

Universidad Autónoma de San Luis Potosí
Facultad de Ingeniería
Centro de Investigación y Estudios de Posgrado

Desarrollo de Sistemas de Cómputo Heterogéneo Basados en el Algoritmo de Búsqueda de Abejas para el Seguimiento de Objetos en Video

T E S I S

Que para obtener el grado de:
Doctor en Ciencias de la Computación

Presenta:
Oscar Ernesto Pérez Cham

Asesor:
Cesar Augusto Puente Montejano

Coasesor:
Carlos Soubervielle Montalvo



San Luis Potosí, S. L. P. Agosto de 2021

Universidad Autónoma de San Luis Potosí
Facultad de Ingeniería
Centro de Investigación y Estudios de Posgrado

Desarrollo de Sistemas de Cómputo Heterogéneo Basados en el Algoritmo de Búsqueda de Abejas para el Seguimiento de Objetos en Video

T E S I S

Que para obtener el grado de:
Doctor en Ciencias de la Computación

Presenta:
Oscar Ernesto Pérez Cham

Asesor:
Cesar Augusto Puente Montejano

Coasesor:
Carlos Soubervielle Montalvo

San Luis Potosí, S. L. P. Agosto de 2021

Universidad Autónoma de San Luis Potosí
Facultad de Ingeniería
Centro de Investigación y Estudios de Posgrado

Development of Heterogeneous Systems Based on the Honeybee Search Algorithm for Video Tracking

T E S I S

Que para obtener el grado de:
Doctor en Ciencias de la Computación

Presenta:
Oscar Ernesto Pérez Cham

Asesor:
Cesar Augusto Puente Montejano

Coasesor:
Carlos Soubervielle Montalvo

San Luis Potosí, S. L. P. Agosto de 2021

UASLP
Universidad Autónoma de San Luis Potosí

FACULTAD DE INGENIERÍA

17 de junio de 2021

M.I. OSCAR ERNESTO PÉREZ CHAM
P R E S E N T E.

En atención a su solicitud de Temario, presentada por los **Dres. Cesar Augusto Puente Montejano y Carlos Soubervielle Montalvo** Asesor y Coasesor de la Tesis que desarrollará Usted, con el objeto de obtener el Grado de **Doctor en Ciencias de la Computación**, me es grato comunicarle que en la sesión del H. Consejo Técnico Consultivo celebrada el día 17 de junio del presente año, fue aprobado el Temario propuesto:

TEMARIO:

"Desarrollo de Sistemas de Cómputo Heterogéneo Basados en el Algoritmo de Búsqueda de Abejas para el Seguimiento de Objetos en Video"

1. Introducción
2. Marco Teórico de las Ramas de Computación Relacionadas
3. Diseño de Autómata para el Algoritmo de Búsqueda de las Abejas
4. Desarrollo de un Sistema Heterogéneo CPU-GPU
5. Desarrollo de un Sistema Heterogéneo SoC-FPGA
6. Discusión y Conclusiones

   Referencias

"MODOS ET CUNCTARUM RERUM MENSURAS AUDEBO"

A T E N T A M E N T E

UNIVERSIDAD AUTONOMA
FACULTAD DE INGENIERIA
DIRECCION

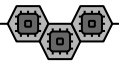
DR. EMILIO JORGE GONZÁLEZ GALVÁN
DIRECTOR

Copia. Archivo.
*etn.

www.uaslp.mx

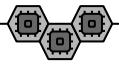
Av. Manuel Nava 8
Zona Universitaria • CP 78290
San Luis Potosí, S.L.P.
tel (444) 826 2330 al 39
fax (444) 826 2336

Acknowledgments



Gratitude towards thesis advisors Dr. Cesar Augusto Puente Montejano and Dr. Carlos Soubervielle Montalvo; members of the thesis committee Dr. Alejandra Guadalupe Silva Trujillo, Dr. Marcela Mejía Carlos and Dr. Ruth Mariela Aguilar Ponce; as well as research stay overseer and guest reviewer Dr. Gustavo Olague Caballero for their academic contribution and supervision during the development process of this thesis project and related products. This research was financially supported by CONACYT (Consejo Nacional de Ciencia y Tecnología) Ciencia Básica.

Contenidos



1 Introducción **1**
- 1.1 Preguntas y Objetivos . . . 3
- 1.2 Contribuciones . . . 4
- 1.3 Organización . . . 5

2 Marco Teórico de las Ramas de Computación Relacionadas **6**
- 2.1 Autómatas y Lenguajes Formales . . . 6
  - 2.1.1 Máquinas de Moore y Mealy . . . 7
  - 2.1.2 Implementación Electrónica de Autómatas Finitos . . . 8
    - 2.1.2.1 Unidad de Control Microprogramada . . . 8
    - 2.1.2.2 Unidad de Control Cableada . . . 8
    - 2.1.2.3 Comparación de Métodos de Implementación . . . 9
- 2.2 Seguimiento de Objetos en Video . . . 9
  - 2.2.1 La Librería de Videos Ordinarios de Ámsterdam . . . 11
  - 2.2.2 Correlación Cruzada Normalizada con Media Cero . . . 14
- 2.3 Meta-heurísticas Basadas en Población . . . 15
  - 2.3.1 Algoritmos Evolutivos . . . 16
  - 2.3.2 Inteligencia de Enjambre . . . 17
  - 2.3.3 El Algoritmo de Búsqueda de las Abejas . . . 17
    - 2.3.3.1 La Estrategia Evolutiva de HSA . . . 18
    - 2.3.3.2 Selección por Torneo . . . 18
    - 2.3.3.3 Individuos Aleatorios . . . 18
    - 2.3.3.4 Mutación Polinomial . . . 18
    - 2.3.3.5 Cruce Binario Simulado . . . 19
    - 2.3.3.6 Operador de Repartición . . . 19
    - 2.3.3.7 Exploración, Reclutamiento y Recolección . . . 20
    - 2.3.3.8 Reconstrucción Tridimensional de Escenas con HSA 22
- 2.4 Sistemas de Cómputo Heterogéneo . . . 23
  - 2.4.1 Sistemas de Cómputo Heterogéneo CPU-GPU . . . 23
  - 2.4.2 Sistemas de Cómputo Heterogéneo CPU-FPGA . . . 24
- 2.5 Trabajo Relacionado . . . 26

2.6 Conclusiones del Capítulo 27

3 Diseño de Autómata para el Algoritmo de Búsqueda de las Abejas 28
3.1 Descripción del Autómata 28
3.2 Flujo de los Individuos 30
3.2.1 Identificación de Individuos Inútiles 31
3.3 Complejidad Computacional con CPU 31
3.4 Experimentos Usando CPU 32
3.4.1 Experimentos de Reconstrucción Tridimensional de Escenas . 32
3.4.2 Experimentos de Seguimiento de Objetos en Video 36
3.5 Conclusiones del Capítulo 38

4 Desarrollo de un Sistema Heterogéneo CPU-GPU 41
4.1 Restricciones del Sistemas Heterogéneo CPU-GPU 41
4.1.1 Desventajas de las Meta-heurísticas Ingenuamente Paralelas . 41
4.1.2 Jerarquía y Coordinación de los Elementos de la GPU 42
4.1.3 Generación de Números Aleatorios 43
4.2 HSA Paralelo con GPU 44
4.3 Complejidad Computacional con GPU 45
4.4 Métrica de Eficiencia Puntaje-Tiempo 45
4.5 Experimentos Usando GPU 46
4.5.1 ZNCC Exhaustivo Paralelo 46
4.5.2 Número de Generaciones Eficiente 47
4.5.3 HSA Paralelo contra Propuestas Recientes 47
4.5.4 Búsqueda Exhaustiva contra HSA 49
4.6 Conclusiones del Capítulo 52

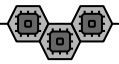
5 Desarrollo de un Sistema Heterogéneo SoC-FPGA 53
5.1 Justificación 53
5.2 Metodología de Codiseño 54
5.3 Materiales 54
5.3.1 Entramado Programable 55
5.3.1.1 LUT 56
5.3.1.2 DSP48 56
5.3.2 Núcleos de Propiedad Intelectual Suaves 56
5.3.2.1 AXI y DMA 57
5.3.2.2 Bloque CORDIC 58
5.3.2.3 Bloque Divisor 58
5.4 Diseño del Sistema 58
5.4.1 Ambiente de Desarrollo en PS 60
5.4.2 Comunicación de Datos de Imagen 60
5.4.3 Máquina de Estados de la Unidad de Control de ZNCC 61
5.4.4 Máquina de Estados de la Unidad de Control de Acumulación 61
5.4.5 Bloque de Promedio 63
5.4.6 Bloque de SSE 63
5.4.7 Bloque de RSSE 64

5.5 Experimentos con SoC-FPGA 66
5.5.1 Calibración con Imagen Estática 66
5.5.2 Un Video con Búsqueda Exhaustiva 67
5.5.3 CPU-GPU contra SoC-FPGA Usando HSA 67

6 Discusión y Conclusiones **71**
6.1 Contribuciones 72
6.2 Trabajo Futuro 73

Referencias **74**

Contents



1 Introduction **1**
- 1.1 Questions and Objectives . . . 3
- 1.2 Contributions . . . 4
- 1.3 Organization . . . 5

2 Theoretical Framework on Relevant Computing Fields **6**
- 2.1 Automata and Formal Languages . . . 6
  - 2.1.1 Mealy and Moore Machines . . . 7
  - 2.1.2 Hardware Implementation of Finite Automata . . . 8
    - 2.1.2.1 Microprogrammed Control Unit . . . 8
    - 2.1.2.2 Hardwired Control Unit . . . 8
    - 2.1.2.3 Comparison of Hardware Implementation Methods 9
- 2.2 Video Tracking . . . 9
  - 2.2.1 The Amsterdam Library of Ordinary Videos . . . 11
  - 2.2.2 Zero Mean Normalized Cross-Correlation . . . 14
- 2.3 Population Based Meta-Heuristics . . . 15
  - 2.3.1 Evolutionary Algorithms . . . 16
  - 2.3.2 Swarm Intelligence . . . 17
  - 2.3.3 The Honeybee Search Algorithm . . . 17
    - 2.3.3.1 The Evolution Strategy of HSA . . . 18
    - 2.3.3.2 Tournament Selection . . . 18
    - 2.3.3.3 Random Individuals . . . 18
    - 2.3.3.4 Polynomial Mutation . . . 18
    - 2.3.3.5 Simulated Binary Crossover . . . 19
    - 2.3.3.6 Sharing Operator . . . 19
    - 2.3.3.7 Exploration, Recruitment and Foraging . . . 20
    - 2.3.3.8 Three-Dimensional Scene Reconstruction with HSA 22
- 2.4 Heterogeneous Computing Systems . . . 23
  - 2.4.1 CPU-GPU Heterogeneous Systems . . . 23
  - 2.4.2 CPU-FPGA Heterogeneous Systems . . . 24
- 2.5 Related Work . . . 26

2.6 Chapter Conclusions 27

3 Automata Design for the Honeybee Search Algorithm 28
3.1 Automata Description 28
3.2 Flow of Individuals 30
3.2.1 Identification of Useless Individuals 31
3.3 Computational Complexity with CPU 31
3.4 Experiments Using CPU 32
3.4.1 Three-dimensional Scene Reconstruction Experiments 32
3.4.2 Video Tracking Experiments 36
3.5 Chapter Conclusions 38

4 Development of a CPU-GPU Heterogeneous System 41
4.1 CPU-GPU Heterogeneous System Constraints 41
4.1.1 Disadvantages of Naive Parallel Meta-heuristics 41
4.1.2 Hierarchy and Coordination of GPU Elements 42
4.1.3 Random Number Generation 43
4.2 Parallel HSA with GPU 44
4.3 Computational Complexity with GPU 45
4.4 Score-Time Efficiency Metric 45
4.5 Experiments with GPU 46
4.5.1 Parallel Exhaustive ZNCC 46
4.5.2 Efficient Number of Generations 47
4.5.3 Parallel HSA versus Recent Proposals 47
4.5.4 Exhaustive Search versus HSA 49
4.6 Chapter Conclusions 52

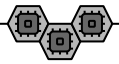
5 Development of a SoC-FPGA Heterogeneous System 53
5.1 Justification 53
5.2 Codesign Methodology 54
5.3 Materials 54
5.3.1 Programmable Fabric 55
5.3.1.1 LUT 56
5.3.1.2 DSP48 56
5.3.2 Soft Intellectual Property Cores 56
5.3.2.1 AXI and DMA 57
5.3.2.2 CORDIC block 58
5.3.2.3 Divider block 58
5.4 System Design 58
5.4.1 PS Development Environment 60
5.4.2 Image Data Communication 60
5.4.3 State Machine of the ZNCC Control Unit 61
5.4.4 State Machine of the Accumulation Control Unit 61
5.4.5 Average Block 63
5.4.6 SSE Blocks 63
5.4.7 RSSE Block 64

5.5 Experiments with SoC-FPGA 66
5.5.1 Calibration with a Static Image 66
5.5.2 One Video with Exhaustive Search 67
5.5.3 CPU-GPU versus SoC-FPGA Using HSA 67

6 Discussion and Conclusions **71**
6.1 Contributions 72
6.2 Future Work 73

References **74**

List of Figures



1.1 The problem of video tracking. 1
1.2 A visual hierarchy of the contents of the thesis. 5

2.1 Sample frames with ground truth box from the ALOV benchmark. . . 12
2.2 Visualization of image frame I and template t. 14
2.3 HSA has three main phases: exploration, recruitment and foraging. . . 21
2.4 Overview of the CPU-GPU heterogeneous system 24
2.5 Overview of the CPU-FPGA heterogeneous system. 25

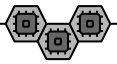
3.1 State diagram for the proposed Moore machine. 29
3.2 Visualization of the count of individuals per state. 33
3.3 Stereo-pairs and results. 35
3.4 Visualization of the convergence of individuals. 36
3.5 Comparison of time costs between HSA and exhaustive search. 37
3.6 Comparison of accuracy between HSA and exhaustive search. 38
3.7 Effects of the reallocation mechanism on video tracking. 39

4.1 Hierarchy of GPU components and OpenCL concepts. 42
4.2 General overview of parallel HSA with GPU. 44
4.3 Effects of the number of generations of HSA. 48
4.4 Effects of GPU parallelization. 50

5.1 The LUT is the core component of any FPGA. 55
5.2 The DSP48 block ALU. 56
5.3 A general overview of the AXI channels. 57
5.4 Top level blocks of the ZNCC processing unit. 59
5.5 State diagram of the State Machine of the ZNCC Control Unit 61
5.6 State diagram of the State Machine of the Accumulation Control Unit 62
5.7 Average block. 63
5.8 SSE block. 64
5.9 RSSE block. 65

