseverance Mars rover relies heavily on Xilinx manufactured FPGAs for different tasks, such as image processing pipelines XilPerservance.

2.2 FPGA Overview

A field programmable gate array, or FPGA, is a digital integrated circuit that consists of millions of logic blocks that can be configured to perform different operations. The process of configuring the logic blocks is called FPGA design and can be compared to the design of integrated circuits. Reconfigurability and ease of design are the main advantages when using an FPGA as a computing platform. When comparing an FPGA design with a solution that is developed using software, the FPGA stands out with process pipelining and a high level of parallel processing. The difference between executing a software program on a processor and executing the equivalent operation on an FPGA is that the compiled software instructs the processor to do operations in a generic way, as in loading a register and shifting a bit. An FPGA does not need to explicitly load a register, because the registers are by default connected to the input of the operation, which is defined by the FPGA design. A bit shift operation is typically an inexpensive

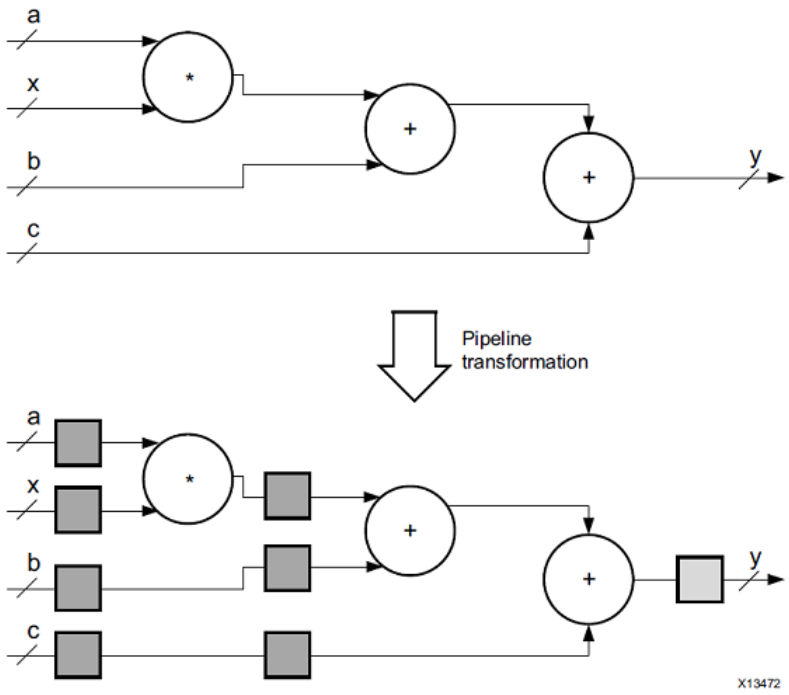

Figure 2.2: Figure showing two execution paths for the expression $y = (a \times x) + b + c$. The operations on the first path depend on the output from previous operations, whereas the operations on the second path depend on separate input register. The second path is a pipeline transformation of the first path. [2]

operation regardless of the computing platform, but the FPGA can pipeline the result directly into another register or operation, which saves clock cycles. The pipeline principle is demonstrated in Figure 2.2 by comparing it with a non-pipelined solution. The pipeline transformation enables the inactive parts of the calculation to be utilized when they would normally be idle, waiting for all parts to finish. [2]

Xilinx is one of the key companies involved in FPGA production, both currently and historically. They introduced the world to FPGAs in 1985 and have since then led the programmable logic technology industry, with a 51% market share of programmable logic device suppliers in 2017. Major end market categories for Xilinx are aerospace, defense, and communications. The main competitor of Xilinx is Intel with a market share of 37%. [8][9]

Due to the significance of Xilinx, and relevancy in this project, FPGA technology discussed in this work is Xilinx based. Differences between manufacturers include technological and architectural nuances. For example, Xilinx exclusively

manufactures static random access memory (SRAM)-based FGPAs. This refers to the technology of the programmable parts of the FPGA. Other available technologies are flash and anti-fuse, which are less common than SRAM. [10]

The basic internal elements of an FPGA consist of circuits such as look-up tables, flip-flops, wires, and input/output pads. Using these elements the FPGA design is implemented in the hardware. The architecture is illustrated in Figure 2.3 which shows a matrix arrangement of configurable logic blocks which are connected to each other with an interconnect, consisting of wires and switch matrices. The wires also run to the input/output pads, which enables interfacing with off-chip components such as synchronous dynamic random access memory (SDRAM) or sensors. Configurable logic blocks (CLBs) contain the look-up tables and flip flops. An overview of a CLB is shown in Figure 2.4. [2]

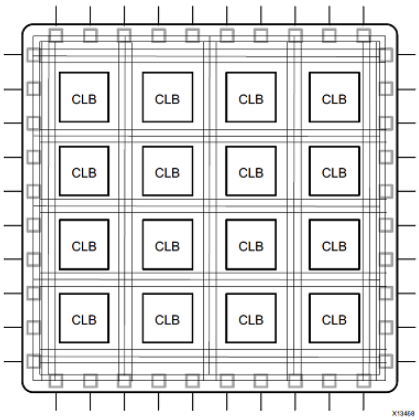

Figure 2.3: FPGA architecture. An FPGA consist of configurable logic blocks, input/output pads, and an interconnect [2].

The slices in Figure 2.4 represent core elements of an FPGA. Slices consist of look-up tables, storage elements, multiplexers, and carry logic. Depending on the FPGA, there are different numbers of slices per CLB. The slices are collectively responsible for implementing a significant portion of the FPGA design. [3]

Look-up tables, or LUTs, are mainly used as logic elements in FPGAs. They can implement any boolean logic operation by combining memory cells with multiplexers. A look-up table is essentially a truth table that can be used for reading the result of boolean operations. Memory cells and multiplexer routings are initialized by the FPGA design, which is what makes the look-up table element configurable. An example of a four memory cell wide LUT is shown in Figure 2.5. [2]

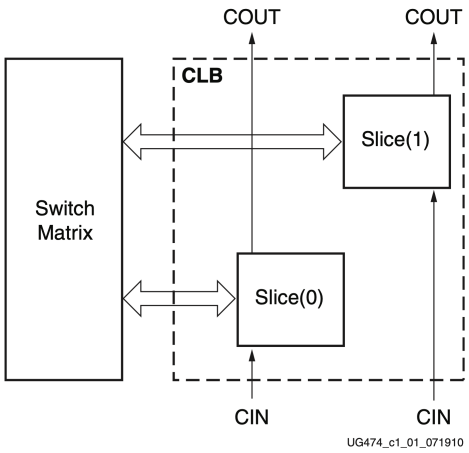

Figure 2.4: Diagram of a configurable logic block. A CLB consists of logic element containers, called slices, inputs, and outputs. Both slices are attached to the switch matrix, connecting the CLB to other CLBs. [3]

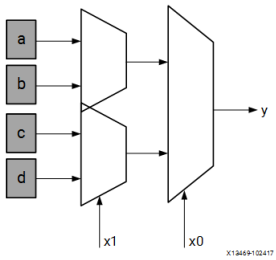

Figure 2.5: Example of a look-up table with four memory cells and two multiplexers. The FPGA design initializes memory cells **a-d** and uses inputs **x1** and **x0** to select value **y**. [2]

The circuit elements discussed in this chapter form the basic blocks of FPGA technology. A larger set of elements are often included in an FPGA to enable some optimizations and further functionality in the design. For example, there are different types of slices a CLB can contain, which make it more suitable for storage purposes rather than logic. Signal processing pipelines are a popular use case for FPGAs and they often contain slices specialized for operations in this problem domain. Digital signal processing can be accelerated by parallelizing operations in a single instruction multiple data (SIMD) fashion, thus to maximize the advantage there can often be found a large number of these slices. Figure 2.6 shows an example of an FPGA configuration with some of these specialized blocks mapped out. The green CLBs are designated to memory usage and are closely coupled to the red digital signal processor (DSP) blocks to enable fast processing

of the stored data. The purple external memory controllers use the input/output pads to interface with external memory such as SDRAM. [2] [3]

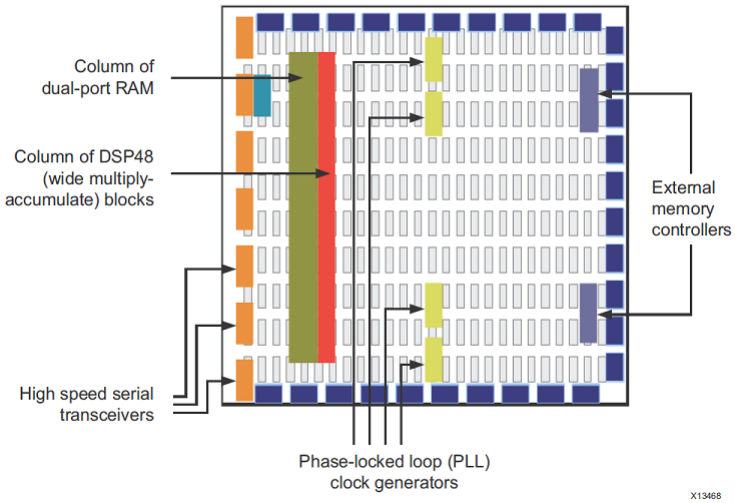

Figure 2.6: DSP system design on FPGA [2].

When comparing FPGAs with each other the main performance metrics are power consumption and number of logic cells. The number of logic cells determines the size and complexity of the design that the FPGA can fit. Additionally, the FPGAs can often be compared by specifications such as specialized CLB slices, I/O interfaces, and communication rates. Clock speed is seldom compared because although there is an upper limit, it is often design specific. [3] [2]

2.3 System design with HDL

The main tool for creating an FPGA design is a hardware description language (HDL). VHDL and Verilog are examples of commonly used HDLs. Compared with conventional computer programming languages, HDLs are used in a data flow fashion to describe the timing and logic of an integrated circuit. The data flow description of a circuit in an HDL is such that all code blocks are executed simultaneously by default. This is one of the main advantages of FPGAs as the system is inherently parallelized. It creates also one of the biggest challenges in HDL based design as it complicates the timing of events and resource concurrency.

The output of an HDL design is produced by a build process that computes the high level data flow description into low level circuit elements and routing.

This can be compared to the compilation phase of code written in a statically typed software programming language.

The synthesis process is where the HDL design known as register transfer level code is transformed to a gate level description called netlist. The netlist is a mapping of circuit components that are found inside the FPGA. The convenience of HDL based design is being able to describe the circuit at a higher level. In practice this means that registers and buses are created and connected to be inferred into low level circuitry at the synthesis phase. When developing a design against a specific FPGA, a bitstream file is created to configure the device with. The end product of an HDL based design is the bitstream file or, alternatively, the synthesis result as it can be used for simulation purposes by a test bench. [11]

2.4 System design with IP cores

Designing a hardware system often involves components of a reusable nature, such as dividers, communication protocols, and memory controllers. Intellectual property (IP) cores are distributions of components such as these. An IP core can be of varying complexity as it can consist of a solution for a specific task or describe a complete system on a chip (SoC). IP cores are licensed through a patented design which is what the name refers to. They can often be configured to suit different properties of the implementation such as data width and clock rate. The core is then connected to the system using an HDL. The core itself may consist of software or hardware macros. The software macro, or soft IP core, is an HDL implementation of the IP core logic that is built to the specific FPGA with the rest of the system design. A hardware macro, or hard IP core is prebuilt and FPGA specific. [12] [11]

The hardware sector and consumer electronics industry rely heavily on reusable IP cores. For example, the SoC or motherboard found in mobile phones often includes advanced RISC machine (ARM) based processor cores licensed from ARM holdings. The producer of the mobile phone benefits from avoiding the development of the processing unit but still has the ability to customize the SoC to a high degree by combining different processing units, memories, and other IP cores. Figure 2.7 shows the architecture of an SoC which has an ARM based CPU, and other modules commonly found in mobile phones. In addition to the CPU, any of the remaining modules might also be licensed IP cores. [4] [13]

In aerospace applications, the advantages of IP cores are utilized in the same

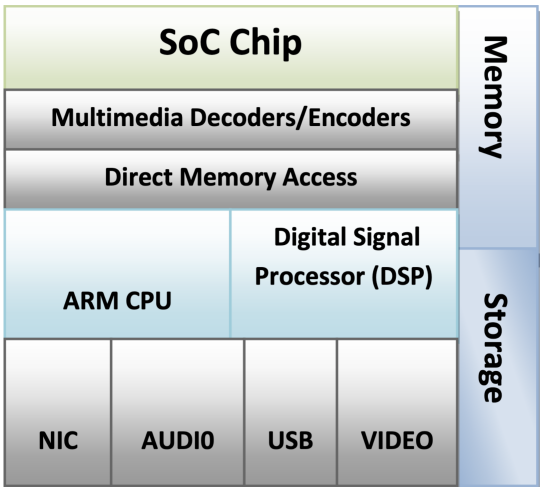

Figure 2.7: Common SoC found in smartphones includes ARM based CPU and DSP. [4]

manner as within the mobile phone industry. Additional requirements such as fault tolerance limits, redundancy, and formal verification of the IP cores increase the advantages of their usage when comparing with in-house developed solutions. This, however, comes with pricier licenses compared with non-aerospace grade IP core designs. Also, due to the specialized requirements, lower competition in the market sector can be expected, increasing the prices even more. [14]

In addition to FPGA products, Xilinx also provides design tools and IP cores to help with development on their FPGA platforms. Their IP core library ranges from common integrated circuits (IC) to complete microcontrollers. Some of the cores are available free to use through the Vivado design suite, and a majority of them are soft cores to allow compatibility across their FPGAs. Integration of the IP cores and other development are also done through Vivado. [11]

3. System requirements

A hardware implementation of an image analysis software component is to be developed to be used on an FPGA. The software in question is a product of a research project regarding detection and analysis of space debris in low earth orbit. Using computer vision among other analysis, space debris is identified from image data. With the help of a star catalog, the image is mapped to a position in space to provide positional data of the detected debris. A diagram of the system is shown in Figure 3.1. The software component relevant to this project is the pre-processing and segmentation part of the diagram, which is in the input phase of the complete system. [5] [15]

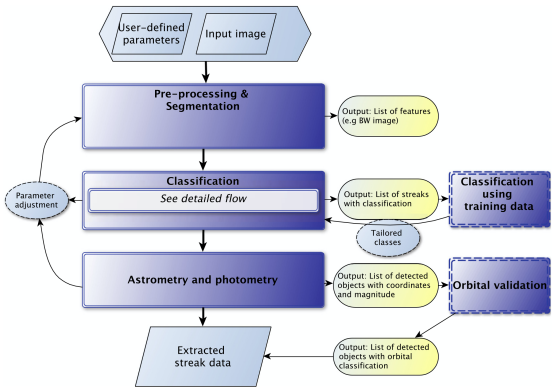

Figure 3.1: Flowchart of StreakDet software [5].

The star extraction process in StreakDet passes the input image through a series of algorithms to identify stars. The algorithms are well-known image filters and analysis methods, that in combination with each other, produce a way to extract objects from an image. The main task in this project is to implement the algorithms to a hardware design using a hardware description language.

Star extraction is a key stage of a star tracker system which uses a star catalog to match the extracted stars against. Using the star matches and catalog information the position and orientation of the captured image can be determined.

The goal of this project is to create a pre-processing pipeline to be used in a star tracker.

3.1 System specification

The specification consists of development and system requirements as well as a description of tools and components. System requirements are created based on meetings and discussions with Aboa Space Research Oy. Some requirements are derived from higher level requirements to suit the development process. Table 3.1 presents an overview of the system.

Req. Id	Description
1	Client endpoint for FPGA module
1.1	Bitstream can be written to FPGA
1.2	Image processing pipeline parameters can be written to FPGA
1.3	Allows for debugging of system
1.4	Provides visual results with metadata
2	Validation and verification of star extraction
2.1	First stage validation and verification with software prototype
2.2	Second stage validation and verification on FPGA platform
3	Star extraction result and operation
3.1	Image coordinates of extracted stars are stored
3.2	Centroid of detected stars is calculated
3.3	Error handling to avoid false detection

Table 3.1: System requirements specification

3.1.1 System components

Processing nodes

The system consists of two nodes, the host machine and the target FPGA. Because the application of this system will be different in a production environment, the FPGA part should not be too dependent on the host machine. When integrated

into a larger system with shared memories and communication, the FPGA will have to be able to read an input image directly from memory. The same applies to the output, the register where the output is written needs to be configurable and not sent to a host machine by default.

FPGA module

The FPGA module Opal Kelly XEM7305 chip with a Xilinx Spartan-7 FPGA will function as a testbed for development. The module contains a part from the FPGA external SDRAM memory and communication interfaces such as USB 3. These features allow for host machine communication and data storage of features such as images, star catalog, and output. These data could be stored on the FPGA itself but using the SDRAM is a sustainable solution because it allows for shared memory usage in production and more flexible image and star catalog sizes.

4. FPGA-based computer hardware

Common computer hardware is the corner stone of any digital circuit that needs to perform common tasks, such as storing and transmitting data. With the help of the basic elements of an FPGA discussed in Chapter 2, circuits of this nature can be created. Although an 8-bit adder circuit can be regarded as common computer hardware, the focus of this chapter is on larger circuits used as hardware modules, such as memory and communication interfaces. The components discussed are essential building blocks for the image processing pipeline. Figure 4.1 shows an overview of these components on the XEM7305 FPGA module.

4.1 Communication

The system requirements state a communication channel to be available for shared memory usage and controlling the FPGA. Machine-to-machine communication is enabled through the interfaces available on the Opal Kelly FPGA module. The default interface on XEM7305 is USB C. [6]

The advantage of the USB is flexibility and ease of use. Transfer speed is not a high priority in this case because of the relatively small size of data, which is less than a megabyte. The USB module on the XEM7305 contains a USB C port and a Cypress FX3 peripheral controller, shown in Figure 4.1. The peripheral controller operates the data bus by implementing a communication protocol stack consisting of a physical, link, and protocol layer. It allows the usage of the USB communication link on the FPGA module by exposing a programmable interface. The programmable interface can be used by various systems on the FPGA module, such as an image sensor or the FPGA itself. To allow the usage of the peripheral controller's programmable interface in the FPGA design, a hard

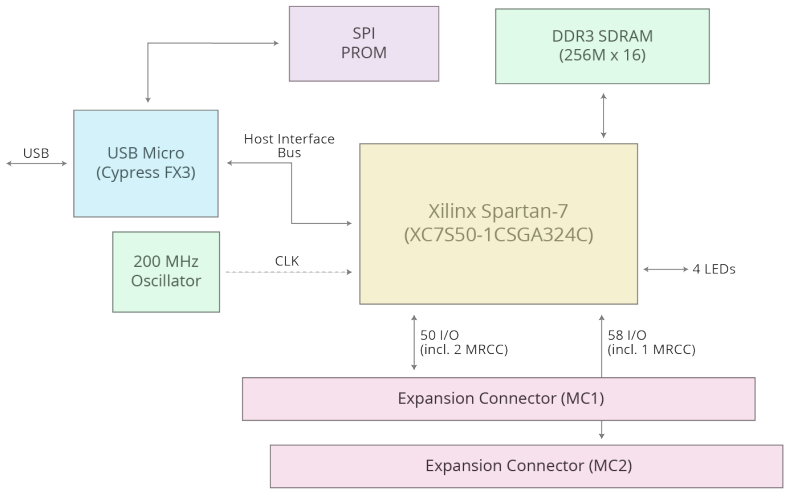

Figure 4.1: Overview of the FPGA module showing the FPGA itself in the center, SDRAM off-chip memory on the top right, and USB interface on the left. [6]

IP core is provided. [6] [16]

Both the USB 3 standard and the FX3 peripheral controller support serial and parallel communication, but the hard IP core connecting to the FX3 interface supports only serial communication. This simplifies the use of the IP core, requiring less FPGA fabric, but also reduces the throughput. [6] [16]

The internal communication mechanism of an FPGA consists of the FPGA interconnect. The interconnect is the physical layer of the communication protocol stack, leaving the rest of the stack to be implemented in the FPGA design with HDL or IP cores.

The limiting factor and bottleneck of the system is often some off-chip hardware module that places certain requirements on the communication, for example, the USB or memory controller. When working outside these types of requirements, the throughput of the FPGA-based system can be optimized by increasing the bit width between modules. When applying this to the image processing pipeline context, this might be practical within communication of image data. The implementation of this requires more memory cells and wires which, in turn, increases the footprint of the FPGA design.

When compared with a software program, the same optimization method of maximizing bit width might be considered. This, however, does not inherently result in a gain in throughput because conventional CPUs have a fixed bit width for their operations. When this limit is exceeded, the operation needs to be

performed across multiple clock cycles which counteracts the wanted result. The bit width of an operation on an FPGA during a clock cycle is consistent with the HDL design.

4.2 Memory

To enable processing of image data, a set of storage mechanisms is necessary. The use of a certain memory type is mostly dependent on the size of the data and how it will be used. As in conventional CPU architectures, different levels of memory are used to satisfy these dependencies.

FPGAs are generally not well suited for storing a large quantity of data, as it is considered inefficient use of available CLBs. Since customization is a key feature, the most standardized parts, such as memory, are often placed outside the FPGA. Off-chip memory allows storage of large data, such as images, freeing up CLBs to other usage such as computation and logic. The drawback with off-chip components is that they are often a bottleneck in the system. This is due to a number of factors such as restrictions in clock speed, concurrency, and bus width. A larger FPGA with an increased number of CLBs could offset the need for off-chip memory avoiding the bottleneck, with consequences being greater power consumption and physical size, potentially invalidating the solution.

An overview of the XEM7305 module and the memory types can be seen in Figure 4.1 where the off-chip SDRAM memory is on the top right and the lower level memories inside the Xilinx Spartan 7 FPGA in the center.

4.2.1 SDRAM

One of the most common storage mechanisms is SDRAM which is useful for storing data that is frequently used and ranges from kilobytes to megabytes in size. Key features include random access, i.e. same access time for any part of the memory, and high bandwidth. It is often paired with FPGAs, because it is next in the memory hierarchy after on-chip storage mechanisms.

At the core of SDRAM lie capacitors trapping the charge representing a bit. An overview of a typical SDRAM circuit is shown in Figure 4.2. The capacitors are refreshed periodically to retain the stored data. This makes SDRAM a volatile memory, requiring power to keep its state. The advantage over closely related memory types, such as flash memory, is that it does not degrade at the same

rate of read/write cycles and it is truly random access, giving the advantage to SDRAM as a volatile working memory in most FPGA applications. [7] [17]

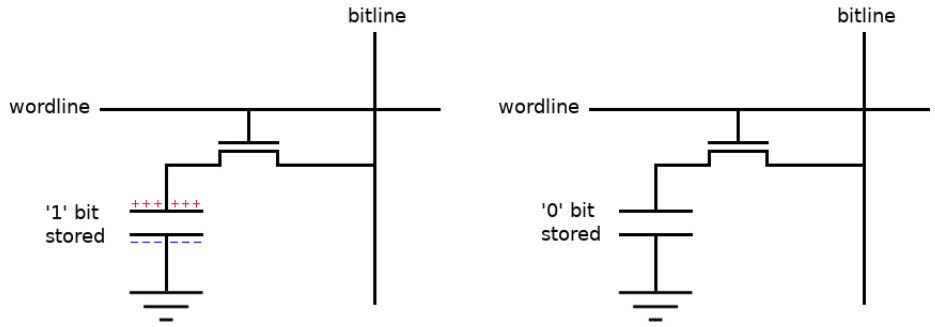

Figure 4.2: SDRAM memory cell. The capacitor traps the charge which is controlled by the transistor at the top. [7]

The memory cells in SDRAM is ordered in rows and columns, forming an array. A collection of these arrays is referred to as memory banks. The number of rows, columns, arrays, and banks varies depending on the type of SDRAM but is usually a power of two. Figure 4.3 shows the geometry of an SDRAM with four banks with four arrays each. The data width of a memory is the number of arrays that are stacked inside each memory bank. When accessing a specific row and column, the number of affected bits is the same as the data width of the memory.