5.10 Static image used for calibration. 66
5.11 Exhaustive search with video. 67
5.12 SoC-FPGA versus other architectures. 68
5.13 SoC-FPGA power consumption and real-time video tracking. 70

List of Tables

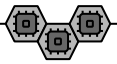


2.1 Main differences between Moore and Mealy machines. 7
2.2 Criteria to select a method to implement a Control Unit. 9
2.3 Categories of the ALOV benchmark. 11
2.4 Optimal configuration parameters of HSA. 22
2.5 Summary of Related Work. 26

3.1 Transition function T. 29
3.2 Results of three-dimensional scene reconstruction. 34

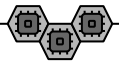
4.1 Parallel HSA with ZNCC as fitness function versus recent trackers. . . 47

List of Abbreviations



- ABC – Artificial Bee Colony
- ABEL – Advanced Boolean Expression Language
- ACO – Ant Colony Optimization
- ALOV – Amsterdam Library of Ordinary Videos
- ALU – Arithmetic Logic Unit
- API – Application Programming Interface
- ARM – Acorn RISC Machines
- AXI – Advanced eXtensible Interface
- CMAR – Control Memory Address Register
- CORDIC – COordinate Rotation DIgital Computer
- CPU – Central Processing Unit
- CLB – Configurable Logic Block
- CU – Control Unit
- CUDA – Compute Unified Device Architecture
- CV – Computer Vision
- DMA - Direct Memory Access
- DSP – Digital Signal Processor
- EA – Evolutionary Algorithm
- ES – Evolution Strategy
- FPGA – Field-Programmable Gate Array
- GA – Genetic Algorithm
- GPU – Graphics Processing Unit
- HDL – Hardware Description Language
- HSA – Honeybee Search Algorithm
- IOB – Input Output Block
- IoU – Intersection over Union
- JPEG – Joint Photographic Experts Group
- LUT – LookUp Table
- MDR – Memory Data Register
- NCC – Normalized Cross-Correlation
- ND-Range – Multi-Dimensional Range
- OpenCL – Open Computing Language
- OpenCV – Open Computer Vision
- PASCAL – Pattern Analysis, Statistical modelling and ComputAtional Learning
- PCI – Peripheral Component Interconnect
- PL – Programmable Logic
- PLD – Programmable Logic Device
- PS – Processing System
- PSO – Particle Swarm Optimization
- RAM – Random Access Memory
- RISC – Reduced Instruction Set Computer
- ROI – Region of Interest
- ROM – Read-Only Memory
- RMI – Regional Mutual Information
- RSSE – Root Sum of Squared Errors
- SI – Swarm Intelligence
- SIMD – Single Instruction Multiple Data
- SoC – System-on-Chip
- SRAM – Static RAM
- SSE – Sum of Squared Errors
- UART – Universal Asynchronous Receiver Transmitter
- USB – Universal Serial Bus
- VHDL – VHSIC HDL
- VHSIC – Very High Speed Integrated Circuit
- VOT – Visual Object Tracking
- VTB – Visual Tracker Benchmark
- ZCC – Zero Mean Cross-Correlation
- ZNCC – Zero Mean Normalized Cross-Correlation

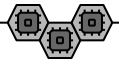
List of Symbols



- A – set of input characters, input alphabet
- a – number of ages or generations
- α – size of mutant offspring population
- B – set of output characters, output alphabet
- β – size of crossover offspring population
- c – niche count function
- d – distance measurement function
- e – a number of work-items
- ε – count of useless individuals
- F – F-score
- f – fitness or objective function
- f_s – sharing fitness function
- G – output function
- g – set of inequality constraint functions
- γ – size of random offspring population
- H – Score-Time Efficiency
- h – set of equality constraint functions
- η – characterization parameter of a probability distribution function
- I – an image frame, a matrix
- I_w – width of I
- I_h – height of I
- $\bar{I}$ – the mean of a region in I
- i – first index
- j – second index
- k – dimensions of the domain of f
- L – lower boundaries
- l – number of inequality constraints
- λ – size of offspring population
- M – a Moore machine
- M_B – extended Moore machine
- m – number of equality constraints
- μ – size of parent population
- n – a number of things
- O – order of the function
- P – probability distribution function
- (p, q) – a point in I
- Q – count of individuals in state
- r – number of recruits
- ρ – uniformly distributed random real value
- S – set of states
- s – element of S
- Sh – sharing function
- σ_s – sharing radius parameter
- T – transition function
- t – an image template, a matrix
- t_w – width of t
- t_h – height of t
- $\bar{t}$ – mean of t
- $\bar{\tau}$ – average time per frame
- U – upper boundaries
- (u, v) – a point in t
- x – value in the domain of f
- ZNCC – function to get the ZNCC value

Chapter 1

Introduction



In simple words, video tracking is the problem of using a computer to automatically follow a known object that can be seen on video as time goes on (Figure 1.1). This simple definition is adequate for non familiarized readers but a formal definition of this central concept is given further into the document (Section 2.2). There are a considerable number of computational techniques or methods that attempt to solve this problem, known as trackers (Kristan et al., 2020). The ultimate goal of a video tracker is to be able to replicate the results of a person that is tasked to mark the location of an object in each one of the static frames in a video. However, this level of accuracy has not been achieved by video trackers. In this sense, the problem of video tracking can not be considered a solved problem. Still, there are many real world applications where video tracking can be helpful. Maggio and Cavallaro (2011) classify those automation applications in six categories: medical applications and biological research; surveillance and business intelligence; robotics and unmanned vehicles; media production and augmented reality; tele-collaboration and interactive gaming; as well as art installations and performances. This means that any improvement on video tracking can translate into benefits in any of those relevant human endeavours.

The video tracking community is heavily focused on accuracy, but there are other considerations such as the speed of video trackers (Galoogahi et al., 2017). The mo-

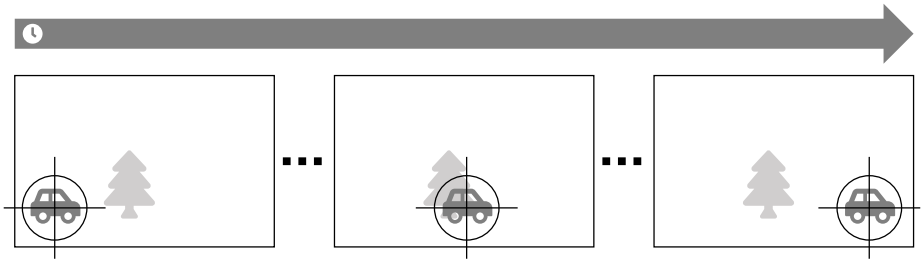


Figure 1.1: The problem of video tracking. A video tracker is a computer program that tries to follow an object that is visible in a video.

tivation of this research project is to contribute to the video tracking community by proposing alternative methods to accelerate video trackers without negative repercussions on accuracy. Video tracking as many other computer vision tasks has to perform a massive amount of operations on a massive amount of data. Digital video is a collection of digital images, and a digital image is a function that returns an amplitude value (known as pixel) that is assigned to a specific two-dimensional spatial coordinate. This translates into a great deal of data that has to be processed every time a video tracker pursues the current location of the object of interest.

In the attempt to get benefits in terms of efficiency rather than efficacy, other works in the literature have used tools such as heterogeneous computing systems and population based meta-heuristics. Heterogeneous computing systems use different combinations of processor hardware to take advantage of parallel processing whenever possible. There are different heterogeneous computing systems, but this work focuses on two of them: the CPU-GPU (Graphics Processing Unit) heterogeneous system, and the SoC-FPGA (System-on-Chip Field-Programmable Gate Array) heterogeneous system. This work is also focused on studying a meta-heuristic, HSA (Honeybee Search Algorithm), which replaces the mechanical and meticulous inspection of all the possible answers with a mindful search procedure that cuts down the waste of resources. The intricacies of how these tools operate will be explained in following chapters.

This research project studies how to design and evaluate a real-time video tracking HSA-based multiprocessing system that uses a SoC-FPGA platform. This task is performed with the intention of answering if a given video tracker can be accelerated while maintaining its precision using that system. However other important variables are observed such as power consumption and area. The presented study follows a general research methodology which is divided in three main phases: research definition, model design, and model evaluation. This general research methodology helps to generate specific methodologies for each problem that is encountered in this work. The following chapters will describe the specific methodologies that were applied.

1.1 Questions and Objectives

The main question that this research will attempt to answer is enunciated as follows:

> "Can the implementation of a video tracker be accelerated without losing precision by using an HSA-based multiprocessing system?"

As an informed guess to answer the question of this research, the hypothesis is the following:

> "A given video tracker can be accelerated while maintaining its precision using an HSA-based multiprocessing system."

The general objective of this research has its origin in the research hypothesis and is enunciated as follows:

> "Design and evaluate a real-time video tracking HSA-based multiprocessing system that uses a SoC-FPGA platform."

The general objective can be decomposed into specific objectives. The specific objectives are the following:

1. "Model the behavior of a honeybee that searches for food as an automaton that can be translated into an electronic design considering different computer vision applications such as video tracking and three-dimensional scene reconstruction."
2. "Analyze, implement and evaluate different types of video trackers in order to select a suitable video tracker."
3. "Design and implement SoC-FPGA and CPU-GPU multiprocessing systems that perform video tracking using HSA."
4. "Propose a metric that helps to compare video trackers in terms of both accuracy and speed."
5. "Validate the proposed systems."
6. "Performance Evaluation of the video tracker implementations in terms of speed, power consumption, area and portability."

1.2 Contributions

The work of this thesis has derived in the following contributions:

1. The journal articles of Perez-Cham et al. (2020) and Perez-Cham et al. (2021).
2. The international congress article of Perez-Cham et al. (2018).
3. An automaton that describes the behaviour of a honeybee that searches for food.
4. The design, implementation and evaluation of a CPU-GPU heterogeneous computing system that performs video tracking using HSA.
5. The design, implementation and evaluation of a SoC-FPGA heterogeneous computing system that performs video tracking using HSA.
6. A new metric that helps to compare video trackers in terms of accuracy and speed.

1.3 Organization

The thesis is organized as follows: Chapter 2 provides a framework that deals with different areas such as automata and formal languages, video tracking, population based meta-heuristics and heterogeneous computing systems. Then, Chapter 3 provides a formal understanding of the HSA meta-heuristic and how it can be used for video tracking and other computer vision problems. Later, Chapter 4 explains how HSA was adapted to be used on a CPU-GPU heterogeneous system and studies the advantages and disadvantages. Finally, Chapter 5 delves into the specifics of the SoC-FPGA implementation of HSA for video tracking and discusses the advantages of this platform. Figure 1.2 provides a visual summary of the hierarchy or dependencies of the contents of this thesis, it shows which subject of the theoretical framework is important to understand each thesis contribution. For example, the formal analysis of HSA is based on the original HSA meta-heuristic, which in turn is based on the general concepts of population based meta-heuristics. Another example is the proposal of parallel HSA for video tracking which is developed taking into consideration what was observed during the development of HSA for video tracking.

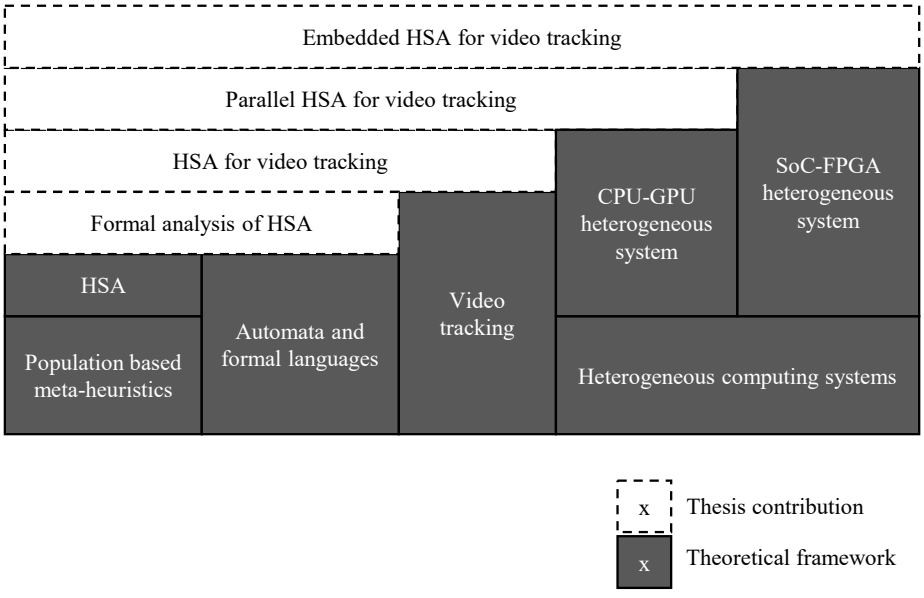
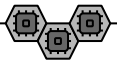


Figure 1.2: A visual hierarchy of the contents of the thesis.

Chapter 2

Theoretical Framework on Relevant Computing Fields



As commented briefly in the Introduction, this research project requires knowledge from different areas such as automata and formal languages (Section 2.1), video tracking (Section 2.2), population based meta-heuristics (Section 2.3) and heterogeneous computing systems (Section 2.4). These subjects will be described in the following chapter to provide an adequate theoretical framework for later chapters. There is special interest in the definition of the Zero Mean Normalized Cross-Correlation (Section 2.2.2) video tracker and the Honeybee Search Algorithm (Section 2.3.3) meta-heuristic which are used on the experiments of this work. The Moore machine model (Section 2.1.1) is another key concept that will be used to formally analyse processes (Chapter 3) or to implement sequential hardware components (Chapter 5).

2.1 Automata and Formal Languages

Automata (automaton in singular) and formal languages are the mathematical foundations of compilers, but they are also useful to describe the functionality of computer architectures. Since this project is focused on working with several different architectures, some background on automata is desirable. An automaton is informally a theoretical construct composed of states that receives a sequence of input characters in strict discrete time steps. A working automaton always has a current state which changes in time according to the combination of input and current state; these changes from state to state are called transitions. Depending on the ending states of the automaton, the input character sequence is evaluated and classified as correct or not correct. A series of input characters that is considered correct by the automaton is one of the possible words that belongs to the formal language that derives from that automaton (Anderson, 2006).

Automaton is a broad term that applies to many different abstract mathematical constructions. When an automaton has a finite number of states it is called finite, in the opposite case, it is called infinite. Another possible distinction of automata

is determinism, an automaton is deterministic when its transitions and outputs are determined strictly by the current state and input character. In other words, if the same state can transition to two or more states when the same event happens, it is not deterministic. The rest of this chapter will focus on finite deterministic automata as their properties make them convenient to work with computer architectures. There are many different finite deterministic automaton types, but there are two main models when dealing with digital hardware implementations: Moore and Mealy machines. The following section will explain their differences and when to use them for digital design.