To increase SDRAM performance, double data rate (DDR) is used to access data at double the clock rate. DDR enables accessing a memory bank on both the rising and falling edge of the clock cycle. A prefetch mechanism enables the transfer of multiple bits concurrently per bank access, increasing the throughput even more. This mechanism enables efficient use of the internal SDRAM bus width by also transferring data from neighboring addresses. The gain comes from the fact that the time to access a single address is the same as accessing the neighboring ones as well. The prefetch mechanism gives an advantage to data which is structured in a way where each access is in adjacent addresses. The width of the prefetch data depends on the DDR version. Other variations between DDR versions are the supported clock frequency and burst length, which are features related to the prefetch mechanism. The burst length specifies the number of SDRAM words contained in each read or write operation. [18]

The XEM7305 features a Micron DDR3 SDRAM chip with a 32-bit wide interface. The memory part has 8 banks, 32 768 row addresses, 1024 column

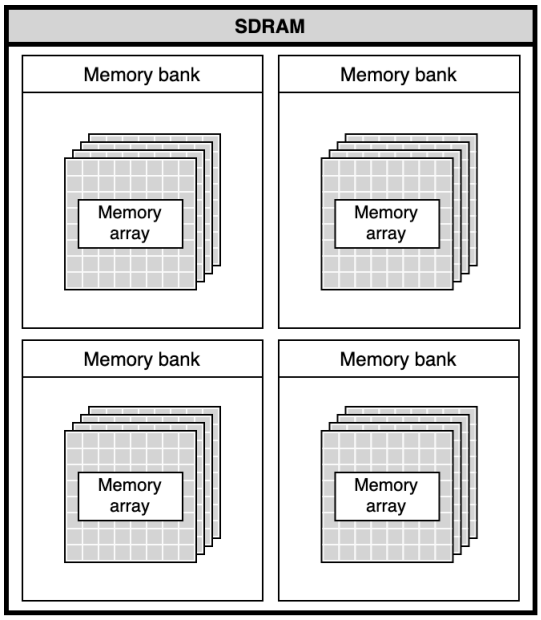

Figure 4.3: SDRAM geometry.

addresses, and 16 bits to each address, giving the a capacity of 4Gb. As previously discussed, the DDR version dictates the prefetch data width, which in DDR3 is 8 words, as is the burst length. The word length of the memory is 16 bit, which results in 128-bit burst transfers. The SDRAM is rated with a clock frequency of 1600 MHz, giving a theoretical peak bandwidth of 3.2 GB/s. In practice, the bandwidth is lower and depending on system state and access location. [6]

The oscillator on the FPGA board provides a 200 MHz signal, which is also the limit of the Spartan 7 FPGA. This allows a maximum clock frequency of 325 MHz to be generated through the SDRAM controller interface and then scaled up to 650 MHz with DDR at the SDRAM. [6] [19]

The memory controller interface connects the FPGA to the SDRAM controller, which in turn sends commands to the SDRAM core. The interface is unique to each memory controller and depends on all parameters discussed in this chapter such as memory geometry, DDR version, and clock frequency. [6]

4.2.2 Block RAM

When data size is within a few megabytes or less, lower level storage mechanisms become suitable to use as memory. The block random access memory (BRAM) is a type of on-chip memory, which uses memory cells available on the FPGA. The specification of an FPGA chip often includes the available BRAM capacity.

Although BRAM is a type of on-chip memory, it is separate from the CLBs on the FPGA. In the case of SRAM based FPGAs, the memory cells of the BRAM are essentially SRAM cells, arranged in arrays. The BRAM is made available to the other CLBs by being attached to the FPGA interconnect.

BRAM can be configured in size and functionality depending on the needs. Common BRAM arrays are 36Kb that can either be divided into smaller arrays, or combined with other arrays to form a larger memory resource. A useful feature often found regarding BRAM is multiport support. This enables concurrent usage of the memory resource. In case of concurrency issues, the configurable operation mode of the BRAM dictates how to resolve them. For example, in a dual port configuration, the read first operation mode will prioritize a read operation when the other port requests a write operation on the same address. [20]

Similarly as SDRAM, BRAM also is synchronous and requires a clock signal. The latency depends on the configuration and state of the memory, but is usually within a few clock cycles. The low latency, together with the high configurability, makes BRAM a useful resource in most FPGA designs. [20]

4.2.3 Distributed RAM

The lowest level of memory found in FPGAs is distributed RAM (DRAM). It is a type of on-chip memory which utilizes the same CLBs that are common with other logic of the FPGA design. This makes the use of DRAM a trade-off between fast memory and CLB resources. If the other parts of the FPGA design are large, there might not be room for DRAM, leading to slightly less performant BRAM to be used instead.

The slices in Xilinx CLBs are either of type SLICEL or SLICEM. The slices in a Xilinx 7 series FPGA CLB consist either of two SLICELs or one SLICEL and one SLICEM. DRAM is available only on SLICEM type of slices. The SLICEM contains a superset of SLICEL elements, allowing SLICEM to be used for non-storage purposes as well. The geometry of DRAM memory is defined by the configuration of SLICEM slices. The configuration and decision to use DRAM is inferred from the FPGA design. A single SLICEM CLB in the Xilinx 7 series FPGA allows the LUT to be used for 64 bits of RAM storage. The DRAM capacity is increased beyond this by connecting DRAM enabled CLBs together. The DRAM capacity and number of SLICEM enabled CLBs are often mentioned in the FPGA product specification. [3]

Since DRAM resides in the FPGA fabric, it communicates through the FPGA interconnect making read and write operations fast. Read and write operations require a single clock cycle to complete, making DRAM also very simple to use from a HDL design perspective. [20]

4.3 Clock

As most digital hardware systems, FPGAs use clocks to synchronize operations. The clock is an off-chip component, providing the signal through input/output pads to the FPGA. Figure 4.1 shows the off-chip clock circuit on the left side of the FPGA on the XEM7305 module. The clock signal is generated by an oscillator circuit that produces a specific frequency. This frequency may then be used as such or transformed into additional clock signals within an HDL design or IP core. For example, the DDR SDRAM utilizes the clock signal on both the rising and falling edge to increase the throughput of data. The clock signal on the XEM7305 is passed from the external clock module to the SDRAM module within the HDL design. This ensures that the SDRAM and FPGA design are synchronized. [6]

The clock module on the XEM7305 provides a 200 MHz frequency signal. An additional 100.8 MHz clock signal is provided by the USB interface, which is required when interacting with the USB peripheral controller. A third clock is used by the SDRAM, which is derived from the 200 MHz clock module. [6]

In addition to using the clock signal to drive tasks, a common practice within system modeling with an HDL is to drive tasks using signals that occur more spontaneously. This could, for example, be a signal originating from a sensor on the FPGA module that has an interface to the FPGA design without a dependence to a clock. This enhances the suitability of using FPGAs for real-time requirements since the reaction to an input can be within clock cycles instead of milliseconds. Another reason to use FPGAs for real-time applications is predictability. For example, when using a clock divider to run a task at every 5 ms, the task will run precisely at 5 ms intervals as long as the clock oscillator itself is accurate. Software based real-time applications cannot guarantee the same level of confidence for the timing requirements due to possible scheduling and concurrency related overhead.

5. Star tracker systems

Determining the position of an object in space is a task where multiple observations and instruments are used to produce an understanding of the orbit or trajectory. The position can be described by different means. Astronomical coordinate systems and reference frames are used depending on the type of object and orbit. Objects in earth orbit are commonly described using the equatorial coordinate system. The equatorial coordinate system is geocentric and uses right ascension and declination as spherical coordinates, comparable to the geographical coordinates longitude and latitude. Figure 5.1 illustrates the spherical coordinates in the context of a satellite orbit. [21]

To determine the position of a spacecraft in earth orbit, estimations on can be done remotely by the use of orbital mechanics. The position of a satellite with a known orbit can be determined by its six Keplerian orbital elements. Five of the elements contribute to size, shape, and orientation of the orbit. The sixth element allows for calculation of the satellite position at a given moment in time. As time passes, the elements used for position estimation need to be adjusted to maintain an accurate model of the orbit. Remote observations and onboard measurements are used for making this possible. [21]

Satellites are monitored and tracked remotely by ground stations across the globe using radar and radio communication systems. The onboard systems of a satellite often include a number of instruments for determining the position and attitude. This information is often needed for managing the mission of the satellite. If the mission involves remote sensing tasks, such as multispectral imaging for example, maintaining the attitude of the spacecraft at a specific position in orbit might be mission critical.

The attitude of a satellite is the orientation of the spacecraft relative to a frame of reference. The attitude matrix is a common way to express the attitude. The 3 x 3 matrix contains angles of each axis, allowing accurate interpretation and calculation to be made. Figure 5.1 shows the vectors of a satellite attitude. These

vectors are used for calculating the reference frame angles, giving the attitude matrix of the satellite. [22]

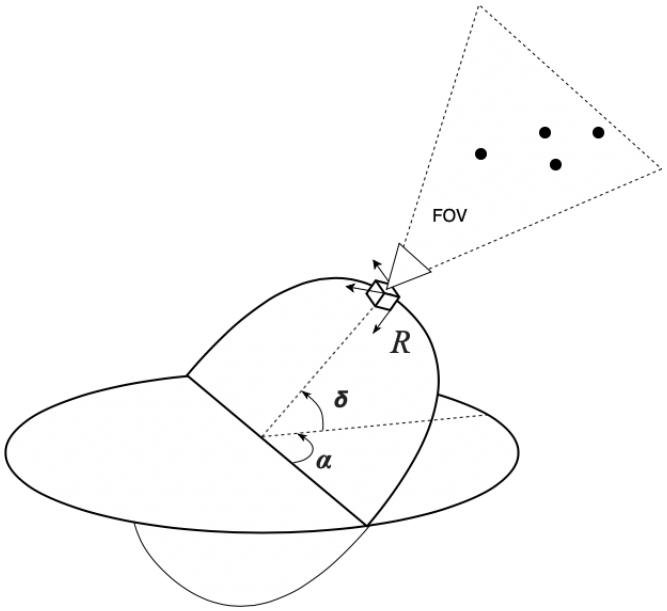

Figure 5.1: Diagram of star tracker equipped satellite in orbit. Attitude and position elements are described by following symbols; α being the right ascension, δ declination and R the attitude matrix. The star tracker camera system is visualized with four stars in the field of view (FOV).

A star tracker is an instrument that provides navigational data, such as position and attitude, by recognizing stars from optical images. The system consists of a camera and a computer. The system operates by capturing an image, processing it, and then comparing the extracted data to a star catalog. The star catalog contains columns of information about known stars. [21]

Star tracker systems depend on image processing and analysis that run on varying computing platforms. The platform and purpose of the star tracker in focus is an FPGA-based star tracker for spacecraft attitude determination. The general idea of star tracker based attitude determination is explored in this chapter to explain the context of a pre-processing pipeline for such a system.

5.1 Argumentation

There are multiple ways to determine the position and attitude of a spacecraft. When the accuracy of the attitude needs to be better than 0.1°, a star tracker is to be used. When the high accuracy of a star tracker is not needed, other methods, such as navigation satellite systems or momentum wheels, can be used for determining the attitude and position. Often however, a star tracker is used in combination with other attitude and position determination systems to reinforce the estimates.

Other than accuracy advantages, the star tracker is also more independent than navigation satellite systems and mechanically simpler than momentum wheels. On the other hand, star trackers require more complexity in a computational sense. [21]

A unique feature of star trackers is the ability to find the attitude and position of a spacecraft without any previous knowledge, also known as Lost-In-Space attitude determination. Momentum wheel-based navigation, for example, requires a reference to a previous position to determine its current position. The Lost-In-Space feature is also available using navigation satellite systems, although with limited range. [21]

5.2 Image acquisition

Images are usually captured by a charged-coupled device, or CCD, sensor. In contrast to another common image sensor type, the CMOS sensor, CCD is more resistant to noise, resulting in higher quality images. [21]

Star tracker cameras use a fixed focal length lens and a large field of view (FOV) to capture as many stars in the image as possible. A large FOV camera system consists of a large image sensor and a lens with small focal length. To maximize the brightness of the stars, the lens is equipped with a wide aperture for allowing more light to enter the image sensor. The camera system FOV is visualized in figure 5.1. [21]

Star tracker camera sensors generally use the visible light spectrum to capture images. The images are stored in single channel, or black and white, format. Any color information is useless in star trackers, which benefits from high contrast and brightness of image features. This makes the image features more likely to be extracted and used for star detection. [21]

Star trackers in low-earth orbit satellites are faced with challenges due to the moon, earth, and sun. Images containing these objects are often overexposed, and unusable for the star tracker. To overcome these problems, a satellite can be fitted with two cameras, oriented 90°apart. Another option is to mount the camera on a side facing away from bright objects, and maneuvering the spacecraft to maintain the orientation. Other objects such as nearby satellites can also cause overexposure. The effect of these type of objects are less predictable, making it more difficult to mitigate the problems caused by them. [21]

5.3 Image processing

The image processor of a star tracker extracts data from the captured image to use for star detection. The extracted data and processing algorithms vary between implementations, but the general idea is that the extraction process enhances the image features, scans the image for targets, and stores information about the targets for star detection.

At the image processing stage, issues such as overexposed images and false star extractions are encountered. In worst case, this leads to halting the star tracker, losing the currently prepared attitude and position update. In best case, the issue can be corrected and the currently prepared update can be resumed.

The ideal output of the image processing is to have at least four bright stars extracted from the background. Using the relative position of each extracted star, the star tracker system is able to proceed to star detection. The image data is only used up to this stage, making it unnecessary from this point forward.

5.4 Star detection

The identification of stars is generally carried out by comparing identifiers, consisting of multiple stars. The identifiers are created by calculating the angular separation between bright stars found in the image. The angular separation can be determined relatively easily with a fixed FOV camera. [21]

The angular separation between any two stars is generally a unique identifier. This makes the process of star detection essentially a look up operation, where known angular separations are compared against the identifiers created from the captured image. [21]

Four stars are used for creating the identifiers. The first two can be used as unit vector, while the third is required to resolve ambiguity. A 4th star is required for positive identification. [21] [5]

Given a set of recognized stars, or identifiers, the attitude and position can be calculated. The error detection and correction in this step is crucial, but somewhat easier than up to this point. False positives, as in stars that are mistakenly recognized can be voted out when multiple stars has been matched. False positives are recognized by verifying their position in relation to other recognized stars. False negatives are more difficult to detect, and not a problem as long as there are enough recognized stars.