2.1.1 Mealy and Moore Machines

A Mealy machine can be defined as a 6-tuple $\{S, s_0, A, B, T, G\}$. Where S is the set of states and $s_0 \in S$ is the initial state. A and B are the input and output alphabets, where an alphabet is a set that contains characters that the machine uses to communicate with an external entity. $T : S \times A \rightarrow S$ is the transition function that determines the transition from one state to another. These transitions have three important components: two elements of S and one element of the input alphabet A. If $s_b = T(s_a, A_0)$, this means that the state s_a transitions to s_b when the input character A_0 is received. $G : S \times A \rightarrow B$ is the output function that determines the current output character of the machine based on the current state and the current input character. Similarly, a Moore machine is a 6-tuple $\{S, s_0, A, B, T, G\}$ but with the distinction that the output character indicated by function $G : S \rightarrow B$ is not dependent on the current input character at all, only on the state (Hopcroft et al., 2001).

The digital electronic implementation of a Mealy machine has certain advantages over a Moore machine (Table 2.1). Since the Mealy machine can provide different outputs depending on a combination of current state and input, it is possible to send different output signals during the same state. This will generally result in a lower number of states than the Moore machine (Özgül et al., 2017). Since there are fewer states, the number of time steps to process the same sequence of input characters will also be lower causing a faster input reaction. However, the output function of the Mealy machine depends on two different sets which introduces complexity in the design of the automaton. This output function must be rigorously defined to avoid undesired output signals whenever the input of the automaton changes suddenly while the automaton is performing a transition. If this were to happen it would compromise the stability of the system. Therefore, the Moore machine is considered a safer option in most cases. If

Table 2.1: Main differences between Moore and Mealy machines.

Property	Moore Machine	Mealy Machine
Number of logic cells	Higher	Lower
Input reaction	Slower	Faster
Simplicity of usage	Simple	Hard
Risk of false output	None	Possible

the output depends only on the current state, then the discrete time steps are respected strictly.

2.1.2 Hardware Implementation of Finite Automata

Commonly, the functionality of the Control Unit (CU) of a given digital sequential logic circuit is described using a finite automaton (Hrúz and Zhou, 2007). A finite automaton such as a Moore machine can be implemented in digital hardware using several different methods (Hennessy and Patterson, 2011; Patterson and Hennessy, 2016). However, this section describes only some of the most popular methods to perform this implementation and discusses their advantages and disadvantages.

2.1.2.1 Microprogrammed Control Unit

A microprogrammed CU uses binary control variables stored in memory such as RAM or ROM to implement a finite automaton (Sklyarov, 2000). This is an alternative to a hardwired CU in which control signals are generated by hardware using conventional combinational logic design. On a microprogrammed CU, a Control Memory Address Register (CMAR) stores the address that is used at a given time to read a new microinstruction from the memory (Spanjersberg, 2012). Another important component is the Memory Data Register (MDR) that stores the microinstruction (binary word) that is read from memory and later translated into the specific work signals used in the datapath. This approach is commonly used to implement very complex finite state machines but has the disadvantage of using an important deal of resources.

2.1.2.2 Hardwired Control Unit

This section will describe two different methods to implement a hardwired CU. The first approach is the sequence register and decoder method. This method uses a sequence register to save the current state of the automaton and a decoder to generate an output signal. When using this method, it is important to consider that a register composed of n flip-flops can have up to 2^n states, which could mean lower hardware requirements if the number of states is considerable. The decoder is used to translate the register state into individual signals for each state (Mano, 2003). For example, a 4 state automaton would use a 2-bit register. The available states would be ‘00’, ‘01’, ‘10’ and ‘11’. But the decoder would translate the state ‘00’ to ‘0001’, ‘01’ to ‘0010’, ‘10’ to ‘0100’ and ‘11’ to ‘1000’. This decoding step will allow the corresponding binary signal to cause or stop certain behaviors in the datapath of the design. Since the state register can be easily configured as a binary counter it may be beneficial to use this method on a state machine that has a cyclical behaviour. Still, it is possible to use separate functions of the automaton inputs to set/reset each of the flip-flops of the register.

Another method to implement a hardwired CU is to use individual flip-flops for every state. Only the flip-flop that corresponds to the current state should be set to ‘1’, the rest should be ‘0’ until the state changes. At plain sight, this method increases the number of flip-flops to be used in the system, but there are certain advantages. If the automaton does not describe a cyclical or linear process the flip-flop per state implementation transition

Table 2.2: Criteria to select a method to implement a Control Unit.

Criteria	Microprogrammed Control Unit	Sequence Register and Decoder	Flip-Flop per State
Many states (> 30)	Yes	No	No
Few states (≤ 30)	No	Yes	Yes
Cyclical or linear	Yes	Yes	No
Branching	Yes	No	Yes

logic is simpler which translates in lower hardware requirements. This method does require reviewing excitation or state tables, but the transitions are implemented in a much more straightforward manner because the input signals of each flip-flop can be provided independently.

2.1.2.3 Comparison of Hardware Implementation Methods

The CU designs that are proposed in this work are relatively simple and have a low number of states. In general, the hardwired CU approach is preferred since a microprogrammed CU is considered a misuse of memory blocks. If any given CU is of cyclical or linear nature, this kind of behaviour benefits from an implementation using the sequence register and decoder method. On the other hand, the flip-flop per state method would be better suited if a branching behaviour is observed. Table 2.2 provides a brief summary of the described criteria. Chapter 5 will detail and use these ideas to propose certain components of a SoC-FPGA heterogeneous system.

2.2 Video Tracking

Digital video is a sequence of images taken over time, and each image is a two-dimensional function that receives a spatial coordinate and delivers the value of a pixel which quantifies the product of the reflectance of an object and the illumination (Forsyth and Ponce, 2012). The inspection of a digital video or a digital image is possible by performing statistical analysis of the raw pixel data, this is usually the objective of Computer Vision algorithms (Szeliski, 2010). In the specific case of video tracking, the intention is to find the position of an object in a given image frame. However, the object must be located in a previous frame. Just like the image file itself, a given object inside that image is a collection of pixels. This means that the video tracking problem is in essence the problem of performing a comparison between two sets of data (Maggio and Cavallaro, 2011).

There are many different algorithms that have been proposed to solve the problem of video tracking which are commonly known as video trackers. The efforts to propose and improve video trackers show progress but none of the proposals have achieved a perfectly accurate performance (Kristan et al., 2020). Early video trackers like the

Mean-Shift Tracker (Comaniciu et al., 2000) and the Zero Mean Normalized Cross-Correlation (Nakhmani and Tannenbaum, 2013) are mainly based on statistical analysis and linear algebra to perform pattern recognition. Still, many of these techniques are found at the core of newer proposals that use similar image matching criteria but incorporate machine learning techniques such as the support-vector machine that is fundamental to the Struck video tracker (Hare et al., 2015), or the decision tree that is used in the tracker of Asgarizadeh et al. (2012) to select between NCC or RMI as similarity measure. Currently, the majority of the highest accuracy video trackers are based upon Convolutional Neural Networks (Kristan et al., 2020) such as the SiamMask video tracker (Liang et al., 2019) that employs fully convolutional Siamese networks. Even though these proposals lead in terms of accuracy, they have the disadvantage of requiring a training phase which introduces problems such as sampling bias (Hu et al., 2017), susceptibility to adversarial attacks (Xiao et al., 2020), and a high demand of memory and computational resources to store and process massive amounts of training data.

An important goal of the video tracking community is to increase the accuracy of trackers but there is also concern about the problem of real-time video tracking. The precise definition of the term real-time in this context is disputable. Some authors draw the line of real-time video tracking at the standard speed at which cameras can capture image data which is currently around 240 frames per second (Galoogahi et al., 2017). Still, the point of comparison that other authors use is based on human perception of movement around the mark of 30 frames per second (Zhang and Zheng, 2020). Taking those views into consideration, this work will consider that a video tracker with a speed between 30 and 240 frames per second is an acceptable real-time tracker. The need to increase the speed of trackers and other computer vision tasks has encouraged the adoption of hardware based acceleration tools such as Graphics Processing Units and more recently, Field-Programmable Gate Arrays (Asano et al., 2009). This work studies the possibility of using a heterogeneous computing system along with a population based meta-heuristic to accelerate the speed of a simple tracker.

2.2.1 The Amsterdam Library of Ordinary Videos

The Amsterdam Library of Ordinary Videos (ALOV) is a video tracking benchmark that was proposed by Smeulders et al. (2013). This benchmark provides 314 video sequences with annotated initialization and ground truth data[1]. These video sequences are arranged in 14 categories that pose different challenges for video trackers (Table 2.3). The number and diversity of video sequences (Figure 2.1) makes this benchmark a good alternative to test the performance of video tracker proposals. Another attractive quality of the ALOV benchmark is the focus on common video sequences where pixel information is limited to color. There are other benchmarks that present greater levels of difficulty by including video sequences where the pixel contains thermal, infrared, or depth data (Kristan et al., 2020). This benchmark was initially used to perform a thorough comparison of 19 video trackers, where the Struck (Hare et al., 2015) tracker outperformed other competitors in terms of accuracy. However, the ground truth data of this benchmark is not only useful for evaluation, other works like Held et al. (2016) used the raw data of the benchmark to train deep regression networks for video tracking.

The evaluation of accuracy that the ALOV benchmark uses is based on Intersection over Union (IoU), a concept that is widely used by different video tracking benchmarks. The output of a video tracker is commonly a rectangular area that encloses the suspected position of the object of interest, this area is called the truth of the video tracker. However, the truth of the video tracker is not necessary the actual position of the object of interest, that area is called the ground truth. If the intersection and union areas of truth and ground truth can be measured, then the level of accuracy of the video tracker response can be geometrically interpreted as $\text{IoU} = \text{intersection area} \div \text{union area}$. Note that the IoU value is a real number that ranges from 0 to 1 and that a greater value means a greater accuracy. The evaluation process recommended by the ALOV benchmark is to use the standard PASCAL criterion (Hoiem et al., 2009) to detect true positives and

[1] `http://alov300pp.joomlafree.it`

Category	Description	Videos
Light	Sudden or intense changes on the main source of light.	33
Surface Cover	A piece of fabric covers the surface of the object.	15
Specularity	Shiny objects that reflect light into the camera lenses.	18
Transparency	The tracked object is transparent.	20
Shape	The tracked object changes its shape dramatically.	24
Motion Smoothness	The movement is so slow that it is not detected.	22
Motion Coherence	The movement of the object is erratic.	12
Clutter	The patterns of object and the background are similar.	15
Confusion	The object of interest is surrounded by similar objects.	37
Low Contrast	The colors of the object and background are similar.	23
Occlusion	The object of interest is behind another object.	34
Moving Camera	The camera makes sudden movements.	22
Zooming Camera	The camera zooms in or out while the object stands.	29
Long Duration	A duration between one and two minutes.	10

Table 2.3: Categories of the ALOV benchmark.



Figure 2.1: Sample frames with ground truth box from the ALOV benchmark.

false positives. This criterion establishes that the answer of a video tracker is a true positive when IoU ≥ 0.5. Given that this criterion evaluates only one frame of a given video sequence another metric is used to measure the result of the full video, the F-score. Equation 2.1 is used to get the F-score F based on the number of true positives F_a, false positives F_b and false negatives F_c. Another recommendation of the ALOV benchmark is to use survival curve plots (Collett, 2015) to allow a simple visual inspection of the general trends that are observed in the F-score measurements of many video trackers with several video sequences.

$$F = \frac{F_a}{F_a + \frac{1}{2}(F_b + F_c)} \tag{2.1}$$

One of the original trackers that were tested using the ALOV benchmark is the pure ZNCC tracker. According to their results, the ZNCC tracker is not currently a viable contender in terms of accuracy, but the benchmark recognised that this proposal has historically remained relevant as a fundamental tool used by newer proposals (Li et al., 2020; Lin et al., 2017). The ZNCC function can be used as a simple template matching norm, which is a basic component of any tracker. Similarly, this research selects the ZNCC tracker to study the effect of parallelization and a meta-heuristic on a simple video tracker. This is seen as a proof of concept which could lead to a better tracker proposal in the future. The video sequences of this benchmark are used to test the results of the experiments. However, other benchmarks such as the VOT challenge (Kristan et al., 2020) and VTB (Wu et al., 2013) should be considered for future experiments.

2.2.2 Zero Mean Normalized Cross-Correlation

The Zero Mean Normalized Cross-Correlation (ZNCC) is used to measure the similarity between two ordered groups of data. This grade of similarity is presented as a scalar number that ranges between -1 and 1 (Bätz et al., 2014). If ZNCC returns 0, this means that the compared groups of data show absolutely no correlation or similarity. If the absolute ZNCC value is greater than zero there is a degree of correlation or similarity between the compared groups of data. However, negative ZNCC values indicate that the relationship between the compared groups of data is inversely proportional. ZNCC can be used for video tracking and other computer vision applications if a region inside an image file is interpreted as a two-dimensional array of values. The values of those two-dimensional arrays are commonly integers that range from 0 to 255 and represent a given gray level that is proportional to the product of the object reluctance and illumination.

When ZNCC is used for video tracking, the goal is to make comparisons between groups of pixels in order to find the spatial coordinate (p,q) where a given image template t is most certainty located within an image frame I (Figure 2.2). In order to find the most similar (p,q), every suitable point in I must be checked against template t using ZNCC. There is a range of possible values for (p,q), it depends on the size $I_w \times I_h$ of I and $t_w \times t_h$ of t. The range of p extends from 0 to $I_w - t_w$ while the range of q goes from 0 to $I_h - t_h$. Then, the total number of possible values for (p,q) is of $(I_w - t_w + 1) \times (I_h - t_h + 1)$.

The first step to get the ZNCC value for a given point (p,q) is to use Equation (2.2) to get the mean $\bar{t}$ of all the the gray pixel values $t(u,v)$, where (u,v) is the location of any pixel contained by t with top left corner $(0,0)$ and bottom right corner $(t_w - 1, t_h - 1)$. Note that the value of $\bar{t}$ does not depend on (p,q) which means that it remains constant until template t is changed.

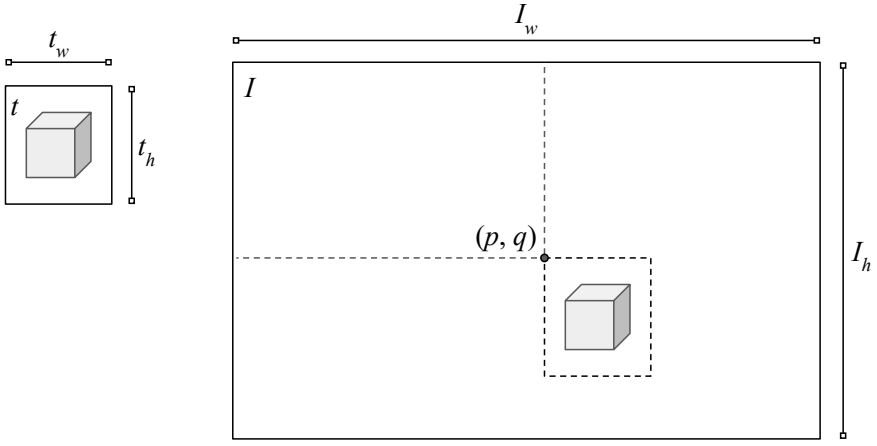


Figure 2.2: Visualization of image frame I and template t. Template t can be contained inside frame I in a rectangle of size $t_w \times t_h$ which has point (p,q) as top left corner.

$$\bar{t} = \frac{1}{t_w \times t_h} \cdot \sum_{u=0}^{t_w-1} \sum_{v=0}^{t_h-1} t(u,v) \tag{2.2}$$

The second step is to use Equation (2.3) to find $\bar{I}(p,q)$ which is the mean of all the gray pixel values $I(u+p, v+q)$, where $(u+p, v+q)$ is the location of any pixel inside the region that has (p,q) as top left corner and $(p+t_w-1, q+t_h-1)$ as bottom right corner.

$$\bar{I}(p,q) = \frac{1}{t_w \times t_h} \cdot \sum_{u=0}^{t_w-1} \sum_{v=0}^{t_h-1} I(u+p, v+q)] \tag{2.3}$$

Once the means $\bar{t}$ and $\bar{I}(p,q)$ are known, the value $\mathrm{ZNCC}(p,q)$ can be found using equations (2.4), (2.5), (2.6) and (2.7). Where ZCC is the zero mean cross correlation, and SSE is a sum of squared errors.

$$\mathrm{ZCC}(p,q) = \sum_{u=0}^{t_w-1} \sum_{v=0}^{t_h-1} [I(u+p, v+q) - \bar{I}(p,q)][t(u,v) - \bar{t}] \tag{2.4}$$

$$\mathrm{SSE}_I(p,q) = \sum_{u=0}^{t_w-1} \sum_{v=0}^{t_h-1} [I(u+p, v+q) - \bar{I}(p,q)]^2 \tag{2.5}$$

$$\mathrm{SSE}_t = \sum_{u=0}^{t_w-1} \sum_{v=0}^{t_h-1} [t(u,v) - \bar{t}]^2 \tag{2.6}$$

$$\mathrm{ZNCC}(p,q) = \frac{\mathrm{ZCC}(p,q)}{\sqrt{\mathrm{SSE}_I(p,q) \cdot \mathrm{SSE}_t}} \tag{2.7}$$

2.3 Population Based Meta-Heuristics

The fields of Swarm Intelligence and Evolutionary Algorithms developed independently but they can be grouped on a broader category: population based meta-heuristics (Bitam et al., 2010). These meta-heuristics are inspired by life and biological processes to solve optimization problems that arise in these areas. More specifically, these different algorithms are intended to solve general optimization problems. The common optimization problem (Luenberger and Ye, 1984) can be expressed as follows:

$$\begin{aligned} \text{Minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, i = 1, \ldots, l \\ & h_j(x) = 0, j = 1, \ldots, m \end{aligned}$$

Where $f : \mathbb{R}^k \to \mathbb{R}$ is the objective function, $x \in \mathbb{R}^k$ is a solution to the optimization problem, $g_i : \mathbb{R}^k \to \mathbb{R}$ is the i-th inequality constraint, and $h_j : \mathbb{R}^k \to \mathbb{R}$ is the j-th equality constraint. A solution x is called feasible if it satisfies all the constraints. The set that contains all feasible solutions is called the feasible region. The answer to this

optimization problem should be some value x^* such that x^* is feasible and $f(x^*) \leq f(x)$ holds for any x. This value x^* is called an optimal value.

Note that the definition of an optimization problem does not restrict the objective function in many ways. Meaning f does not have to be differentiable or have any other property that helps to find the optimal value x^*. As a last resort, a common approach to find the optimal value is to compute all the values in the feasible region one at a time, this is called exhaustive search.

The motivation behind the study of population based meta-heuristics is to provide alternative methods to search for a sub-optimal approximation to x^* with the benefit of lower execution times. In the terminology of population based meta-heuristics, a specific value for x is called an individual, function f may be called the objective function or the fitness function, and a group of individuals is called population. These methods are usually iterative, there is an initial population that changes over time using the fitness function to guide the progress. Each step in this iteration process is called a generation. Metaphorically speaking, several individuals compete against each other and it is determined which ones are better than others based on their fitness $f(x)$. The best individuals get rewarded by some process while the worst individuals are penalized. This goes on for several generations until the population converges to the optimal value x^*.

2.3.1 Evolutionary Algorithms

Evolutionary Algorithms are several different meta-heuristics that draw inspiration from biological evolution to perform intelligent directed search on a given feasible space. The methods that are used to introduce change on the population of these algorithms are called evolutionary operators and these are usually of the types: mutation, crossover, and selection to mirror how those same forces guide the evolution and adaptation of living species. Metaphorically, the fittest individuals survive and generate offspring during several generations until a stop condition is met (Tomassini, 1996).

The purpose of the selection operators is to give the best individuals a greater chance to generate offspring for the next generation. One of the most common and simple selection operators is the tournament selection. This operator uses two individuals that are chosen randomly from the population and decides which one is better using the fitness function. The best individual gets to be part of the list for crossover (or the next generation in some cases). The number of tournament matches depends on how many individuals have to be selected and this depends on the specific meta-heuristic.

Different kinds of evolutionary algorithms have very distinctive mutation and crossover operators which have to match the underlying numerical representation of individuals. For example, Genetic Algorithms use binary strings to represent individuals and the mutation operator randomly selects one bit of the string to flip its value (Golberg, 1989). Other Evolutionary Algorithms such as Evolution Strategies use arrays of floating-point values and may use normally distributed numbers to generate mutation (Deb, 2011). The difference between mutation and crossover is mainly the number of input individuals that are used to generate a new individual. While mutation operators use a single individual, crossover operators use two or more individuals to generate new individuals which are called offspring.

2.3.2 Swarm Intelligence

Swarm Intelligence meta-heuristics are inspired by the collective behaviour of biological agents during their lifespan (Blum and Merkle, 2008). Some prominent Swarm Intelligence meta-heuristics like Particle Swarm Optimization (Clerc, 2010), Artificial Bee Colony (Karaboga and Basturk, 2007), and Ant Colony Optimization (Dorigo et al., 2006) are inspired by social animals like ants, bees, fireflies, birds, or fish. But the number of meta-heuristics is constantly increasing and there are also rarer sources of inspiration, such as flowers (Yang, 2012) or even bacteria (Chen and Lin, 2009). Many of these meta-heuristics have been tested and validated with benchmarking functions and real-world applications. But there are currently no standardized decision criteria to select one above others for any given problem.

Furthermore, there are also many different meta-heuristics that are based on the behaviour of the same animal species. That is the case of honeybees that have inspired several unique meta-heuristics. In the case of honeybee-inspired meta-heuristics, there is also a sub classification based on the specific behaviour that is imitated. There are three main sources of inspiration or metaphors: search for food sources, search for a new hive site, and mating (Bitam et al., 2010). The meta-heuristics inspired by the search for food conducted by honeybees are greater in number; some examples are Artificial Bee Colony (ABC), the Honeybee Search Algorithm (Olague and Puente, 2006a; Perez-Cham et al., 2020) and Mutable Smart Bee Algorithm (Mozaffari et al., 2012). The search for a new hive site metaphor is different to the search for food because there is a greater emphasis on convergence of the full population of individuals; an example is the meta-heuristic of Quijano and Passino (2010). Finally, the honeybee mating metaphor that is used by Haddad et al. (2006) is based upon how male bees compete against each other to inseminate the queen bee and how the queen bee has control over the fertilization of eggs.

The Honeybee Search Algorithm (HSA) shares many characteristics with other honeybee inspired meta-heuristics that mimic the search for food of honeybees. But HSA is also different from other approaches because it is a hybrid proposal that combines techniques from both Swarm Intelligence and Evolutionary Algorithms for real-valued multi-dimensional optimization problems. Another characteristic of HSA is that it was specifically intended to be used on computer vision applications (Olague and Puente, 2006b). These reasons justify this work's selection of HSA to accelerate the ZNCC tracker.

2.3.3 The Honeybee Search Algorithm

The general image processing problem benefits from the metaphor of flying individuals searching for food because in both cases the search space is of considerable dimensions and hints of the optimal solution are widely dispersed (Perez-Cham et al., 2021). In that sense, HSA is heavily influenced by the dynamic flies heuristic of Boumaza and Louchet (2001) but with a greater emphasis on coordinated work. The evolutionary algorithm that is fundamental to HSA is the $(\mu + \lambda)$ Evolution Strategy (ES). This evolutionary algorithm has two populations: the parent population which has a size of μ individuals and the offspring population conformed by λ individuals. The $(\mu + \lambda)$-ES

is an elitist ES which means that the individuals of the parent population can be selected to be part of the next generation's population. Note that in the terminology of ES the scalar variables μ and λ are also used to name or label the populations. It is important to mention that the $(\mu+\lambda)$-ES that is used by HSA is based on the canonical $(\mu+\lambda)$-ES but it is not exactly the same.

2.3.3.1 The Evolution Strategy of HSA

The differences between the HSA $(\mu+\lambda)$-ES and the canonical $(\mu+\lambda)$-ES are mainly related to the generation of the λ offspring population. HSA replaces the ordinary mutation operator of $(\mu+\lambda)$-ES which is applied by adding normally distributed random values to the individual. Instead, HSA uses the Polynomial Mutation Operator (Deb, 2011). Another variation is that the common $(\mu+\lambda)$-ES does not use any crossover operator while HSA uses the Simulated Binary Crossover (SBX) operator (Deb and Beyer, 2001). Also, to avoid local optima HSA randomly generates some of the members of the λ population. These three subdivisions of the λ population are called: α or mutant sons, β or crossover sons, and γ or random sons. To respect the common ES terminology α, β and γ are also the scalar variables that determine the size of the populations in such a manner that $\lambda = \alpha+\beta+\gamma$. Depending on the specific problem, HSA may use the sharing operator of Goldberg and Richardson (1987) that causes the effect of dispersion on the population instead of clustering. This is useful whenever a diverse population is required.

2.3.3.2 Tournament Selection

Each time that an individual is used for crossover or mutation HSA uses tournament selection. This consists in selecting at least 2 random individuals x_i and x_j from the parent population and using the fitness values $f(x_i)$ and $f(x_j)$ to select the best one (Blickle and Thiele, 1996). The best individual is considered the match winner. Mutation requires only one match while crossover requires two matches.

2.3.3.3 Random Individuals

The initial populations and the γ offspring populations are generated randomly but inside the feasible region. A new random individual x_a is a k-dimensional vector. The new random value for the i-th component of x_a is $(U_i - L_i)\cdot\rho + L_i$, where ρ is a uniformly distributed random number that ranges from 0 to 1, U_i is the upper boundary of the i-th dimension and L_i is the lower boundary of that same dimension.

2.3.3.4 Polynomial Mutation

This kind of mutation operator applies variation using polynomial distribution perturbation (Khare et al., 2003). The mutation operation modifies a parent solution x_a to generate a child solution x_b. Individuals are k-dimensional vectors, and the i-th dimension has a lower boundary L_i and an upper boundary U_i. Equations (2.8) and (2.9) are used to get the i-th component of individual x_b where P_δ is obtained using a uniformly

distributed random variable ρ that ranges from 0 to 1, and a user-defined non-negative value η_m that characterizes the probability distribution function.

$$x_{bi} = x_{ai} + (U_i - L_i) \cdot P_\delta \tag{2.8}$$

$$P_\delta = \begin{cases} (2\rho)^{1/(\eta_m+1)} - 1 & \text{if } \rho < 0.5 \\ 1 - [2(1-\rho)]^{1/(\eta_m+1)} & \text{otherwise} \end{cases} \tag{2.9}$$

2.3.3.5 Simulated Binary Crossover

The SBX operator (Deb and Beyer, 2001) emulates the working principle of the single point crossover operator on binary strings used by Genetic Algorithms. This operator receives two parent solutions x_a and x_b and uses them to create two children solutions x_c and x_d. Equations (2.10), (2.11) and (2.12) are used to generate the i-th component of k-dimensional vectors x_c and x_d where P_β depends on a uniformly distributed random variable ρ that ranges from 0 to 1, and on a user-defined non-negative value η_c that characterizes the probability distribution function.

$$x_{ci} = 0.5[(1+P_\beta)x_{ai} + (1-P_\beta)x_{bi}] \tag{2.10}$$

$$x_{di} = 0.5[(1-P_\beta)x_{ai} + (1+P_\beta)x_{bi}] \tag{2.11}$$

$$P_\beta = \begin{cases} (2\rho)^{1/(\eta_c+1)} & \text{if } \rho < 0.5 \\ \left(\dfrac{1}{2(1-\rho)}\right)^{1/(\eta_c+1)} & \text{otherwise} \end{cases} \tag{2.12}$$

2.3.3.6 Sharing Operator

This sharing operator (Goldberg and Richardson, 1987) is based on equations (2.13), (2.14) and (2.15) where the fitness $f(x_i)$ of individual x_i is used to obtain a sharing fitness $f_s(x_i)$. This sharing fitness will be lower than the actual fitness if many individuals gather around the same spot. This depends on the niche count $c(x_i)$ that increases whenever the individual x_i detects that any other individual x_j is inside the same niche. Function $\mathrm{Sh}(x_i, x_j)$ is used to determine if individuals x_i and x_j are in fact inside the same niche and this depends on a distance measurement $d(x_i, x_j)$ and the sharing radius parameter σ_s.

$$\mathrm{Sh}(x_i, x_j) = \begin{cases} 1 - \dfrac{d(x_i, x_j)}{\sigma_s} & \text{if } d(x_i, x_j) \leq \sigma_s \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

$$c(x_i) = \sum_{j=1}^{\mu+\lambda} \mathrm{Sh}(x_i, x_j) \tag{2.14}$$

$$f_s(x_i) = \frac{f(x_i)}{c(x_i)} \tag{2.15}$$

Note that the distance measurement $d(x_i, x_j)$ used in the experiments of this work is either the two-dimensional or three-dimensional Euclidean distance (Liberti et al., 2014). Also consider that the sharing operator is applied on both the μ and λ populations of any $(\mu + \lambda)$-ES that is performed.