5.5 Attitude and position

The attitude and position are determined based on the recognized stars. With the known coordinates of the stars, the position and attitude of the satellite are calculated with the help of linear algebra. Finding the attitude can be generalized to solving to the Wahba problem, where a rotation matrix between two coordinate systems are sought. By solving this problem, for example with singular value decomposition, the attitude matrix of the satellite can be obtained. To maximize the accuracy of the attitude estimation, the stars in the largest identifiers in the image are used for the matrix calculations.[22] [5]

The position of the satellite can be estimated using trigonometric calculations using the recognized stars. The known positions and separation angles of the stars, together with the camera system FOV parameters, enables the satellite position to be calculated in relation to the stars. The position estimation is also reinforced with the help of multiple results. [23]

6. Pre-processing pipeline documentation

The goal of the documentation of the pre-processing pipeline is to explain the features and technical details of the implemented system. Each feature contributes to filling the requirements stated in the system specification. Figure 6.1 shows an overview of the final product which includes the required components.

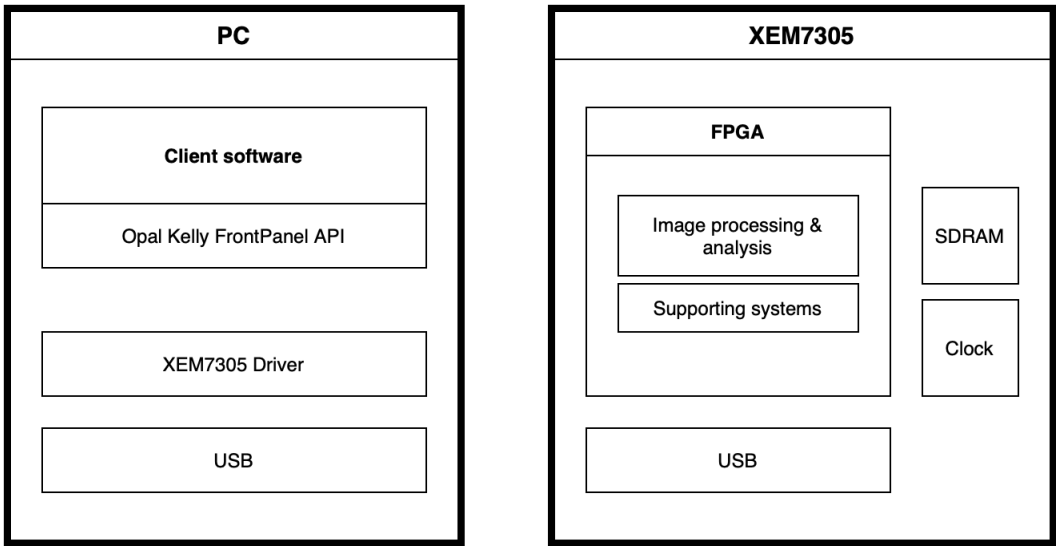

Figure 6.1: Diagram of star tracker pre-processing pipeline. The FPGA, performing the image processing and analysis, on the right and the PC, being the user interface, on the left.

6.1 Functional overview

The star tracker pre-processing pipeline is a system that uses an input image to extract features from. The features, in this case stars, can be used for further

implementation of a star tracker.

The system can be integrated to a completely embedded environment, where it does not depend on a user interfacing client. An integration to a star tracker can be independent and highly isolated, with minor adjustments made to the off-chip module interfaces. Only a few essential parameters, triggers, and data are required.

From a user experience point of view, the pre-processing pipeline works by powering on the FPGA, then connecting it to the pre-processing pipeline client. The client then programs the FPGA with the pre-processing pipeline FPGA design. After successfully programming the FPGA, the pre-processing pipeline is configured with a valid image and parameters. The FPGA executes the image processing and analysis after which the client reads the result, consisting of an image and metadata, for the user to evaluate.

From an integrated system point of view, the pre-processing pipeline is powered on and configured by an external system, for example, a program on a microprocessor. The external system has the same control and responsibility as the client software. To make the integrated system autonomous, a more sophisticated parameter tuning is needed for the pre-processing pipeline to be effective. This is desirable when integrating a system to a spacecraft, since it often requires a high degree of autonomy.

6.2 FPGA design

The FPGA design is modeled with Verilog HDL. Xilinx Vivado design suite is used for generating bitstream files that are programmed to the Opal Kelly XEM7305 FPGA module.

The star tracker pre-processing pipeline FPGA design contributes to all system requirements in Table 3.1. The requirements are satisfied by combining a set of components. Some components contribute to several requirements, while others are less multipurpose. Figure 6.2 shows an overview of the components in the design.

6.2.1 Support systems

The supporting systems included in the FPGA design consist of components that do not contribute to image processing and analysis directly, but provide other

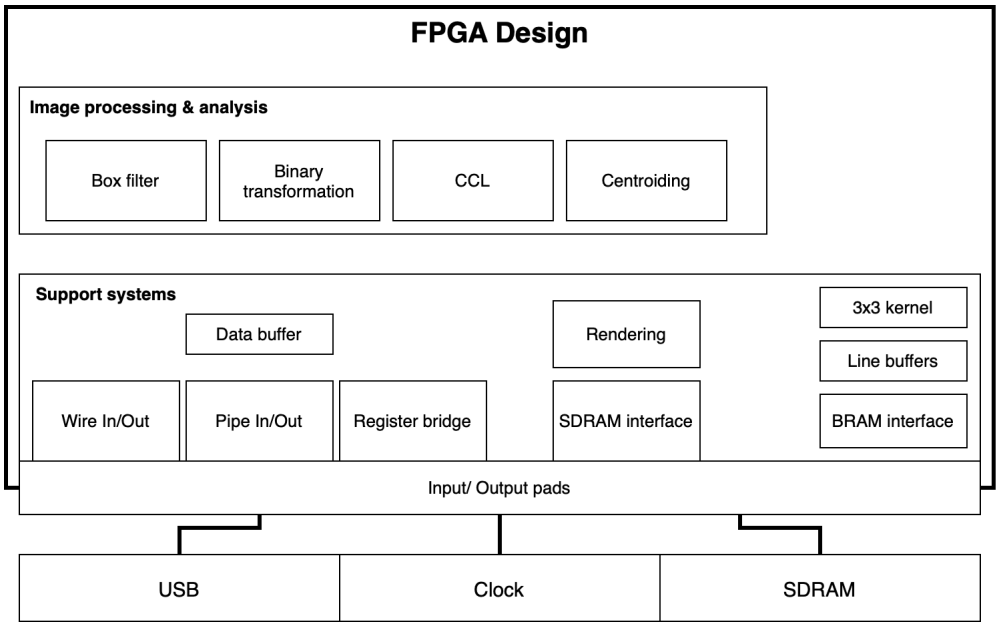

Figure 6.2: Diagram showing FPGA design as functional blocks where top section is furthest from off-chip modules while bottom section is closest to off-chip modules. Boxes in the diagram are not in scale to the actual footprint of the components on the FPGA.

necessary features, such as storage and communication.

The core supporting systems of the FPGA design consist of modules provided by Opal Kelly. These include communication, buffering, and storage solutions that are specific to the FPGA and off-chip components.

The bottom section of Figure 6.2 shows the main functional components of the support systems in the FPGA design. The isolation of this section can be made in a functional, and in some cases, technical sense.

Communication and host interface

The Opal Kelly host interface, called okHost, is a gateway for controlling and monitoring the FPGA. It contains the logic for communication with the FX3 USB microcontroller, exposing signal endpoints for controlling USB communication. The host interface also contains a clock signal derived from the external clock module on the XEM7305 board. The 100.8 MHz clock signal synchronizes the logic in the FPGA design with the USB microcontroller operations. [6] [24]

Opal Kelly provides a set of IP core modules supporting data transfer from single bits to megabytes. Different synchronization and control methods are used

for each module, resulting in small scale data transfers being simpler than large scale transfers. Three Opal Kelly communication modules are used within the FPGA design, each with its own purpose.

The Opal Kelly register bridge connects the host interface directly to a dedicated address space on the FPGA. The register bridge is connected to a 32-bit wide address bus, with 32 bits of storage for each address to use synchronously. The register bridge interface contains separate 32-bit data buses for input and output, and a register address bus. This adds complexity to the FPGA design, because a mapping mechanism, for example a multiplexer, is needed to make all 32 addresses usable. The register bridge is used through the Opal Kelly API, in the client software. Figure 6.2 shows the register bridge connected to the input/output pads, which act as interface between the register bridge and the off-chip modules. The required off-chip modules for the register bridge are clock and USB. [24]

Wire In and Wire Out modules are directional 32-bit buses that transfer data asynchronously. The Wire In/Out modules are by their asynchronous nature, and limited capacity, good for state management and triggering events. The asynchronous data are nevertheless, in many cases, synchronized to a clock in the design. Figure 6.2 shows the Wire In/Out connected to the input/output pads, giving them access to the USB off-chip module. [24]

Pipe In and Pipe Out modules are used for multibyte synchronous data transfer. The data buses on both modules are 32 bit wide. The block throttled variant of the modules adds negotiation signals to the communication system, which allows the FPGA to control the flow of data. The PC can initiate the data transfer, but unless the `EP_READY` control signal is high, data are not sent. This is useful in cases where incoming data needs to be stored using buffers. Figure 6.2 shows the Pipe In/Out connected to the input/output pads, allowing it to interface with clock and USB off-chip modules. [24]

FPGA state management and event triggers

The Wire In module is used on the FPGA to enter states that need to be externally managed. Table 6.1 shows a description and wire values of them. The Wire In uses only the first four bits of the 32-bit data and the values are set so that only one state can occur at a given moment, which ensures the predictability of the system. Each state in Table 6.1 is mapped to an index, which is used to read the value corresponding to the state on the 32-bit Wire In data bus.

The Wire Out module has the single purpose of providing verification of the

setup of the FPGA. The signal is expected to be high after a small delay after the FPGA is powered on and the bitstream is programmed. Table 6.2 shows the description and wire value of the Wire Out state.

Wire value	**Description**
0x0001	Read storage to Pipe Out. Used when reading image back from the FPGA to the client.
0x0002	Write to storage from Pipe In. Used when writing an image from the client to the FPGA.
0x0004	Resets FIFO buffers. Used between image reads and writes.
0x0008	Triggers image processing and analysis. Used when the FPGA is configured and ready to perform star extraction.

Table 6.1: Description of Wire In state control bits.

Wire value	**Description**
0x0001	Memory calibration signal. Used to verify successful initialization of the SDRAM memory.

Table 6.2: Description of Wire Out state bits.

The Wire In/Out modules are a good suit for the state triggers because of the reactive and simple user experience. The state management and event trigger system contributes to satisfying the system requirements. The features provided by the Wire In module are fundamental to the requirements 1.3, 1.4, 2.2, and 3.3.

Parameter configuration and metrics

To allow data transfer for accessing register values on the FPGA, the Opal Kelly register bridge is used. The registers are addressed so that the first 15 contain writable data, and the second 15 contain readable data. This prevents any concurrency issues with register access. Pre-assigned values to the writable addresses ensure that the pre-processing pipeline parameters have default values to use in case none are assigned through the register bridge. Tables 6.3 and 6.4 explain the use of the register bridge.

The features enabled by the register bridge for parameter configuration and metrics contribute to system requirements 1.2, 1.3, 1.4, 2.2, and 3.3.

Address	Data	Description
0x0000	32-bit int	Minimum brightness of an image feature. Used to filter out faint objects in the image.
0x0001	8-bit int	Binary transformation threshold. Used for filtering faint pixels out when preparing image for feature detection.

Table 6.3: Description of writable values used by the FPGA.

Address	Data	Description
0x000F	32-bit int	Image processing and analysis timer. Used for evaluating performance. Stored in register as clock cycle count.
0x0010	32-bit int	Box filter timer. Used for evaluating box filter algorithm performance.
0x0011	32-bit int	CCL timer. Used for evaluating CCL algorithm performance.
0x0012	32-bit int	Feature rendering timer. Used for evaluating performance of feature rendering.

Table 6.4: Description of readable registers on the FPGA.

Image transfer and SDRAM storage

The size and format of the pre-processing pipeline input image require some mechanisms to be implemented to allow transfer and storage of the image data. The image is fixed as to size and format, namely 270 x 480 pixels, single channel, and eight bits per pixel. The image is not compressed to decrease image processing complexity, making the total image size 129.6 kb.

The Pipe In module is used to receive the image data on the FPGA. The module is connected to, and partly managed, by a FIFO buffer. The buffer is also connected to the SDRAM interface, which pulls data from the buffer to storage. The entire data flow is thus controlled by the SDRAM, which is the bottleneck of the image transfer system.

The 32-bit wide data bus of the Pipe In module is connected to the input of the FIFO buffer. The buffer capacity is 128 bits to suit the input data bus width of the SDRAM interface. Both Pipe In and FIFO modules are synchronized to the host interface clock. The SDRAM interface is synchronized to its own 87 MHz

clock, also derived from the 200 MHz system clock. The modules are synchronized together with control signals to manage the data flow and to mitigate any issue with clock speed differences.

Beyond the input image, the SDRAM is used to store intermediate image processing results. The sequential process writes an image to storage after each step. Since each output is only needed as the input to the next step, the stored images can be overwritten. The process requires a storage capacity of three times the image size, one of the images being the original. The original image is kept in case the process is to be re-executed. In an integrated environment, the storage capacity could be scaled down to two times the image size, since the original image would be useless.

After image processing and analysis the output image is stored to SDRAM. The output image is transferred from SDRAM to an external endpoint using the Pipe Out module through another FIFO buffer. The image output transfer uses a similar FIFO buffer as the input image transfer, except with reversed port widths and other control signals.

The image transfer and storage systems contribute to system requirements 1.4 and 2.2.

Line buffer and BRAM storage

The image data are accessed by the image processing and analysis components through line buffers implemented as BRAM. Each line consists of 270 8-bit pixels, and three lines are stored in the buffer. This geometry enables image processing on both the horizontal and vertical axis of the image. Figure 6.3 shows the line buffer in the middle, as the intermediate storage mechanism.

The dual port BRAM has two data buses and two address ports to allow simultaneous read and write operations. The interface also includes a 270-bit data mask to select which pixels to write to BRAM. This adds complexity to the line buffer system but reduces the number of registers needed.

The line buffer system is the intermediate storage between the image storage and image processing. The read and write operations are triggered by counters that are updated during the image processing. The line buffer is filled by initiating a read operation to the SDRAM. After each burst read, 128 bits of data are pushed to the buffer and the SDRAM read address is updated. The read address is incremented by 8 between reads, since the burst mode of the SDRAM covers 8 addresses. The SDRAM read continues until the buffer is full.

The line buffer can be seen as a sliding window on horizontal axis of the image. The buffer, and image processing, is initialized by filling all three buffers. While processing the image, one line after another is read into the buffer. The image columns are mapped to BRAM using addresses and indices. The indices track which address points to which column, while the addresses are fixed variables to point to the start of an image column in BRAM. The addresses are rotated between the indices to predictably and efficiently use the BRAM address space.

A big portion of the logic in the pre-processing pipeline is for moving around image data, making the line buffers a core part of the support systems, and storage mechanisms in the FPGA design. The line buffers contribute to system requirements 2.2, 3.1, and 3.2.

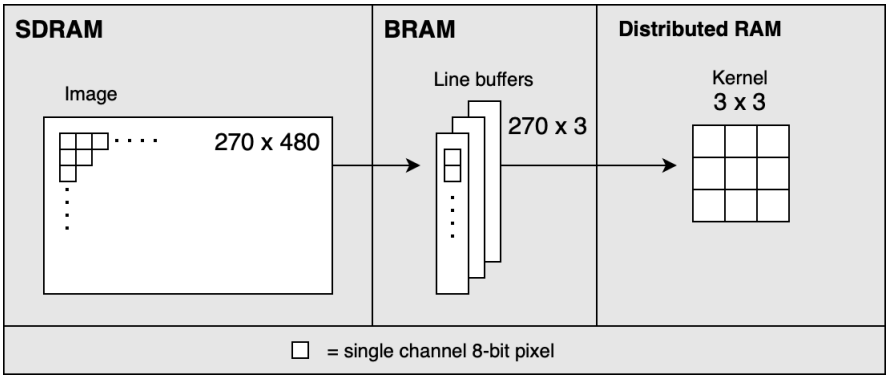

Figure 6.3: Diagram of different levels of memory used for storing image pixels of the star tracker pre-processing pipeline.

3x3 Kernel

The lowest level storage mechanism where image pixels are kept is 3 small line buffers, forming a 3 x 3 pixel sliding window, or image processing kernel. It is implemented in distributed memory, within the FPGA fabric. The kernel consists of three pixels from each column in the line buffer, creating the 3 x 3 pixel geometry shown in Figure 6.3. Data from the line buffer is fed into the kernel data structure which is then used for image processing and analysis. The results of these operations are written directly to a higher level storage. Thus the data flow is directional within this storage mechanism.

Between each processed pixel, the kernel moves one step. This movement consists of three read-operations from BRAM to the kernel, each line buffer contributing with three pixels. The entire kernel is overwritten each time, since the

data transfer is relatively low.

The image processing kernel contributes to system requirements 2.2, 3.1, and 3.2.

Rendering

The rendering process creates an image output from the data generated within image processing and analysis. The data generated from star extraction in the pre-processing pipeline is stored in tables and buffers for convenience. Rendering an image with this data provides visual feedback of the process.

Rendering relies on the kernel and line buffers to write the image data to SDRAM. The image is rendered in the original 270 x 480 single channel 8-bit format to SDRAM between the image processing steps, and for final output. The rendering process contributes to system requirement 2.2.

6.2.2 Image processing and analysis

To extract stars from an image, a number of image processing algorithms are implemented. The general idea is that an image containing bright stars in the foreground and faint objects or empty space in the background is used as input. The first algorithm smoothens the image to create larger and more detectable foreground features. The second process separates the foreground from the background with a threshold. This is to enable connected component labeling (CCL) in the next step, which requires a fully separate foreground and background. The connected component labeling algorithm traverses through the image pixels and assigns them to features. Metrics of the features are stored for calculating the centroid in the final step.

As a core part of the pre-processing pipeline, the image processing and analysis contribute to all system requirements in Table 3.1.

Box Filter

A box filter is applied on the image to prepare the image for CCL. The principle of the box filter is to expand and blur features in the image by replacing a pixel with the average of the surrounding pixels. The 3 x 3 kernel is used for this filter, where the center pixel is the target. This filter is commonly described by the expression

$x_{ij} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$, where x is the currently updated, center pixel in the kernel. The matrix represents the kernel with pixel values. This gives a uniformly applied average of the image where high value pixel areas are expanded without too much loss of detail. Figure 6.4 shows an example of a feature passing through the box filter. The blurring and expansion of the feature are apparent.

A problem with the box filter is the need for integer division. A division operation is problematic when implemented in hardware and is often dedicated a separate IP core module. This increased complexity can, however, be avoided in the case of the box filter because it lacks the need for high accuracy output of the operation, as the pixel values are 8-bit integers. The alternative operation to replace the division takes advantage of the constants in the box filter, which are the output bit width and the denominator. The lack of accuracy is compensated by the performance which is reduced from a multi clock cycle operation to a single clock cycle operation. The division is approximated with the expression $(\sum x_{ij} \cdot 28) \gg 8$, where x is the 3 x 3 kernel.

Image edges are handled by ignoring the pixel value and setting them to 0. This assigns the edges of the image as part of the background feature. The possible loss of edge features is also acceptable because of the difficulty to approximate the complete geometry and center of an edge feature.

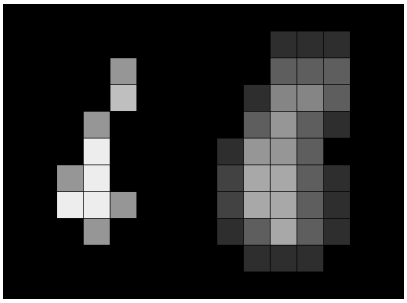

Figure 6.4: Illustration of box filter. The original feature is on the left, while the processed feature is on the right. Box filter averages every pixel within a 3 x 3 image processing kernel.

Thresholding

Thresholding is done to separate the background from the foreground in the image. The foreground needs to contain enough pixels to enable successful star extraction.

The box filter allows foreground feature edges to be brighter, which otherwise might be categorized as background in this step. If the threshold is too low, background and noise blend together with foreground features, making the choice of optimal threshold a complex problem. This is out of scope of this project and the threshold is manually set.

An example solution for setting the threshold could be to use a feedback loop to adjust the parameter. The loop would use previous output of the pipeline to use as input, providing the number of stars, brightness and distance to determine a threshold setting and acceptable result.

The thresholding process transforms each pixel value to either 0, being background, or 255 as foreground. Figure 6.5 shows an example of the thresholding process. The example shows most of the original pixels passing the threshold. These are assigned to foreground, with the value 255. Some pixels around the edges are too faint to pass the threshold. These pixels are assigned to background with the value 0.

As this is a process with no dependency on neighboring pixels, it can be conveniently placed in the output phase of the box filter. The output phase is where the result of the box filter operation is written to SDRAM memory. The register containing the output of the box filter is piped to the SDRAM write bus through the threshold. This makes the thresholding highly efficient because of no additional clock cycles required.

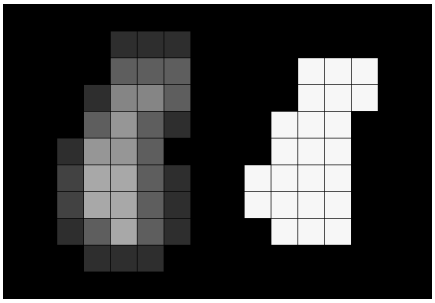

Figure 6.5: Illustration of thresholding process. The feature on the left is the product of box filter, and the feature on the right is the feature after passing the threshold.

Connected component labeling and analysis

The goal with CCL is to assign foreground pixels to features. The algorithm traverses the image using a kernel, updating the feature tables and pixels in the

line buffer. Three feature tables keep information about pixel values and the feature geometry. Figure 6.6 shows the three tables, populated with example data.

Every encountered foreground pixel is updated depending on its neighbors. The focus is on the center pixel in the kernel. It is updated depending on the neighboring pixels. If any neighboring pixel has a value with an entry in the feature table, the current pixel is updated and mapped to the same feature. The geometry of each feature is stored as minimum and maximum coordinates within the image. Also the brightness of each feature is stored. This is used together with the geometry for feature analysis.

The requirement for a component to be connected to a feature is that at least one side of the pixel touches the feature, making it a 4-connected feature. In contrast, an 8-connected feature accepts diagonally connected components. Diagonally connected components are not detected in this CCL implementation because the previously performed box filtering already exaggerates the component connections, filling the gaps between pixels.

The 4-connect comparison is done by comparing the top center and left center to the center pixel in the kernel. The movement of the kernel mitigates the need for comparing all 4 connections surrounding the center pixel in the kernel. This simplifies the design while keeping the line buffer mechanism uniform with the other parts of image processing.

Conflicts in the feature table are encountered when connected components refer to multiple features. This happens when the shape of the feature is unfavorable in regard to kernel movement and the component comparison process. A conflict is resolved by merging the connected features. This is done by merging the brightness of the features involved, updating the feature coordinates, and reassigning the feature label. Depending on the number of mergers, a single feature might be mapped to a number of pixel values. Figure 6.6 shows an example of a resolved conflict, where pixel values **0x04** and **0x05** are mapped to the same feature, **0x03**.

While traversing the image, CCL will enter one of three states for each foreground pixel, while background pixels are ignored. When encountering a foreground pixel in the left center, it can be assumed to belong to a feature. The state where left center is foreground is prioritized, because the feature it refers to has the earliest entry in the feature table. The feature with the earliest entry needs to be selected in order for the center pixel to avoid conflicts. The next

priority is to compare the top center pixel. This state updates the center pixel with the value of the top pixel. A new feature is created when neither the left center or top center pixel is in the foreground. This state adds an entry to the feature tables.

The implemented CCL algorithm is streamlined and simple, but not without drawbacks. The most significant limitation is the number of features that can be stored. In case all foreground features in the image are square, the maximum number of features is 253. This is more than enough for what a star tracker needs, but depending on the number of conflicts the number of available features is significantly lower. This limitation originates from the 8-bit pixel values stored in the feature tables and the lack of re-renders of the CCL image. With re-rendering implemented, the maximum would be guaranteed to be 253 features, instead of 253 as best case. The trade-off for choosing re-render to allow for more features would be a performance loss and a larger memory and FPGA design footprint.

Pixel Value	Feature Label
0x01	0x01
0x03	0x02
0x04	0x03
0x05	0x03

0x00	background
0xFF	foreground

Feature Label	Coordinates
0x01	(x_{min}, y_{min}, x_{max}, y_{max})
0x02	(x_{min}, y_{min}, x_{max}, y_{max})
0x03	(x_{min}, y_{min}, x_{max}, y_{max})

Feature Label	Brightness
0x01	[0x00 - 0xFFFFFFFF]
0x02	[0x00 - 0xFFFFFFFF]
0x03	[0x00 - 0xFFFFFFFF]

Figure 6.6: Example of CCL feature tables. Left table is where the mapping of pixel values to feature labels is stored. Labels are also renderable, as they are stored as 8-bit integers. Center table stores minimum and maximum coordinates of each feature in 35-bit integers. Right table stores feature brightness as 32-bit integers.

Feature detection and centroiding

Feature detection is the process of using the feature tables, shown in Figure 6.6, to create image data to indicate detected features. Centroiding gives a more accurate indicator of the detected features by further analyzing them. Feature detection

and centroiding are the final step of the pre-processing pipeline, resulting in a output image and data points for the extracted stars.