2.3.3.7 Exploration, Recruitment and Foraging

Figure 2.3 shows that HSA is divided into three phases: exploration, recruitment and foraging. The exploration phase is the starting step of the optimization procedure which employs the described modified version of the $(\mu + \lambda)$-ES to conduct an initial inspection of the search space. The populations of the exploration phase are labeled μ_e, λ_e, α_e, β_e and γ_e. The recruitment phase mimics how honeybees coordinate their efforts to exploit food sources. Real honeybees communicate with each other using their dance language to spread very reliable and complex information (Crist, 2004). This information is later used by their peers to make collective decisions. The last phase, foraging, conducts a series of $(\mu + \lambda)$-ES sub phases but with reduced search spaces and a greater number of individuals. These new limits are based on the planification and distribution of resources that was performed on the previous phase. The foraging populations have their own names μ_f, λ_f, α_f, β_f and γ_f to be discernible.

Equation (2.16) is used during the recruitment phase to distribute the individuals of the μ_f population into smaller groups μ_{fi} of size r_i each. There is one group of r_i individuals for every individual x_i of the μ_e population. The size r_i depends on the fitness value $f(x_i)$ of the individual x_i and the total sum of the fitness values of all the μ_e individuals. The individual x_i that has the greatest $f(x_i)$ will also have the greatest value of r_i. Metaphorically, the fittest exploration individual gets to recruit the biggest number of peers from the foraging population to help in the exploitation of the resource that was found during the exploration phase.

$$r_i = \left\lfloor \frac{f(x_i)}{\sum_{j=1}^{\mu_e} f(x_j)} \times \mu_f \right\rfloor \tag{2.16}$$

The foraging populations μ_f and λ_f are subdivided into smaller populations μ_{fi} and λ_{fi}. An independent $(\mu + \lambda)$-ES is performed for every subgroup. The initialization of the foraging phase is different to the initialization of the exploration phase. While the initial individuals of the exploration phase are random, the r_i initial individuals of each μ_{fi} group are generated based on the components of the corresponding x_i individual of the μ_e population. This means that the foraging phase continues the overall search using the preliminary results that were obtained from the exploration phase. Since the foraging phase uses a greater number of individuals it also runs for a lower number of generations. Also, depending on the problem the fitness function that is used in the foraging phase can be different to the fitness function used in the exploration phase.

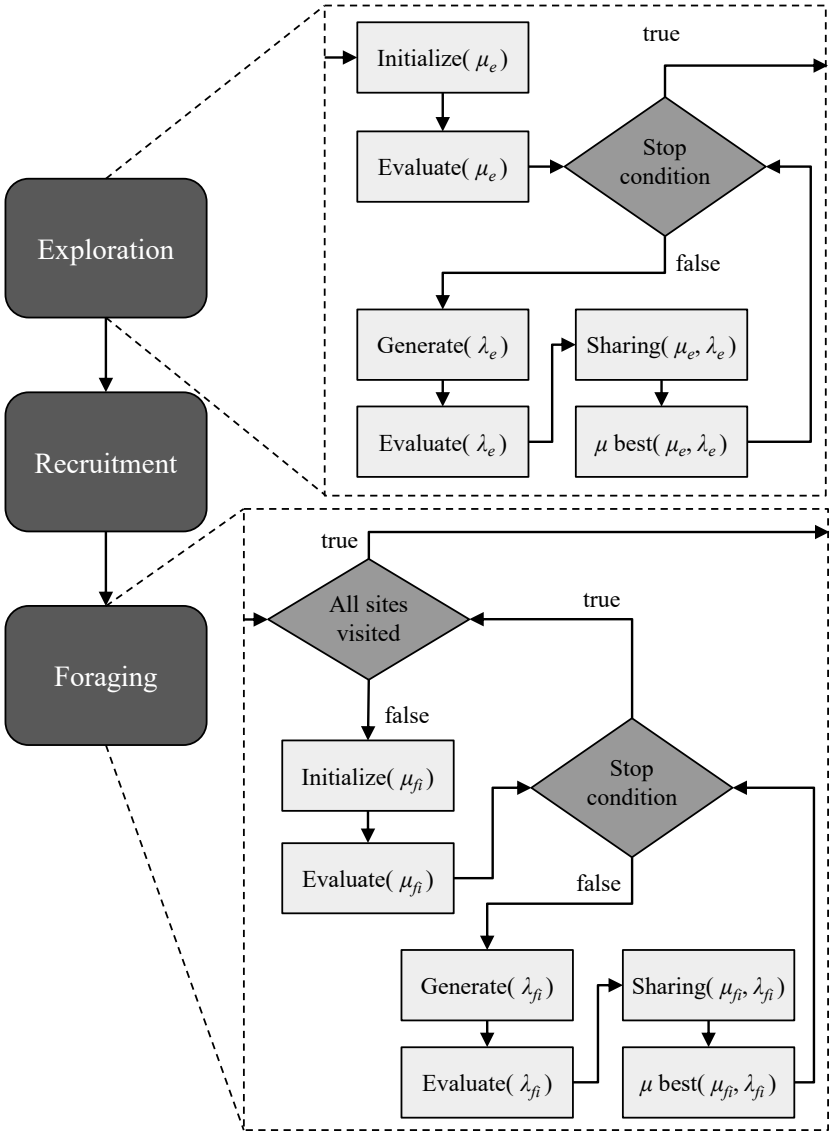


Figure 2.3: HSA has three main phases: exploration, recruitment and foraging. The exploration and foraging phases are based on the canonical $(\mu + \lambda)$-ES but implement different evolutionary operators.

2.3.3.8 Three-Dimensional Scene Reconstruction with HSA

The first application of HSA was three-dimensional scene reconstruction (Olague and Puente, 2006a). Given two different cameras that are positioned to capture the same scene from different angles an attempt is made to generate a faithful three-dimensional model of the visible objects on the scene. The fitness function f that was used on the exploration phase of these experiments was a combination of ZNCC for template matching, Sobel filter (Sobel, 1990) for edge detection and a measurement of the texture homogeneity based on the co-occurrence matrix (Haralick, 1979). In this scenario, the individuals x are four-dimensional points, which are translated to two different points that can be found on the two-dimensional planes of the left and right images. The fitness function was used to determine the grade of similarity of the right and left points which are later used to generate a three-dimensional point using a projection process based on the known position and angles of the cameras that captured the images.

In this case the sharing operator was used to obtain a diverse population of individuals which translated into a dense cloud of three-dimensional points in the result rather than a few clusters of three-dimensional points. Also, the fitness function of the exploration phase does use the Sobel filter to reward individuals that are located near the edges of the objects in the scene, but the fitness function of the foraging phase does not use the Sobel filter as an incentive to the individuals to inspect the surrounding space thoroughly.

The first experiments with three-dimensional reconstruction also established the standard configuration parameters of HSA that are available in Table 2.4. The standard size of α, β and γ populations is established using percentages. The experiments also tested different mutation and crossover operators and different values for their configuration parameters.

Table 2.4: Optimal configuration parameters of HSA. These values were found in the experiments of Olague and Puente (2006a).

Parameter	Description	Value
α_e / λ_e	Percent of mutant sons for exploration	60%
β_e / λ_e	Percent of crossover sons for exploration	10%
γ_e / λ_e	Percent of random sons for exploration	30%
α_f / λ_f	Percent of mutant sons for foraging	60%
β_f / λ_f	Percent of crossover sons for foraging	30%
γ_f / λ_f	Percent of random sons for foraging	10%
η_m	Characterization parameter for mutation	25
η_c	Characterization parameter for crossover	2

2.4 Heterogeneous Computing Systems

Heterogeneous computing system is a general term that refers to any computing system that uses two or more different types of processing elements in coordination. The processing elements that can be used are commonly Central Processing Units (CPU), Graphics Processing Units (GPU), Digital Signal Processors (DSP), or Field-Programmable Gate Arrays (FPGA) but may apply to any other microprocessor design (Prongnuch and Wiangtong, 2016). This trend has grown in use because of the increasing demand for higher computational performance and lower costs in terms of energy consumption, total size or area of the hardware elements, as well as monetary prices (Bean, 2016; Liu and Zhu, 2010). These same motivations have led manufacturers to innovate using the System-on-Chip (SoC) paradigm. SoC is a broad term that refers to any complex integrated circuit that encapsulates all the elements of a fully functional system on the same chip (Martin and Chang, 2001). The common SoC includes components such as a CPU, memory, and hardware acceleration units along with peripheral interfaces.

Different parts of a problem benefit from using different processing elements. A CPU is a better alternative for control intensive operations like searching, parsing, and sorting while a GPU is compatible with image processing or other data intensive tasks thanks to their parallel design (Kaeli et al., 2015). The FPGA is a flexible tool that can handle parallel and sequential tasks and decrease power consumption. However, the flexibility of the FPGA introduces new challenges like the need for hardware/software codesign methodologies (Wolf, 2003). Currently, the most studied heterogeneous systems are CPU-GPU platforms and CPU-FPGA platforms, specifically SoC-FPGA platforms. But CPU-GPU-FPGA platforms are slowly being adopted as they allow the developer to map the computations to specialized processing units capable of performing their tasks more efficiently. However, the CPU-GPU-FPGA platform introduces new problems such as the growing complexity of task scheduling operations (Al-Zoubi et al., 2018).

2.4.1 CPU-GPU Heterogeneous Systems

A GPU is an array of hundreds of processor cores that work together in parallel to help in the acceleration of graphic rendering operations (Owens et al., 2008). However, researchers have adopted this technology and used it for general purpose acceleration of programs. To run a program on a GPU that program should use an API such as NVIDIA Compute Unified Device Architecture (CUDA), or OpenCL which is supported by multiple manufacturers (Kaeli et al., 2015). The design of the general GPU allows the user to run a single program kernel on multiple threads that are executed at the same time on different core processors which is ideal for simple vector operations.

A common GPU is composed of Single Instruction Multiple Data (SIMD) components that are called vector units. Most GPU devices have shared memory that can be read or written by any processor core along with local cache memory that is only available to certain vector units to avoid bottlenecks (see Figure 2.4a). Since the main purpose of the GPU is to help in image rendering, they implement communication channels with common CPU devices to provide access to the global memory. The core processors of a GPU are not intended to communicate with each other very frequently,

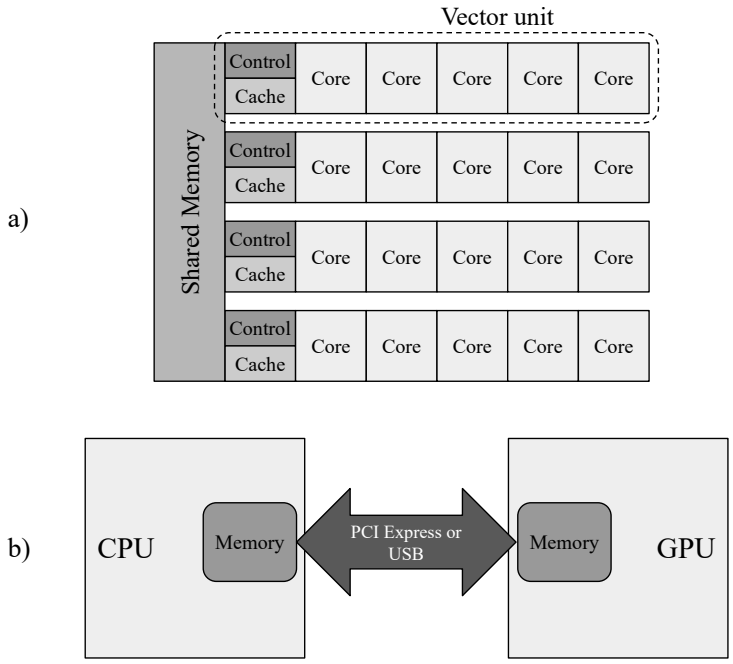


Figure 2.4: Overview of the CPU-GPU heterogeneous system. a) General architecture of a GPU. The shared memory can be read or written by any processor core. The cache memory is only available to vector units (NVIDIA, 2013). b) The communication between CPU and GPU uses serial port interfaces such as PCI express or USB and is commonly controlled by the CPU.

instead they are devised to work independently until their individual tasks are finished. Therefore, a CPU is always in control of the GPU which only enacts very rudimentary synchronization or coordination mechanisms between vector units. As Figure 2.4b illustrates the communication between CPU and GPU uses serial port interfaces such as PCI express or USB and is usually controlled by the CPU using normal memory access instead of faster alternatives like Direct Memory Access (Al-Zoubi et al., 2018; Peng et al., 2009). The concepts that are discussed in this section will be examined with greater detail in Chapter 4.

2.4.2 CPU-FPGA Heterogeneous Systems

Programmable Logic Devices (PLD) were created to help accelerate hardware prototyping and testing (Hamblen et al., 2007). But the role of the PLD has evolved to be more than a development tool. The FPGA is currently the most popular PLD for industrial applications such as automotive, avionics, and automation control (Sangiovanni-Vincentelli et al., 2015). FPGA devices can be reconfigured to behave as a described hardware

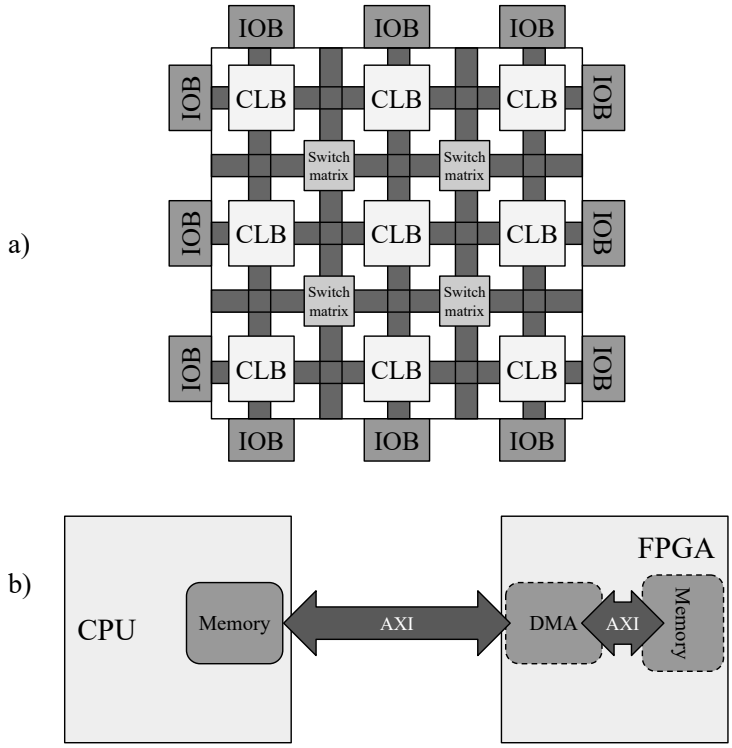


Figure 2.5: Overview of the CPU-FPGA heterogeneous system. a) General architecture of an FPGA. The top-level components are Configurable Logic Blocks (CLB), I/O Blocks (IOB) and switch matrices. b) The SoC-FPGA heterogeneous system has the alternative of using DMA in combination with AXI channels.

design by using Hardware Description Languages (HDL) such as VHDL, Verilog, ABEL, SystemC, etc. (Smith, 1998). Currently VHDL and Verilog have become widely accepted industry standards that dominate the market (Vahid, 2010) with the main difference that VHDL is a strongly typed language which helps to ensure that the design behaves as expected but generates larger source code files. As shown in Figure 2.5a, the main top level components of an FPGA are Configurable Logic Blocks (CLB), I/O Blocks (IOB) and switch matrices which are all independently configurable. CLB components oversee the internal logic behavior, IOB components receive and transmit external data, and all these components are interconnected as required by switch matrices.