The feature detection process filters out features that are too faint or small. The threshold is specified by the parameter configuration system. Table 6.3 shows the register and format of the threshold. The threshold is compared with the brightness of each feature, which is calculated in the CCL process. An example of the feature brightness table is shown in Figure 6.6 on the right. When the brightness of a feature is below the threshold, it is discarded. The process then moves on to validate the next feature in the feature table.

Unlike other parts of the image processing and analysis, feature detection traverses the image one feature at a time. The sliding window favors processing that needs to scan the entire image, possibly applying calculations to every pixel. Feature detection is only interested in processing known coordinates of the image, making the sliding window unfavorable.

A rectangle can be imagined around a feature, given the maximum and minimum coordinates of it. The pixels within this rectangle are read directly from SDRAM to the feature detection process. This bypasses the commonly used sequential SDRAM read procedure. A more complex addressing scheme is required to obtain the correct pixels. Due to the SDRAM burst mode, the correct address needs to take into account both pixel location and burst read suitability. Then an offset value is calculated to access the correct value in the SDRAM read buffer.

The commonly used SDRAM write procedure is also bypassed by the feature detection process. The addressing scheme used for reading the feature pixels is also used for writing them back. The data mask in the SDRAM interface is used by this process to simplify the data flow. It ensures that only a selected part of the write buffer is written into SDRAM. Since the number of pixels updated by this process is relatively low, the absence of a data mask would result in the need of copying the read buffer, only for most of it to be copied back to the write buffer. The configuration of the data mask is relatively simple since the pixel offset is already calculated by the addressing scheme.

Depending on the size of a feature, the pixel values are stored in registers. Centroid calculation is performed on features of suitable size. Suitability comes from the number of registers required for storing all pixel data of the feature. The pixel values for centroid calculation could be stored in BRAM to allow larger features to obtain a weighted centroid.

A position estimate for every feature is calculated to achieve system require-

ments 3.1 and 3.2. The centroid is calculated in one of two ways, a weighted and non-weighted version.

The weighted centroid of a feature estimates the true center of it. The process takes into account pixel brightness values and distance from the average coordinate. An average coordinate is the value between the minimum and maximum coordinates, stored in the feature table. The process creates an integer value for every row and column of the feature using these calculations. Each pixel in the feature contributes with its brightness, while the column or row number contributes with the weight. The calculation can be described with the following expression $C_i = C_i + \texttt{0xffff} - F_{jk}|i - c|$, where C_i is the feature column register, `0xffff` is a constant for the weight, F_{jk} is a pixel value, and $|i - c|$ is the distance from the average coordinate.

A detail to notice in the weighted centroid calculation is the requirement of pixel brightness values. The image processing and analysis have until this point used the output of the previous step as input to the next step. The brightness data used for weighted centroid calculation comes from the output of the box filter. This image is used because the pixels have the same geometry as they have in the feature table, and contain pixel brightness information.

The weighted centroid calculation results in column and row registers which are used to determine and render the estimated center of the feature. The coordinate of the center, relative to the feature, is the indices of the column and row registers with the highest values.

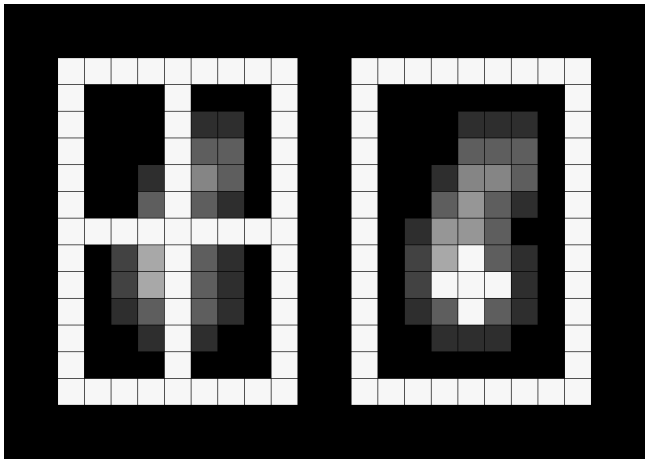

Figure 6.7: Illustration of the two centroid variants. Non-weighted centroid is rendered on the feature on the left. Weighted centroid is rendered on the feature on the right.

The non-weighted centroid calculation uses the minimum and maximum coordinates of each feature to calculate the average coordinate as the center. This estimation of a feature center is not as accurate as the weighted, but acts as fallback in case the weighted option is not suitable.

Figure 6.7 shows an example of the result of feature detection and centroiding. The detected features are rendered with a rectangular frame around the edges, with a small offset from the coordinates of the feature table, for better visibility. Both centroid variants are rendered as a cross on the corresponding feature. The non-weighted variant has a cross extending from the feature frame, while the cross of the weighted variant is significantly smaller. This helps with distinguishing the variants, and the visual analysis of the weighted centroid.

6.3 Client endpoint

The client endpoint is the external operator of the pre-processing pipeline and responsible for controlling the FPGA module. The features provided by the client endpoint contribute to system requirements 1.1, 1.2, 1.3, 1.4, and 2.2. The purpose of the client endpoint is to allow operation and development of the system. This is made easy by the use of Opal Kelly API, which makes controlling the FPGA module dynamic and simple. Optionally, the system could be more self-sustaining by handing over some client responsibilities to the FPGA module. A self-sustaining system might be ideal in an integrated system in production, but inconvenient for development and demonstration.

The client endpoint is a Java-based PC application with a graphical user interface (GUI) that enables debugging, analysis, and parameter configuration. The application uses the JavaFx platform, making it lightweight and portable.

6.3.1 Connecting and initializing FPGA

The base class, called `okCFrontPanel`, is the entry point for operating the FPGA module. It provides methods for device interaction, device configuration, and FPGA communication. The FPGA-module is initialized by calling the API to open a connection. This enables the communication layers required for interacting with the FPGA module. [24]

Before programming the FPGA, the device is configured with a default configuration from on-chip flash memory, and checked for errors. When encountering

errors, they are reported to the user, but no further action is taken.

6.3.2 Bitsream programming

The bitstream file, containing the FPGA-design, is programmed to the FGPA using the `okCFrontPanel` class. To program a bitsream file to the FPGA, an open and initialized device is required. When encountering errors in the programming stage, these are reported to the user, and a reset signal is sent to the FPGA. The memory calibration is initiated on the FPGA as it is programmed. The test result is verified by reading the Wire Out value from the FPGA, shown in Table 6.2. Errors encountered in this stage are simply reported to the user.

6.3.3 Pre-processing pipeline configuration and control

The configurable parameters and triggers are listed in Tables 6.1 and 6.3. To enable these features, a programmed device is required.

The binary transformation threshold and feature brightness threshold are read from user input. The values are validated and written to the register bridge on the FPGA using the `okCFrontPanel` class. Any errors are reported to the user.

To select an image for the pre-processing pipeline, it is first loaded to the client application. The 270 x 480 resolution, 8-bit single channel bitmap file format is used. The selected image can be viewed in the GUI, but this feature is most useful for analysis after executing the pre-processing pipeline. The image is prepared by converting it to a byte array of integers. This is the format required by the Pipe In module. The FPGA is prepared for image transfer by configuring its state with the Wire In values listed in Table 6.1, which resets the FIFO buffers and sets the SDRAM to read mode. The write is then performed using the Block Pipe Out module, by calling the `okCFrontPanel` class.

The pre-processing pipeline can be executed when the parameters and image are configured. The process is started by setting the Wire In value, as seen in Table 6.1. This triggers the pre-processing pipeline on the FPGA. Error detection in this stage is difficult, and only detectable when the output image is viewed.

6.3.4 Results and analysis

The output image can be read from the FPGA when the pre-processing pipeline is completed. This is initiated in a similar manner as writing an image to the

FPGA, except for the Wire In value. After the FIFO buffers and SDRAM are reset, the image is read to the client by the Block Pipe In module, provided by the `okCFrontPanel` class. The image is saved to disk, in the same format as the input image.

The output image can be viewed after it is transferred from the FPGA-module to the client endpoint. Analysis of feature detection is made simple by having the input and output images side by side. The zoom and movement features enhance the user experience further. The detected stars are distinguished by the frame outline and centroid, as seen in Figure 6.7.

7. Results and comparison

7.1 Pre-processing pipeline results

The results of the pre-processing pipeline are presented as four use cases. The standard procedure of the pre-processing pipeline is to use an input file together with input parameters to produce an output. Each use-case contains of two runs, each consisting of input parameters, output image and metrics.

The data set consists of generated test data and image data from the StreakDet input dataset. The images from the Streakdet dataset are selected to highlight features of the pre-processing pipeline, such as thresholds and weighted centroiding.

Test input

A test input, shown in Figure 7.1, is used to visualize the standard procedure of the pre-processing pipeline. The data is synthesized by gathering an overview of the StreakDet dataset and creating a layout of pixels that resembles commonly seen stars.

To showcase weighted centroiding, dimensions and pixel value gradient of the stars are taken into account. The gradient helps with visually verifying the process, since the centroid location becomes more obvious.

Figure 7.2 shows the result when a foreground threshold of 50, and feature threshold of 300 is applied. All features are detected, and one gets a weighted centroid, seen in the top right corner. Feature 7.3 shows the result when increasing the foreground threshold to 100 and keeping the feature threshold at 300. Less pixels are assigned to foreground, making the features smaller. This enables more features to receive a weighted centroid.

The processing time for the test input runs are 0,1ms apart, as seen in table 7.1. The time difference originates from centroiding while other parts of the

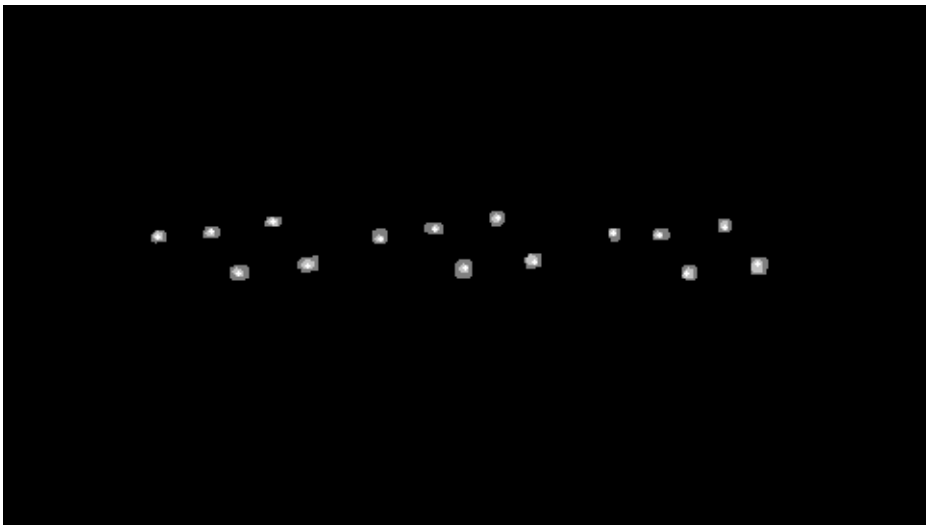

Figure 7.1: Synthesized star image used as test input.

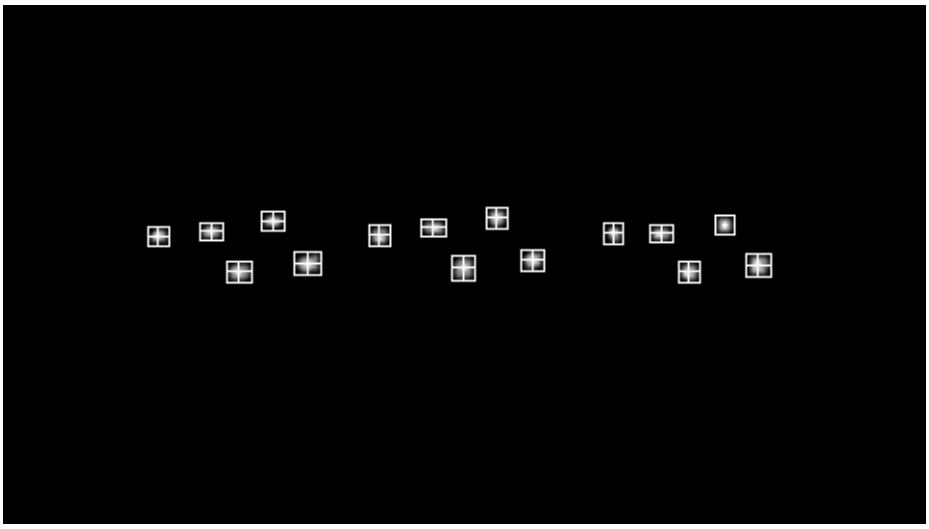

Figure 7.2: Run T1 output image.