It is possible to use the programmable fabric of the FPGA to implement soft processors that behave just like a common CPU. But it can be impractical to build a system from scratch to perform a task that a common CPU can achieve with simple software routines. Therefore, the role of the FPGA on a CPU-FPGA heterogeneous system is to assist the CPU to accelerate operations when the optimization of algorithms is

insufficient. The communication between CPU and FPGA on SoC-FPGA platforms can be implemented in many ways but Direct Memory Access (DMA) is the most adequate for data intensive applications. When DMA is used the data transfer is performed by specialized hardware (see Figure 2.5b) while normal memory access is the job of the CPU (Bean, 2016). This has two advantages: the data transfer is faster, and the CPU is free to perform other operations. This section is only a summary and these terms will be described in depth in Chapter 5.

2.5 Related Work

The review of the literature shows that there have been several research projects that explore the integration of population based meta-heuristics and heterogeneous computing platforms applied to real world problems. Table 2.5 provides a summary of some works in the state-of-the-art that belong to this niche. A generalization that can be made is that most authors focus on the PSO meta-heuristic or variants of it, this work studies the HSA meta-heuristic which has never been used in combination with heterogeneous computing systems or applied to video tracking. Other similar works use an FPGA device to study feasibility of their proposals but this research attempts to contrast SoC-FPGA and CPU-GPU heterogeneous systems to find their advantages and disadvantages to find which is the best option in this particular setting. The work of Morsi et al. (2013) is particularly interesting because the application is video tracking, but this work is different as it compares two different heterogeneous computing platforms. Another interesting work is that of Nogueira and Barboza (2020) because it studies SoC-FPGA and CPU-GPU heterogeneous computing platforms, however the application is not specific as it focuses on continuous optimization problems.

Video tracking is just one of the different CV applications that have been tested.

Source	**Meta-heuristic**	**Application**	**Platform**
Palermo et al. (2008)	Discrete PSO	Multi Objective Design Space Exploration	SoC-FPGA
Tsai et al. (2010)	DNA Algorithm	Fire extinguishment	SoC-FPGA
Morsi et al. (2013)	PSO	Structural SIMilarity index for video tracking	FPGA
Rodriguez and Moreno (2015)	GA	Motion estimation with Particle Filter	FPGA
Elkhani et al. (2018)	Multi-Objective Binary PSO	Feature selection and classification	CPU-GPU
Toumi et al. (2019)	PSO	Bitmap join indexes selection problem in Data Warehouses	CPU-GPU
Nogueira and Barboza (2020)	GRASP (greedy randomized adaptive search procedure)	Continuous optimization problems	SoC-FPGA and CPU-GPU

Table 2.5: Summary of Related Work.

A common trend is to select relatively simple CV algorithms given the difficulty to translate them to hardware. That is one of the reasons why the ZNCC tracker was selected to be used in this work. The precise definition of real-time image processing in these works varies, the range goes from 10 to 30 frames per second. However, this work will consider 30 frames per second as the minimum speed of a real-time tracking system.

2.6 Chapter Conclusions

ZNCC is a digital signal processing technique that was repurposed for video tracking. The consensus of the video tracking research community is that pure ZNCC has been surpassed in accuracy by newer video trackers. However, this algorithm is still studied and improved because it is used as a basic component in more complex solutions. This research project uses the simple ZNCC tracker because similar works limit the scope using relatively simple CV algorithms. The reason is that hardware implementation of algorithms introduces a greater complexity to the project and is still being explored.

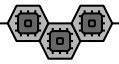
There is a large number of population based meta-heuristics that is constantly growing. The selection of an adequate meta-heuristic for a given application is influenced by the type of optimization problem: multi-dimensional, real-valued, combinatorial, multi-objective, etc. However, the metaphor that inspires the meta-heuristic is also an important factor in this field. This work uses the HSA meta-heuristic that is a hybrid of EA and SI for real-valued, multi-dimensional optimization problems. The inspiration of HSA comes from honeybees, the flying nature of these insects is translated into a search procedure that explores wide search spaces, this is useful in CV applications. The coordination mechanisms of honeybees is also an important benefit of HSA and other meta-heuristics based on these social insects.

Traditionally, the acceleration of computational tasks is achieved through algorithm optimization. Nonetheless, heterogeneous computing systems allow the acceleration of computing tasks by means of hardware parallelism. This work studies CPU-GPU and SoC-FPGA heterogeneous systems to find which is better suited to perform video tracking while also using a meta-heuristic.

The review of the state-of-the-art reveals that there is not a clearly drawn line that establishes the meaning of real-time. This work will consider 30 frames per seconds as the minimal speed of a real-time video tracker, based on human perception of movement which is also the criterion of several works in relevant fields.

Chapter 3

Automata Design for the Honeybee Search Algorithm



This chapter presents a formal language description of the HSA meta-heuristic (Section 2.3.3) which is based on a Moore machine (Section 2.1.1). This automaton describes how the individuals collaborate with their peers while governed by the same set of rules (Section 3.1). To enhance the formal description of HSA a computational complexity analysis is provided in Section 3.3. These sections facilitate the transition from pure software HSA to hardware/software approaches using CPU-GPU and SoC-FPGA heterogeneous systems in following chapters.

Section 3.2 explains the overall flow of individuals based on the proposed automaton. This section also presents a general method that may be used to transfer individuals from one task to another if any excess is detected in the current size of the workforce. Then, Section 3.4 presents the experimental results of Perez-Cham et al. (2021) which use the applications of three-dimensional scene reconstruction and video tracking to find out the benefits of using HSA as well as testing the idea of the individual reallocation mechanism.

3.1 Automata Description

As defined in Section 2.1.1 a Moore machine is a 6-tuple $\{S, s_0, A, B, T, G\}$. The set S of this Moore machine contains 6 states s_0, s_1, s_2, s_3, s_4, and s_5. The input alphabet A contains two symbols 0 and 1, which represent negative feedback and positive feedback. The transition function T describes the way individuals react to the positive and negative feedback messages depending on their current state. Table 3.1 shows which is the next state given the current state and input as determined by function T. Just to respect the convention, the output alphabet B has 6 symbols $\{0,1,2,3,4,5\}$ each one representing the current state. The output function G is defined so that $G(s_0) = 0$, $G(s_1) = 1$, $G(s_2) = 2$, and so on. Figure 3.1 illustrates the proposed Moore Machine with a state diagram where the transitions defined by function T are represented as arrows.

Table 3.1: Transition function T. This function controls how a state transitions to another based on the received feedback.

Current State	**Next State**	
	Negative feedback (0)	**Positive feedback (1)**
Inactivity (s_0)	Exploration (s_1)	Foraging (s_4)
Exploration (s_1)	Inactivity (s_0)	Successful Exploration (s_2)
Successful Exploration (s_2)	Inactivity (s_0)	Recruitment (s_3)
Recruitment (s_3)	Inactivity (s_0)	Recruitment (s_3)
Foraging (s_4)	Foraging (s_4)	Successful Foraging (s_5)
Successful Foraging (s_5)	Inactivity (s_0)	Foraging (s_4)

The states of the proposed Moore machine correspond with the main phases of HSA which are exploration (s_1), recruitment (s_3) and foraging (s_4). The individuals in those states perform the operations of the three main phases of HSA as described in Section 2.3.3. The automaton also includes three states that introduce some changes in the control flow of individuals: inactivity (s_0), successful exploration (s_2) and successful foraging (s_5). These three states are established to allow the individuals to stop wasting resources if it is determined that their activity is not significantly improving the global results. An individual in the inactivity state rests to save resources. Individuals in the successful exploration and successful foraging states are those that have obtained outstanding results in their examination of the search space while they were at the exploration or foraging states. The intention of introducing these states is to implement a resource saving mechanism that will be explained in following sections.

This research proposes to extend the Moore machine model to include a function $Q : S \to \mathbb{N}$ that receives a state $s \in S$ and returns the count or quantity of individuals

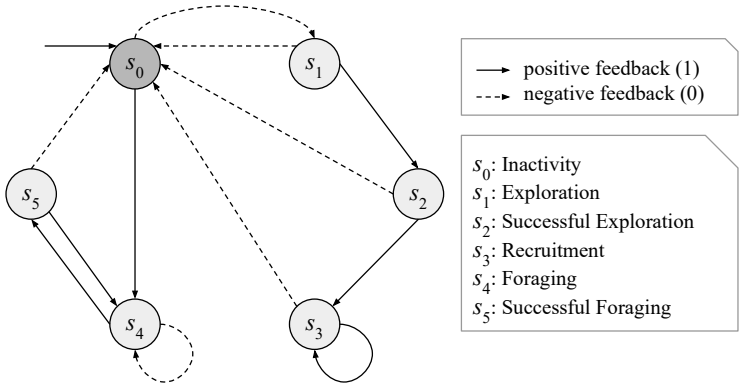


Figure 3.1: State diagram for the proposed Moore machine. The transition function T is detailed in Table 3.1.

that are currently in that state. The value that $Q(s)$ returns is a positive integer obtained by counting the number of individuals that happen to be in that state at the given time. During the transitions defined by function T, the values for $Q(s)$ can be modified by subtracting or adding to the respective current counters in terms of hardware or software. The Moore machine $M = \{S, s_0, A, B, T, G\}$ is part of the extended Moore machine $M_B = M \cup \{Q\}$.

3.2 Flow of Individuals

There is always a total number of $\mu_e + \mu_f$ individuals which are initially set inactive (s_0). This number of individuals comes from the initially defined sizes for the exploration and foraging parent populations (Section 2.3.3). Offspring populations are still used in the interior logic of the ES (Section 2.3.3.1), but from the perspective of the automaton only the μ populations enter and come out of the ES. Once the exploration phase begins, μ_e individuals receive negative feedback, and they transition to the exploration state (s_1) while the rest remain in the inactivity state. The individuals in the exploration state can transition to the successful exploration state (s_2) if they have encountered a promising spot in the search space while the remaining individuals stay in the exploration state. At the end of the exploration phase a small part ε of the μ_e population should return to the inactivity state while the remaining $\mu_e - \varepsilon$ individuals prepare for recruitment. These ε individuals are those that were considered useless or redundant.

During the recruitment phase $\mu_e - \varepsilon$ individuals transition from the successful exploration state to the recruitment state (s_3). The individuals in the recruitment state are used to determine the areas that should be inspected during the foraging phase and how many individuals should be dedicated to each site. At this point $\mu_f + \varepsilon$ individuals that are currently in the inactivity state are available to work on the incoming foraging phase.

As mentioned in Section 2.3.3.7, the total foraging population is divided into smaller μ_{fi} populations of size r_i each. The size r_i of each foraging population is established at the recruitment phase using Equation (2.16). The number of individuals in the recruitment state is $\mu_e - \varepsilon$ so the index i of the subphase will range from 1 to $\mu_e - \varepsilon$. At the beginning of the foraging phase $\mu_f + \varepsilon$ individuals transition from inactivity to the foraging state (s_4).

As each foraging subphase takes place a number ε_i of individuals stay in the foraging state. These ε_i individuals are those that were considered useless or redundant. The remaining individuals of that μ_{fi} population that obtain good results transition to the successful foraging state (s_5). Those ε_i individuals can be reintegrated to the next foraging population in case they can be useful. This reallocation of useless individuals continues until the foraging phase concludes. To remove individuals from the less important sites so they are reused on the more important sites, the foraging populations perform their work in reverse order of importance as defined by the fitness function.

The number ε_i of individuals is determined dynamically on runtime using the method described in the following section. At the end of the full process the individuals in the successful foraging state which should be the best behaved $\mu_f + \varepsilon - \sum_{i=1}^{\mu_e - \varepsilon} \varepsilon_i$ individuals. The detection and reallocation of the total $\sum_{i=1}^{\mu_e - \varepsilon} \varepsilon_i$ worst or redundant individuals is

expected to reduce resource consumption. This section is a summary of the in-depth analysis provided by Perez-Cham et al. (2021).

3.2.1 Identification of Useless Individuals

The sharing operator that HSA uses (Section 2.3.3.6) can be repurposed to find the quantity of individuals that provide redundant information. The experiments that are described in this chapter (Section 3.4) use the value $c(x_i)$, that was defined in Equation (2.14), to quantify if individuals are currently providing redundant information. If this redundancy is detected $c(x_i) > 1$ the individual will be accounted as a possibly useless individual, the count of such individuals will slowly transition to inactivity until a threshold is met (Perez-Cham et al., 2021). The count of possibly useless individuals is used to get the percent of individuals in the μ and λ populations that are under suspicion of being useless. That percentage is dynamically maintained between a lower threshold of 20% and a higher threshold of 60% by dynamically adjusting the sizes of α, β and γ populations and the sharing parameter σ_s. At the end of each $(\mu+\lambda)$-ES the total number ε of individuals has been reallocated as required to keep the percentage inside the threshold range.

3.3 Computational Complexity with CPU

As shown in Perez-Cham et al. (2020) this analysis starts with equations (2.2), (2.3), (2.4), (2.5) and (2.6) that help to find the computational complexity to get the ZNCC value of a given point. The computational complexity to obtain the values for $\bar{t}$ and $\bar{I}(p,q)$ is of $O(t_w \times t_h)$ since both sums process $t_w \times t_h$ elements. The term $\mathrm{ZNCC}(p,q)$ is computed using three different summations and each of them also have a computational complexity of $O(t_w \times t_h)$. This means that the full computational complexity to get $\bar{t}$, $\bar{I}(p,q)$ and $\mathrm{ZNCC}(p,q)$ for a single point (p,q) is $O(t_w \times t_h)$ multiplied by a constant value which is discarded because it does not have an effect on polynomial terms.