Run ID	**Foreground threshold**[1]	**Feature threshold**[1]	**CCL**[2]	**Box filter**[2]	**Centroid**[2]	**Tot.**[2]
T1	50	300	39,39	18,84	0,83	59,06
T2	100	300	39,39	18,84	0,73	58,96

Table 7.1: [1]Input parameter to the pre-processing pipeline, specified as pixel value. [2]Performance metric of the pre-processing pipeline, measured in milliseconds. Data shows result of increasing the foreground threshold, as less time spent centroiding while other processes are unaffected.

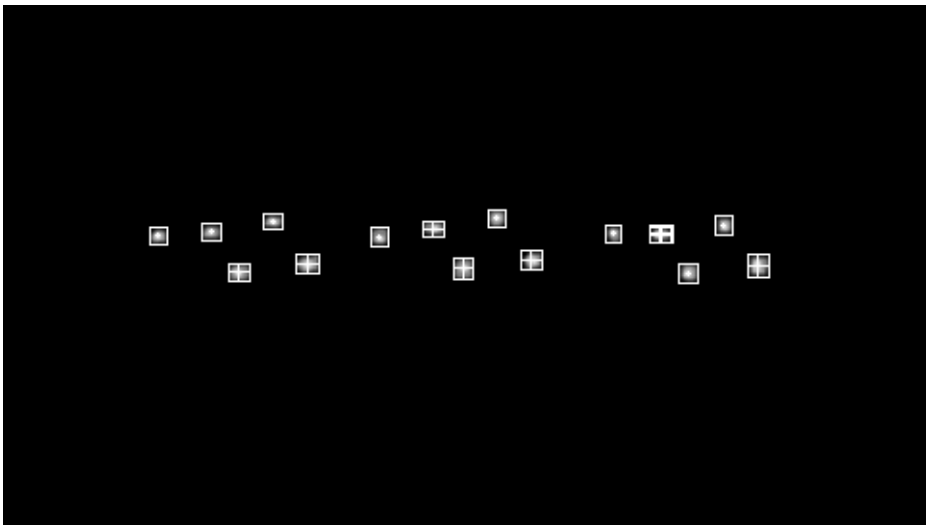

Figure 7.3: Run T2 output image.

pre-processing pipeline remains unaffected by the foreground threshold increase. Since less pixels are processed in run T2 faster processing is expected. The increased number of weighted centroid calculations did also not contribute to any performance loss.

Sample 1

Figure 7.4 shows the input image containing a moderate number of stars with mixed brightness levels. In contrast to the previously reviewed test input, this image consists of a realistic scenario of what an onboard image sensor captures. Also notable is that this is a good sample for star extraction and detection since there are numerous stars that are easily separated from the background and surrounding stars.

The results show that all extracted stars are within limits for weighted centroiding. After run S1.1 the feature threshold is decreased from 800 to 400 to extract more stars. This parameter change to increase star extraction is unsuccessful since Figure 7.6 shows the same number of extracted stars as Figure 7.5.

A 0,12ms difference in processing time between run S1.1 and S1.2, caused by the change in feature threshold, is observed in Table 7.2. Compared to the test runs, shown in Table 7.1, no change can be seen in box filter processing time. The time difference in CCL is likely explained by the lower number of stars extracted, requiring less computation and rendering.

Figure 7.4: Sample 1 input image.

Figure 7.5: Run S1.1 output image.

Run ID	**Foreground threshold**[1]	**Feature threshold**[1]	**CCL**[2]	**Box filter**[2]	**Centroid**[2]	**Tot.**[2]
S1.1	50	800	39,38	18,84	0.19	58,41
S1.2	50	400	39,38	18,84	0.31	58,53

Table 7.2: [1]Input parameter to the pre-processing pipeline, specified as pixel value. [2]Performance metric of the pre-processing pipeline, measured in milliseconds. Data shows increase in centroid time when decreasing feature threshold.

Sample 2

Sample 2 input image has a few stars that stands out with their brightness and dimensions, expected to be extracted. An anomaly caused by the camera system

Figure 7.6: Run S1.2 output image.

or an optical artifact, is seen in on the left side of the image, shown in Figure 7.7. The pre-processing pipeline extracts the entire feature, including the anomaly, as seen in Figure 7.8 and 7.9. In a star tracker this extraction is not valid since it gives a false representation of the celestial body in question.

Run S2.1, shown in Figure 7.8, contains four extracted stars using a foreground threshold of 40 and feature threshold of 800. This would otherwise be sufficient for star detection but since one of the stars is a anomaly the result is not reliable.

Run S2.2, shown in Figure 7.9, contains in addition to the four in S2.1, three additional stars. With a decrease of foreground threshold of 25%, from 40 to 30, the number of extracted stars increased by 75%.

In the case of run S2.2 the 0,8ms longer processing time, shown in Table 7.3 compared to S2.1 is necessary to increase the quality of star extraction to an acceptable level.

Run ID	**Foreground threshold**[1]	**Feature threshold**[1]	**CCL**[2]	**Box filter**[2]	**Centroid**[2]	**Tot.**[2]
S2.1	40	800	39,39	18,84	0.12	58,35
S2.2	30	800	39,39	18,84	0.2	58,43

Table 7.3: [1]Input parameter to the pre-processing pipeline, specified as pixel value. [2]Performance metric of the pre-processing pipeline, measured in milliseconds. Data shows that decreasing foreground threshold increases the processing time.

Figure 7.7: Sample 2 input image.

Sample 3

The Sample 3 input image, shown in Figure 7.10, contains a cloud of faint and small stars. By tuning the input parameters correctly, this proves not to be any obstacle for star extraction as the S3.1 output image shows in Figure 7.11.

To extract the best selection of stars as detection candidates, the parameters can be tuned further. Since the dimensions of the extracted stars are relatively small, an increase in foreground threshold is ruled out due to the effect seen in previous results, for example test input run T1. The feature threshold is thus increased to extract the brightest stars, keeping dimensions and centroid precision. The result of this parameter tuning for S3.2 is shown in Figure 7.12. The increase in feature threshold in run S3.2, decreases the processing time by 0.2ms, as shown in Table 7.4.

A notable aspect of the results is that the rendering of the centroid of a few stars are not visible. This is likely due to the small dimensions of the features, or rounding error of the weighted centroid calculation.

7.1.1 Conclusion

To summarize the results, the star tracker pre-processing pipeline works as intended, extracting stars from the input images. The procedure of the image processing system is clearly seen, when comparing input images to the output. The box filter, for example, is clearly visualized in by the output of the test run,

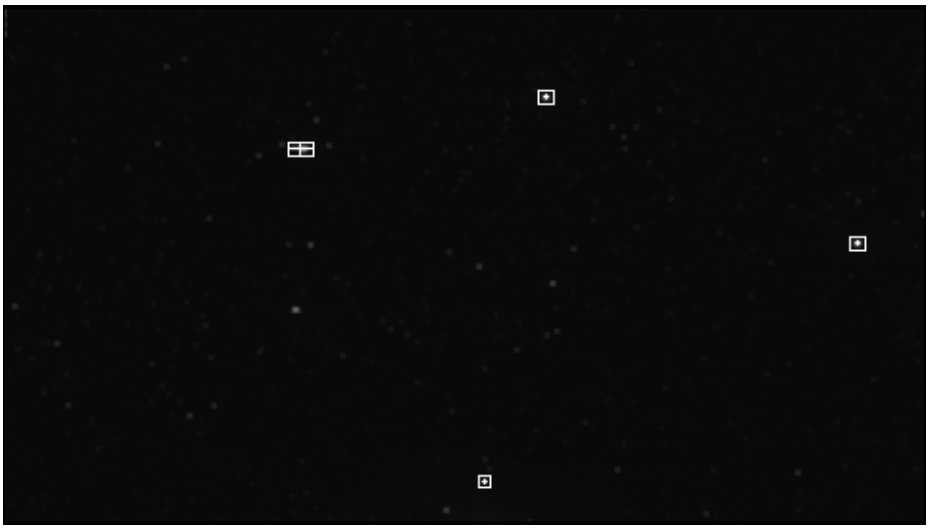

Figure 7.8: Run S2.1 output image.

Run ID	**Foreground threshold**[1]	**Feature threshold**[1]	**CCL**[2]	**Box filter**[2]	**Centroid**[2]	**Tot.**[2]
S3.1	15	50	39,39	18,84	0.3	58,53
S3.2	15	100	39,39	18,84	0.1	58,33

Table 7.4: [1]Input parameter to the pre-processing pipeline, specified as pixel value. [2]Performance metric of the pre-processing pipeline, measured in milliseconds. Data shows that increasing feature threshold decreases processing time.

seen in Figure 7.2. The sample images, such as sample 3 shown in Figure 7.10, benefits more clearly from this step of the pre-processing pipeline by expanding the faint and small stars captured by the camera system as they are noticeably larger, and clearly extracted in the output seen in Figure 7.11.

The more complex CCL procedure is visualized by the frame around the stars, which is enabled by the algorithm successfully identifying the connected pixels. As the most critical part of the pre-processing pipeline, the lack of issues within the procedure is a priority. In addition to confirming the CCL results visually, the output metrics also indicates reliable results, with consistent performance.

Weighted centroiding is also proved to work according the specification, shown by the results such as in Figure 7.3. The dimension limitation of the stars enabling weighted centroiding is a drawback of the procedure. All input images, however, contain stars with dimensions fitting to the limits, making the procedure acceptable for star detection.

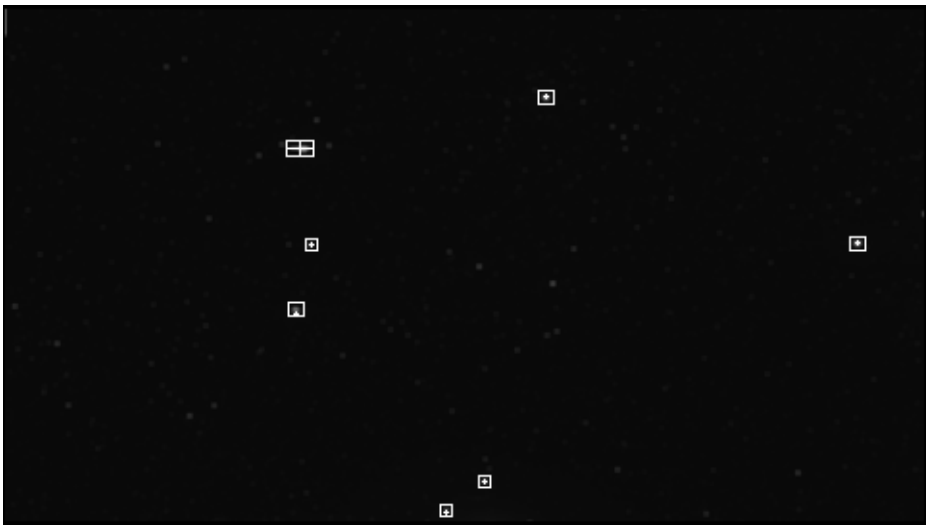

Figure 7.9: Run S2.2 output image.

The effect of the input parameter tuning is observable by the output images and metrics. An increase in foreground threshold decreases the number of pixels included in a feature. This helped with features gaining a weighted centroid. This effect is shown in Figures 7.2 and 7.3.

A decrease in foreground threshold is observed to increase the number of extracted stars, as shown in Figures 7.8 and 7.9.

The feature threshold shows a similar effect, without the loss of pixels in the features. This effect is shown in Figures 7.11 and 7.12.

Improvements based on these results can be made towards anomaly detection and centroid rendering. The results also reinforces the justification for a autonomous parameter tuning, with small adjustments leading to significant changes, as seen in Table 7.3 and Figures 7.8 and 7.9.

Regardless of minor drawbacks and improvement points, the pre-processing pipeline is proven by the results to perform the intended task with a stable performance.

7.2 Pre-processing pipeline performance

The performance of the pre-processing pipeline can be measured in the same way as any algorithm. The implemented performance measurement metric is execution time. Compared to a software based approach, it can be expected that the variation in execution times is smaller in a hardware based approach

Figure 7.10: Sample 3 input image.

since there is no abstraction such as an operating system between the task and computation element. Otherwise, a software based approach executed on a up-to-date CPU with multi-gigahertz clock speed is expected to perform the task in less time. A software based equivalent of this FPGA-based approach would be a aerospace-grade embedded processor. An embedded processor would also be less performant than modern PC CPUs, in terms of execution time.

The execution time comparison between the FPGA and software based approach is executed on a 10th generation intel i5 CPU.

7.2.1 Software based approach

To allow direct comparison between the FPGA-design and a software based counterpart, a test bench in C++ was developed. The software test bench contains same features, regarding image and parameter input, as well as output metrics, as the FPGA-design [25].