In order to perform sequential exhaustive ZNCC every single possible (p,q) has to be computed. Given that the size of the feasible region is $O[(I_w - t_w) \times (I_h - t_h)]$ that is the number of times a ZNCC value should be computed. Equation (3.1) shows the total computational complexity of exhaustive sequential ZNCC in function of the cost of each value $\mathrm{ZNCC}(p,q)$ and the size of the feasible region. Equation (3.2) expands the statement to reveal that the term $t_w \cdot t_h \cdot I_w \cdot I_h$ is the most significant in polynomial terms. This is because $t_w \times t_h$ is expected to be less than $I_w \times I_h$ given that the template t is only a fraction of the image frame I. This is why the polynomial computational complexity of the sequential exhaustive ZNCC is finally reduced to $O(t_w \cdot t_h \cdot I_w \cdot I_h)$.

$$O[(t_w \times t_h) \times (I_w - t_w) \times (I_h - t_h)] \tag{3.1}$$

$$O(-t_w^2 \cdot t_h \cdot I_h + t_w \cdot t_h \cdot I_w \cdot I_h + t_w^2 \cdot t_h^2 - t_w \cdot t_h^2 \cdot I_w) \tag{3.2}$$

If sequential HSA is used instead of exhaustive search the full feasible region is not completely reviewed so the computational complexity is no longer tied to the size of the

image frame I. The computational complexity can be obtained by inspecting Figure 2.3 and Equation (2.16). Operations of the exploration phase like initialization, generation, sharing and selection of the best individuals have a computational complexity of $O(\lambda_e)$ given that the offspring λ_e population is larger than the parent μ_e population. But the most significant computational complexity comes from the evaluation of individuals which depends on the time to evaluate ZNCC(p,q) which has already been established as $O(t_w \times t_h)$. This means that the computational complexity of the evaluation of the λ_e population is $O(\lambda_e \cdot t_w \cdot t_h)$. But the evaluation of the population is repeated for a number a of ages or generations resulting in a computational complexity of $O(a \cdot \lambda_e \cdot t_w \cdot t_h)$. Similarly, the computational complexity of the foraging phase is $O(a \cdot \lambda_f \cdot t_w \cdot t_h)$. Given that λ_f is greater than λ_e, the full computational complexity of HSA with ZNCC as fitness function is bounded as $O(a \cdot \lambda_f \cdot t_w \cdot t_h)$.

3.4 Experiments Using CPU

This section describes experiments with computer vision applications that were performed to study the advantages of using HSA over exhaustive search. Another motivation of these experiments is to study the effects of the individual reallocation mechanism introduced in Section 3.2 and what kind of computer vision problems may benefit from using it. The following experiments used $\sigma_s = 25$ and reuse the configuration parameters of Table 2.4. In the application of three-dimensional scene reconstruction the value $f_s(x_i)$ replaces $f(x_i)$ as fitness function which produces a dispersion effect on the population. This is not done in the video tracking application to ensure the convergence of the population. All the experiments were performed using an Intel Core i7-4790 CPU with clock speed of 3.60 GHz.

3.4.1 Three-dimensional Scene Reconstruction Experiments

These experiments reuse the three-dimensional reconstruction approach that was described in Section 2.3.3.8. Three different stereo-pairs were used for testing. The first stereo-pair (Figure 3.3a) shows a cardboard house with the text 'EvoVisión' on the ceiling and a calibration target with contrasting black squares over a white background as well as other objects, it is borrowed from the initial tests of HSA. The second stereo-pair (Figure 3.3b) shows a computer generated scene where the main objects are coins that have a contrasting texture against the background; it was also used on the first tests of HSA. The third stereo-pair (Figure 3.3c) is borrowed from the Middlebury stereo dataset (Scharstein et al., 2014) and shows some paper cones and a texture rich mask with a human-like face.

The visual results are mainly evaluated in a qualitative fashion. The particular fitness function that was used provides incentives to individuals that explore the spots that show rich textures and contrasting edges as intended in the original experiments of Olague and Puente (2006a). Each stereo-pair shows some attractive objects that were reconstructed accurately. For example, the calibration target and the front wall of the house in the first stereo-pair (Figure 3.3a) attracted individuals because of their sharp edges and the

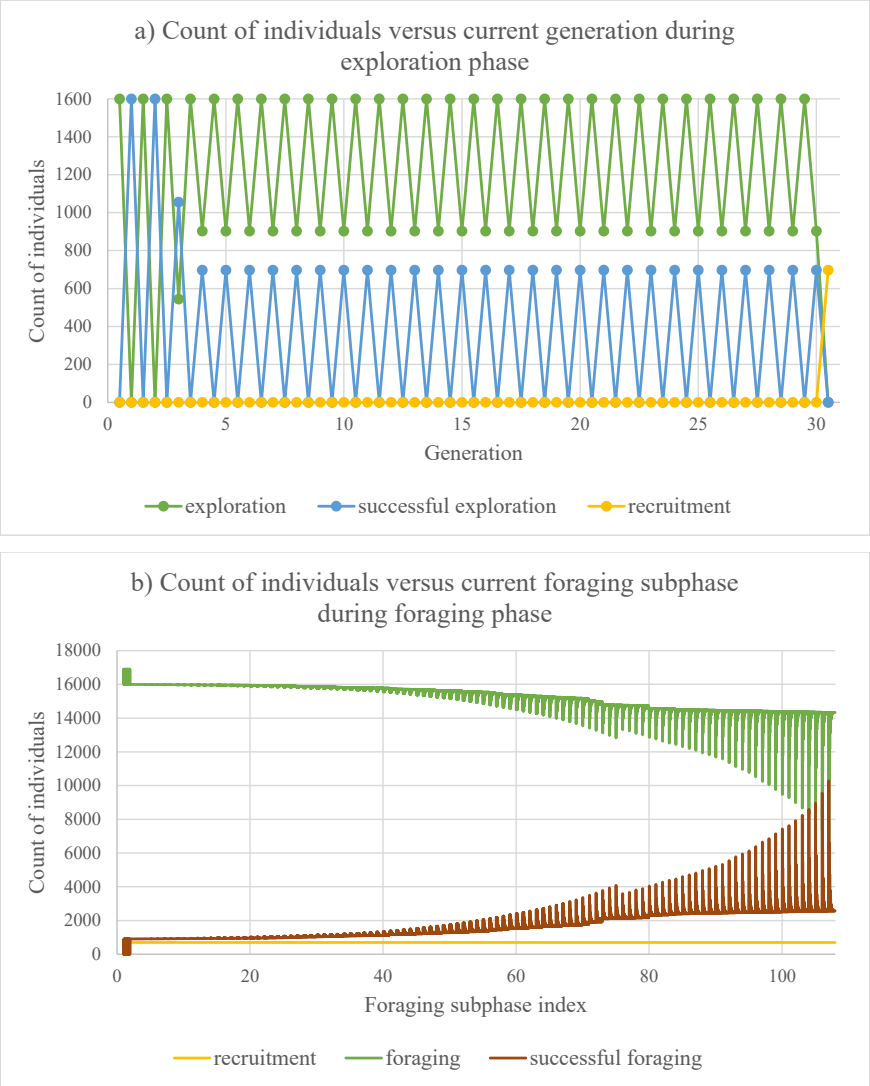


Figure 3.2: Visualization of the count of individuals per state. a) Count of individuals in each state during the exploration phase applying the reallocation mechanism to save resources (EvoVisión stereo-pair). b) Count of individuals in each state during the foraging phase applying the reallocation mechanism to save resources (EvoVisión stereo-pair).

Stereo pair	Using reallocation	Time	Number of usable points	Number of bees used
EvoVisión	No	198,474 ms	7,932	1,600 (μ_e), 16,000 (μ_f)
EvoVisión	Yes	186,861 ms	8,695	697 (μ_e), 2,569 (μ_f)
Coins	No	143,636 ms	14,380	1,600 (μ_e), 16,000 (μ_f)
Coins	Yes	109,812 ms	14,402	227 (μ_e), 14,290 (μ_f)
Cones	No	42,397 ms	13,372	1,600 (μ_e), 16,000 (μ_f)
Cones	Yes	38,322 ms	13,303	102 (μ_e), 1,534 (μ_f)

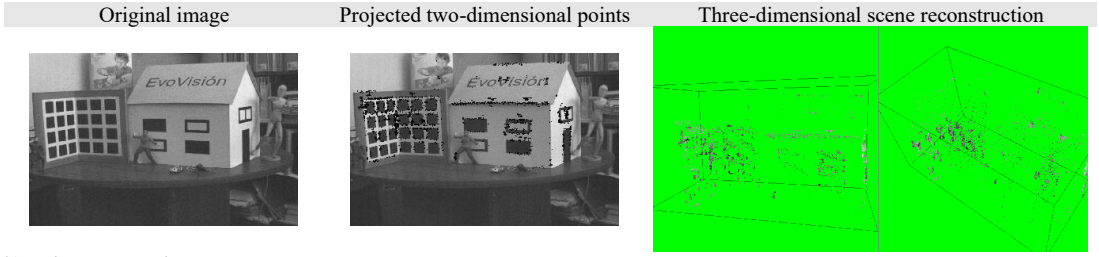
Table 3.2: Results of three-dimensional scene reconstruction. Usable points are those that are feasible and have a fitness value greater than 0. The number of bees used is the count of individuals that were not reallocated.

contrasting tones. The 90-degree angle of the calibration target shows that the reconstruction has a good level of accuracy. In the case of the second stereo-pair (Figure 3.3b) the texture of the flat faces of the coins attracted individuals. The depth of each coin was also detected with a fair level of accuracy in the resulting three-dimensional reconstruction. The most attractive object of the third stereo-pair (Figure 3.3c) is the mask which was reconstructed as an inclined plane on the three-dimensional model. These three example scenes were convincingly reconstructed from a qualitative standpoint.

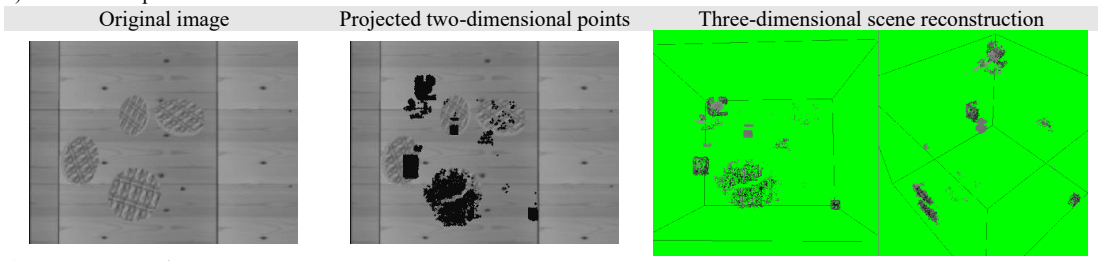
Some quantitative data was also obtained from these experiments to evaluate the implementation of the reallocation mechanism. Table 3.2 shows a summary of the variables that were observed in the experiments. The column that shows the number of usable points provides the count of individuals in the result that have a fitness value greater than zero and those that can be considered feasible individuals (as defined on Section 2.3). The column that shows the number of bees used refers to the count of individuals that kept working until the end of the meta-heuristic.

Note that when the reallocation mechanism is used in all cases the number of bees used reduces. This means that several individuals were considered useless and transitioned to inactivity. The time that it took for the program to finish was noticeably affected by the reallocation mechanism because the number of working individuals was reduced. On the other hand, the number of usable points kept almost constant. These results are considered positive because the three-dimensional reconstruction is very similar, but it was obtained faster which translates in resource savings. Figure 3.2a shows that the number of individuals in the successful exploration state (blue line) starts at 1,600 but gradually reduces until a balance is found by the reallocation mechanism. Similarly, Figure 3.2b shows the same trend on the count of successful foraging individuals (maroon line) during the full extent of the foraging phase that is subdivided in 120 subphases. This confirms that the reallocation mechanism is taking effect in the number of active individuals.

a) EvoVisión stereo-pair



b) Coins stereo-pair



c) Cones stereo-pair

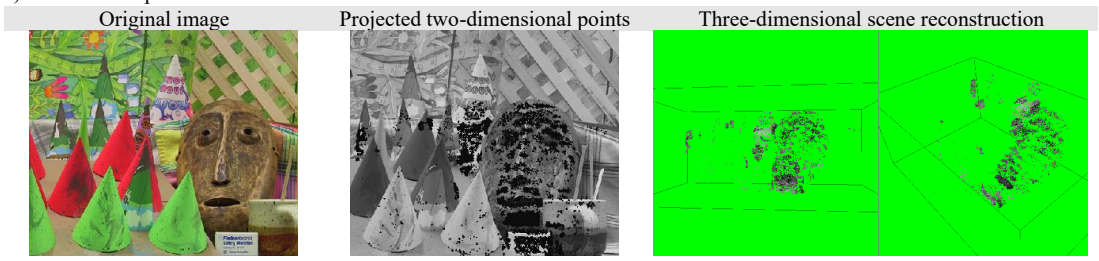


Figure 3.3: Stereo-pairs and results. a) EvoVisión stereo-pair. b) Coins stereo-pair. c) Cones stereo-pair.

3.4.2 Video Tracking Experiments

The following experiments use the exhaustive ZNCC method (Section 2.2.2) and compare the performance against HSA with ZNCC as a fitness function. The initialization of the exploration phase is not fully random; the initial individuals are positioned near the last known location of the object (p,q) on the previous image frame. The sharing operator is used to conduct the reallocation mechanism but the fitness function is not affected, Equation (2.15) is not used. The optimization problem is defined as follows:

$$\begin{aligned} \text{Maximize} \quad & \text{ZNCC}(p,q) \\ \text{subject to} \quad & 0 \leq p \leq I_w - t_w \\ & 0 \leq q \leq I_h - t_h \end{aligned}$$

The first variable that is observed measures the time costs that are spent to process one frame. The average time per frame $\bar{\tau}$ is obtained for every video sequence using Equation (3.3) where τ_i is the time used to process the i-th frame of the video sequence and $|\tau|$ is the total number of frames in the video sequence. The other observed variable is the accuracy that is measured using the F-score that is obtained using Equation (2.1). Since the number of videos is considerable, the results are summarized using survival curve plots (Collett, 2015).

$$\bar{\tau} = \frac{\sum_{i=1}^{|\tau|} \tau_i}{|\tau|} \tag{3.3}$$

The benchmark video data is borrowed from the Amsterdam Library of Ordinary Videos (ALOV) proposed by Smeulders et al. (2013). Initial calibration experiments determined that increasing the number of generations as well as the sizes of μ_e and μ_f populations of HSA increases the time costs, but the accuracy remains almost constant after 2 generations, 50 μ_e individuals, and 500 μ_f individuals. These values are considered the most efficient configuration parameters for this particular application. Figure 3.4 illustrates how individuals of the exploration phase get attracted to the optimal point (p,q).