7.2.2 CCL algorithm comparison

The CCL algorithm has many implementations, mostly within computer vision applications. There are thus many references that can be used for comparison. The implementa The most complex part of the image processing in the pre-processing pipeline is the CCL algorithm, making it a g

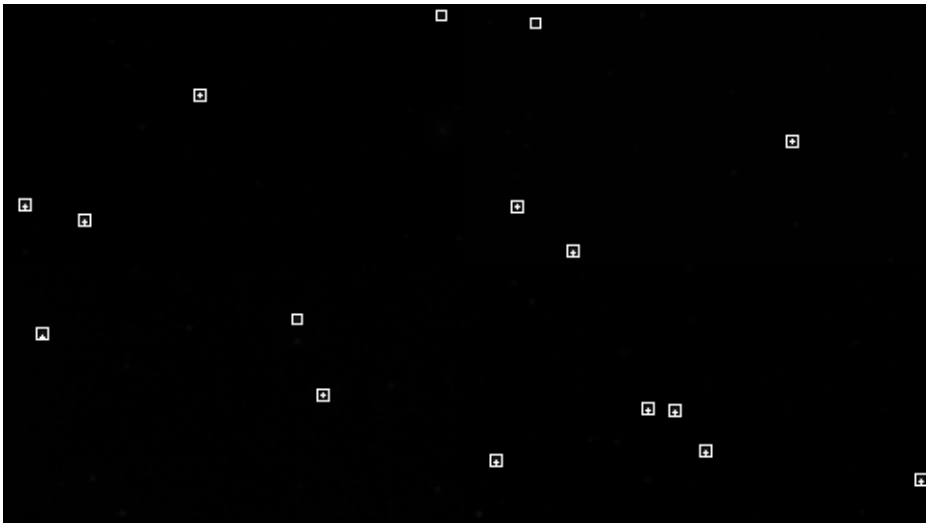

Figure 7.11: Run S3.1 output image.

Run ID	**Foreground threshold**[1]	**Feature threshold**[1]	**CCL**[2]	**Box filter**[2]	**Centroid**[3]
T1	50	300	55,3	12,1	149,1
T2	100	300	55,4	12,6	129,6
S1.1	50	800	53,4	12,6	76,1
S1.2	50	400	52,4	13,2	71,8
S2.1	40	800	55,7	12,5	78,7
S2.2	30	800	55,2	11,7	69,2
S3.1	15	50	53,6	12,2	83,1
S3.2	15	100	53,5	12,5	73,6

Table 7.5: [1]Input parameter to the pre-processing pipeline, specified as pixel value. [2]Performance metric of the pre-processing pipeline, measured in milliseconds. [3]Performance metric of the pre-processing pipeline, measured in microseconds.

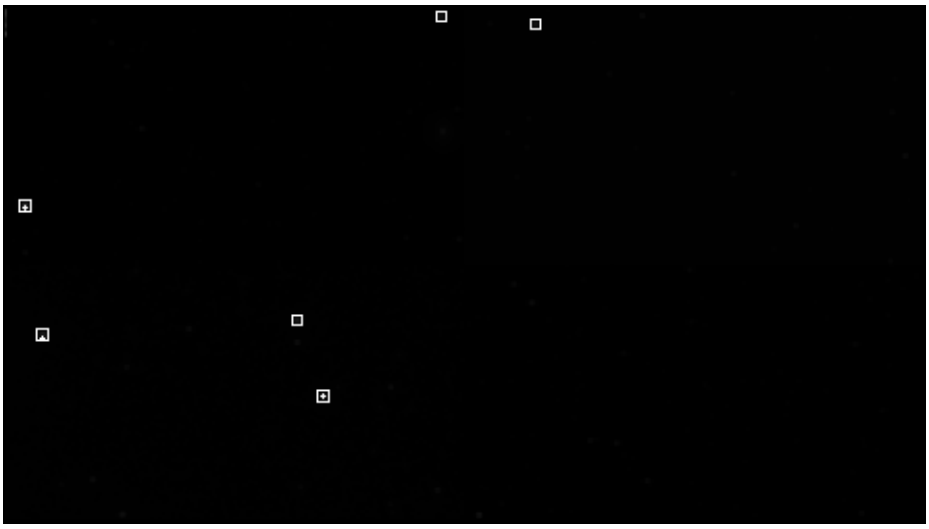

Figure 7.12: Run S3.2 output image.

8. Review and reflection

In this thesis a star tracker pre-processing pipeline has been developed with a technology used on the forefront of image processing and computer vision in general. As the pre-processing pipeline is essentially a sequence of image processing algorithms, the implementation on an FPGA-platform is a considerable option when designing a star tracker system. This thesis gives an overview of the required components and technologies to develop such a system. Beyond implementing the star tracker pre-processing pipeline in a star tracker system, this study gives a complete reference from digital circuitry to system level design, to help with comparing other platform options.

8.1 System requirements

The gap between a functioning algorithm and its implementation in an FPGA design regarding technology is vast. The challenge in implementing a pre-processing pipeline to an FPGA-design is divided between solving problems within development of the image processing algorithm and computer hardware design practices. This resulted in some compromises and recommendations in respect to the system requirements in Table 3.1.

Implementation of requirement 3.2, centroiding of stars, could be revised to increase flexibility regarding feature size. The development was focused on the computation of the weighted centroid, leaving the operational functionality as a lower priority.

Requirement 3.3 involving error handling, could be implemented more thoroughly in the FPGA design. Even though error handling is enabled by exposing parameters in the pre-processing pipeline, some error correction could also be implemented.

8.2 FPGA design

The FPGA design is reviewed based on reports generated by Vivado. Timing and power constraints are commonly placed on FPGA designs to meet certain requirements. The focus in this thesis is to implement the pre-processing pipeline, giving less attention on meeting constraints. Optimizations to meet power consumption and timing requirements can be developed based on the current state of the design.

The reports generated by Vivado for this review focus on the resource utilization shown in Table 8.1 and Figure 8.1.

LUT	SLICEM	SLICEL	DSP blocks	BRAM Arrays
24131	2046	4690	5	142

Table 8.1: FPGA resource utilization. The number of LUTs, slices, DSP blocks and BRAM arrays are listed.

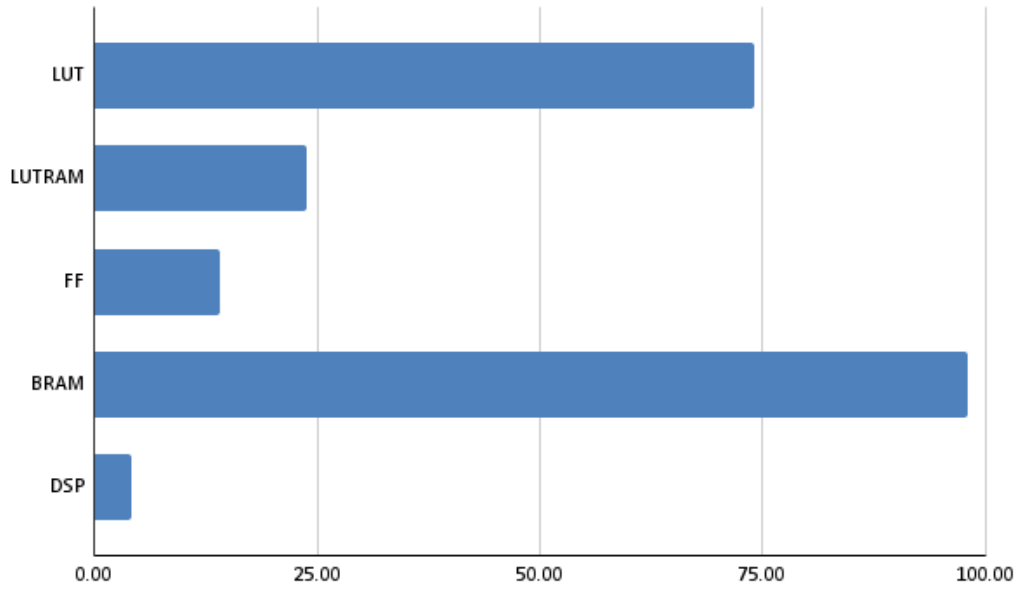

Figure 8.1: Utilization % of available FPGA resources. Total LUT utilization and LUTs used as storage mechanism, LUTRAM, are shown. Together with flip flop utilization, the internal utilization of CLBs is described. The more specialized resources, BRAM and DSP's, are also reported.

As seen in Table 8.1, the size of the FPGA design is reported to be 24131

LUT's, and 9242 flip flops, contributing to the 83% slice utilization on the FPGA. The ratio between flip flops and LUTs is large due to the nature of the CLBs in Xilinx 7 series FPGAs, where each CLB contains more flip flops than LUTs.

In addition to LUTs, Figure 8.1 shows that also BRAM utilization is high, at almost 100%. A part of the BRAM is utilized by the FIFO buffer used for image transfer, while the remaining parts are used by the line buffers. The FIFO buffer uses 38Kb BRAM arrays and the line buffer uses 18Kb arrays. The line buffer uses the 18Kb arrays to enable the multiport feature of the storage mechanism.

The high utilization means that the FPGA part used for this design is at the limit, making it a good fit for the current state of the design. It also means that adding features would be difficult without additional resources. An integration of the pre-processing pipeline thus requires a larger FPGA with more resources.

The low level storage mechanisms have a low utilization compared with the BRAM and overall LUT utilization. This can be seen in Table 8.1, where the count of SLICEM-type slices is relatively low compared to SLICEL. This correlates highly with the LUTRAM utilization in Figure 8.1.

The low utilization of DSP blocks is not unexpected because the parts of the design that most likely use these blocks are not parallelized. The pre-processing pipeline performance could be increased with higher utilization of the DSP blocks, the penalty being a higher power consumption.

The conclusion of the FPGA design based on the reports generated by Vivado is that the high degree of utilization of resources prohibits adding features requiring a large amount of LUTs or BRAM on the same FPGA. Optimizing the design to use more DSP blocks could release more LUTs to use.

Bibliography

[1] Neil C. Thompson and Svenja Spanuth. The Decline of Computers as a General Purpose Technology. Visited 20.1.2021, 2018.

[2] Xilinx Inc. *SDSoC Environment Profiling and Optimization Guide*, 2018.

[3] Xilinx Inc. *7 Series FPGAs Configurable Logic Block*, 2016.

[4] Mahendra Pratap Singh and Manoj Kumar Jain. Evolution of Processor Architecture in Mobile Phones. *International Journal of Computer Applications*, 90:35, 2014.

[5] Jenni Virtanen, Jonne Poikonen, Tero Säntti, Tuomo Komulainen, Johanna Torppa, Mikael Granvik, Karri Muinonen, Hanna Pentikäinen, Julia Martikainen, Jyri Näränen, Jussi Lehti, and Tim Flohrer. Streak detection and analysis pipeline for space-debris optical images. *Advances in Space Research*, 57:1607–1623, 2016.

[6] Opal Kelly Inc. *XEM7305*, 2021.

[7] Robin Mitchell. DRAM, SRAM, FLASH, and a New Form of NVRAM: What's the Difference? Visited 5.10.2021, 2020.

[8] Wim Roelandts. 15 Years of Innovation. *XCell*, 32:2, 1999.

[9] Investor Overview. https://investor.xilinx.com/static-files/2958145c-e3a2-456f-a461-bc9cba375af3. Visited 19.4.2021.

[10] Clive Maxfield. FPGAs from A to Z part 1. Visited 19.10.2021, 2006.

[11] Xilinx Inc. *Vivado Design Suite User Guide*, 2021.

[12] DJ Holding. *Electrical Engineer's Reference Book (Sixteenth Edition)*. Aston University, 2003.

[13] Eli Greenbaum. Open source seminconductor core licensing. *Harvard Journal of Law & Technology*, 25:132, 2011.

[14] Do-254 explained. Technical report, Cadence, 2019.

[15] Aboa Space Research Oy. Starmatch, 2020.

[16] Cypress Semiconductor. *EZ-USB FX3 Technical Reference Manual*, 2019.

[17] Avinash Aravindan. Flash 101: NAND Flash vs NOR Flash. Visited 5.10.2021, 2018.

[18] Micron Technology, Inc. *General DDR SDRAM Functionality*, 2001.

[19] Micron Technology, Inc. *DDR3L SDRAM*, 2011.

[20] Xilinx Inc. *7 Series FPGAs Memory Resources*, 2019.

[21] George Sebestyen, Steve Fujikawa, Nicholas Galassi, Alex Chuchra. *Low Earth Orbit Satellite Design*. Springer, 2018.

[22] Sam Pedrotty, Ronney Lovelace, John Christian, Devin Renshaw, and Grace Quintero. Design and performance of an open-source star tracker algorithm on commercial off-the-shelf cameras and computers. In *AAS 20-028*, page 7, 02 2020.

[23] Serkan Dikmen. Development of star tracker attitude and position determination system for spacecraft maneuvering and docking facility. Master's thesis, Luleå University of Technology, Sweden, 2016.

[24] Opal Kelly Inc. *FrontPanel SDK*, 2016.

[25] Oscar Björkgren. C++ Image processing test bench software. https://github.com/OscarBj/cpp-image-processing-testbench.