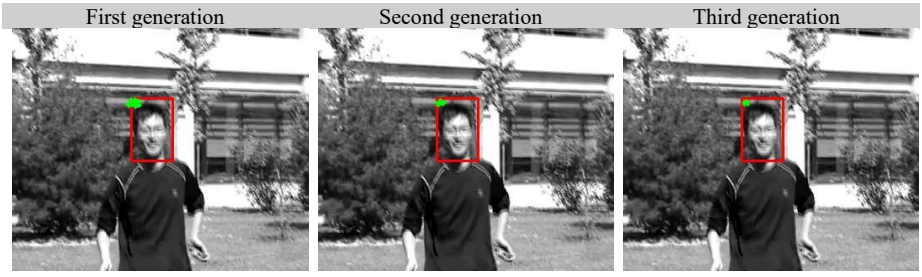


Figure 3.4: Visualization of the convergence of individuals. After some generations the individuals (green points) converge to the optimal value (top left corner of the red box).

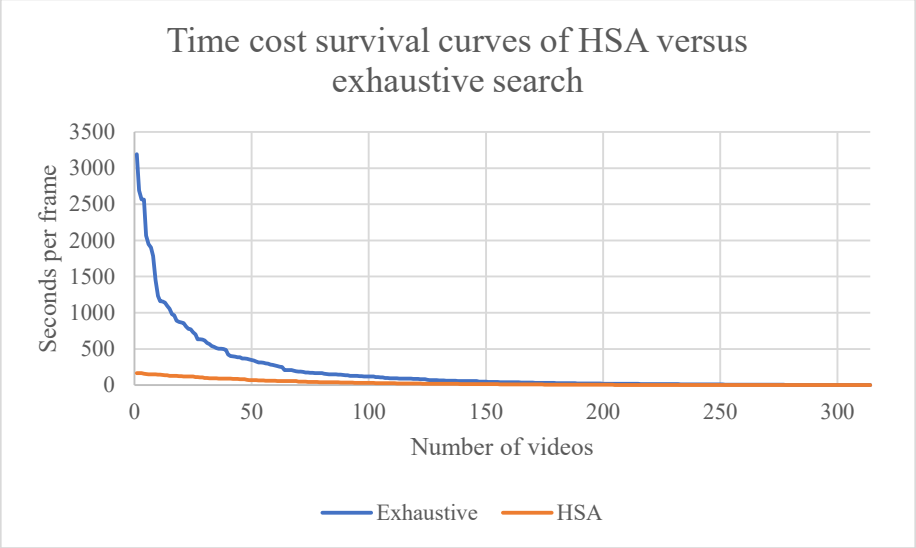


Figure 3.5: Comparison of time costs between HSA and exhaustive search.

The comparison between exhaustive ZNCC and HSA with ZNCC as fitness function reveals that using HSA reduces time costs as predicted with the computational complexity analysis (see Figure 3.5). Exhaustive tests were performed using the 100 videos that had lower values for I_w, I_h, t_w and t_h because it was deemed impractical to use all the 314 videos of the benchmark given the high time costs that were observed using exhaustive ZNCC. Still, to find an approximation of the average time per frame the first frame of each video was processed using exhaustive ZNCC. The exhaustive tests took 202.1399 seconds per frame on average while the tests that used HSA took 0.6788 seconds per frame on average.

In terms of accuracy, the experiments with the 100 video samples prove that there is not an important difference between using HSA or exhaustive search (see Figure 3.6). The average F-score of exhaustive search tests is 0.545375 while HSA tests have an average score of 0.545311. The small difference between the average scores is less than 1% of the overall standard deviation. This is interpreted as a positive result because the smart directed search of HSA reviews only a fraction of the feasible search space, but this does not have a negative effect on accuracy. In conjunction with the reduced time costs, these results provide motivation to study the effects of parallelization of HSA in following chapters. Also, the tests with GPU accelerated exhaustive ZNCC of the next chapter uses all the 314 videos of the benchmark to further confirm these results.

Finally, the experiments that tested the reallocation mechanism show that there are not important benefits on this particular application (video tracking) if the mechanism is used. These tests focused on three variables: the F-score to measure accuracy (Figure 3.7a), the seconds per frame to measure time costs (Figure 3.7b), and the maximum fitness value that was obtained to measure if the underlying meta-heuristic was improved (Figure 3.7c). No important variations were observed on any of these variables.

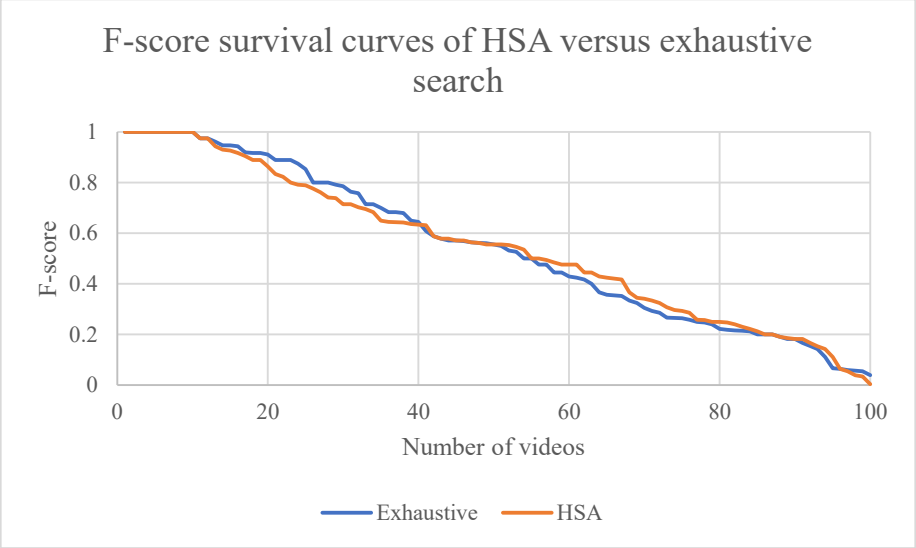


Figure 3.6: Comparison of accuracy between HSA and exhaustive search.

The lack of effects indicates that the reduced number of generations in these experiments was an obstacle because the mechanism detects the redundancy of individuals after a few generations as shown in the three-dimensional scene reconstruction experiments (see Figure 3.2). Also, the nature of the problem is so that the population should always converge to the optimal spot rather than disperse (see Figure 3.4). These findings will be discussed further in the following section to conclude the chapter and establish the next steps.

3.5 Chapter Conclusions

Both practice and theory show that $O(a \cdot \lambda_f \cdot t_w \cdot t_h) < O(t_w \cdot t_h \cdot I_w \cdot I_h)$ which proves that HSA with ZNCC as fitness function presents an improvement against exhaustive ZNCC. But this complexity study attempts to complement the analysis of the extended Moore machine M_B proposed in this chapter to assert which specific tasks of HSA can benefit the most from parallelization. At first glance the terms that have a greater weight on the time costs are the ones related to the fitness function (t_w and t_h) rather than the meta-heuristic itself (a and λ_f). This evidence shows that the bottleneck is in fact the evaluation of the fitness function. The main conclusion of this chapter is that any parallelization efforts on any heterogeneous system should focus on the evaluation of the fitness function.

As shown in Section 3.4 the application of three-dimensional scene reconstruction benefits from using the reallocation mechanism that was introduced in this chapter. The positive effects are mainly reflected in the lower execution times but could be translated to other resource savings. However, the results of the application of video

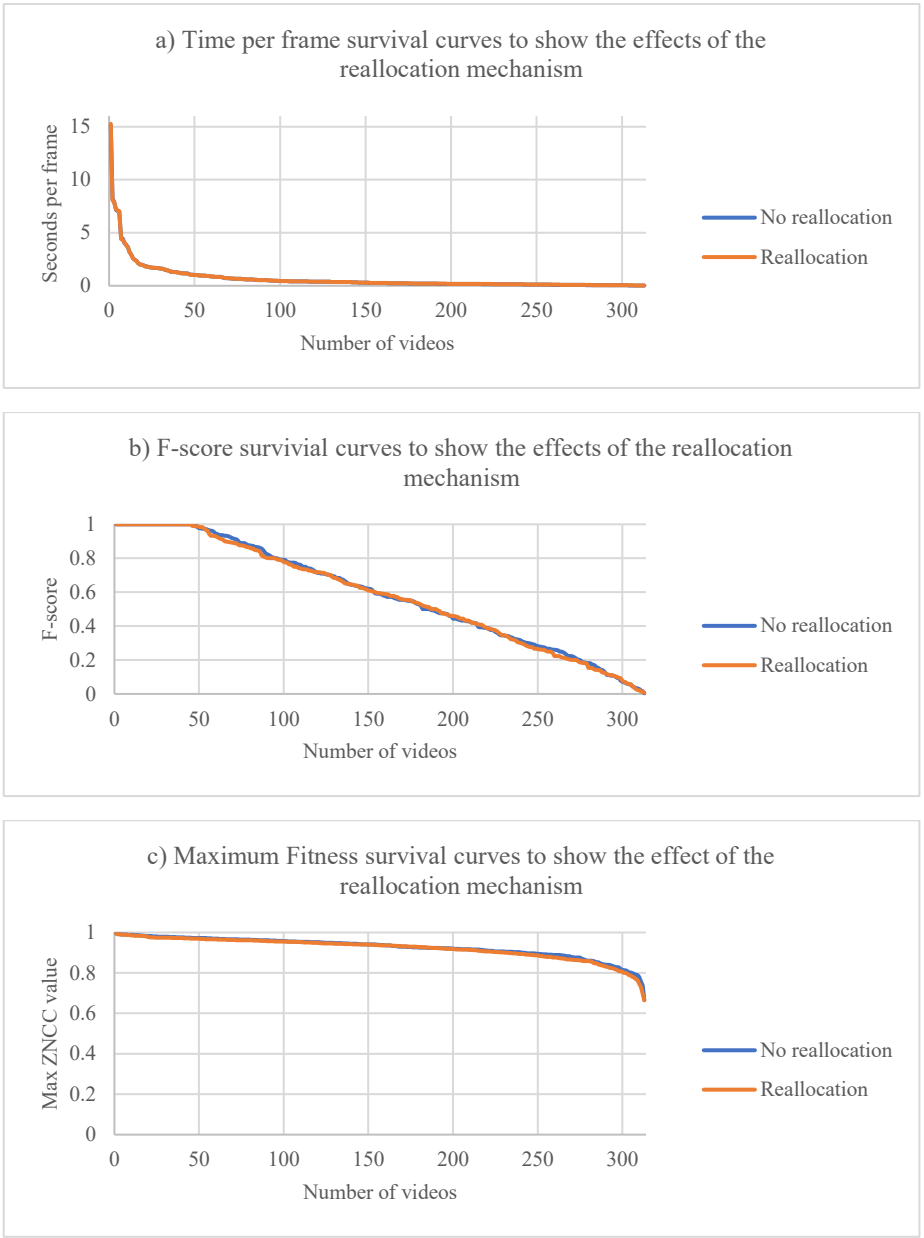


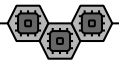
Figure 3.7: Effects of the reallocation mechanism on video tracking. a) Effects on the time costs when using the reallocation mechanism. b) Effects on accuracy when using the reallocation mechanism. c) Effects on the best fitness value when using the reallocation mechanism.

tracking is not significantly improved by using this reallocation mechanism. Since this thesis is mainly concerned with the application of video tracking with ZNCC and the experimental results of Section 3.4 show that this application does not benefit from the reallocation mechanism of individuals the following experiments will not use this mechanism. However, it should be noted that the reallocation mechanism is beneficial to the three-dimensional scene reconstruction application and should be studied further with similar applications.

The study of how individuals flow using the general outline of the M_B automaton reveals which parts of HSA are sequential in nature. The order of exploration, recruitment and foraging phases should be respected even on parallel architectures because each of these need the results of the previous phase as input. Also, introducing the evaluation that determines the proportion of the population that is not useful depends on the fitness values of those individuals. Meaning that the reallocation mechanism of individuals that is described in Section 3.2 would force a sequential evaluation of the foraging sub populations μ_{fi} if implemented. Following chapters delve into the specifics of the parallelization of HSA with ZNCC as fitness function in different heterogeneous computing systems.

Chapter 4

Development of a CPU-GPU Heterogeneous System



This chapter studies a proposal to parallelize the HSA meta-heuristic using a CPU-GPU heterogeneous system and tests it using ZNCC as fitness function to perform video tracking (Section 4.2). The parallelization approach is described after a brief insight on the hardware and software limitations that influenced this work's proposal (Section 4.1). A formal analysis of the computational complexity of this parallel HSA proposal is provided in Section 4.3. Then, a new metric is proposed in Section 4.4 to simplify the comparison of different video tracker proposals taking into account time costs and accuracy. Finally, Section 4.5 presents the experiments and results of Perez-Cham et al. (2020) that study different aspects of the parallel version of HSA with ZNCC as fitness function.

4.1 CPU-GPU Heterogeneous System Constraints

This section explains the limitations of the CPU-GPU heterogeneous system that were encountered during the development of a parallel version of HSA. Section 4.1.1 describes some problems that have been reported by other authors that tried to use GPU devices to parallelize population based meta-heuristics. Then, Section 4.1.2 explains the hierarchy of the hardware components of a GPU and how this is related to the concept of work-items and work-groups. Finally, Section 4.1.3 comments on the difficulties of random number generation in this setting.

4.1.1 Disadvantages of Naive Parallel Meta-heuristics

As mentioned in the conclusions of the last chapter (Section 3.5), the naive parallelization of HSA should focus on the evaluation of the fitness function given that it is the most computationally expensive task. However, the survey of Tan and Ding (2015) points out that one major concern to parallelize swarm intelligence meta-heuristics

using CPU-GPU heterogeneous systems is the high temporal costs of the CPU-GPU communication (more details on Section 2.4.1). Many implementations of swarm intelligence meta-heuristics avoid the high time costs of the communication between CPU and GPU by performing every single operation of the meta-heuristic on the GPU. Only initial data is transferred from the CPU to the GPU and the results are delivered to the CPU. When there is code that cannot be parallelized it is preferable to simply perform it in one of the cores of the GPU rather than waiting for the CPU-GPU communication.

There is a common problem with the implementations of swarm intelligence meta-heuristics on CPU-GPU heterogeneous systems: when global synchronization is required for certain activities it cannot be achieved because GPU hardware is not designed to do so (Orr et al., 2015). Synchronization of GPU cores is only possible between groups that belong to the same logic block depending on the specific model. The problem of synchronization dramatically reduces the computational resources that can be used effectively to parallelize a swarm intelligence meta-heuristic using this type of heterogeneous systems. The next section provides more details on why the need for global synchronization is a problem.

4.1.2 Hierarchy and Coordination of GPU Elements

It is important to make a distinction between the logic layer of a GPU program and the physical layer of the actual GPU hardware (Figure 4.1). The logical layer defined by OpenCL uses the term work-item to refer to a program thread that is executed by a vector unit. This vector unit (Section 2.4.1) is an array of GPU cores which allows the

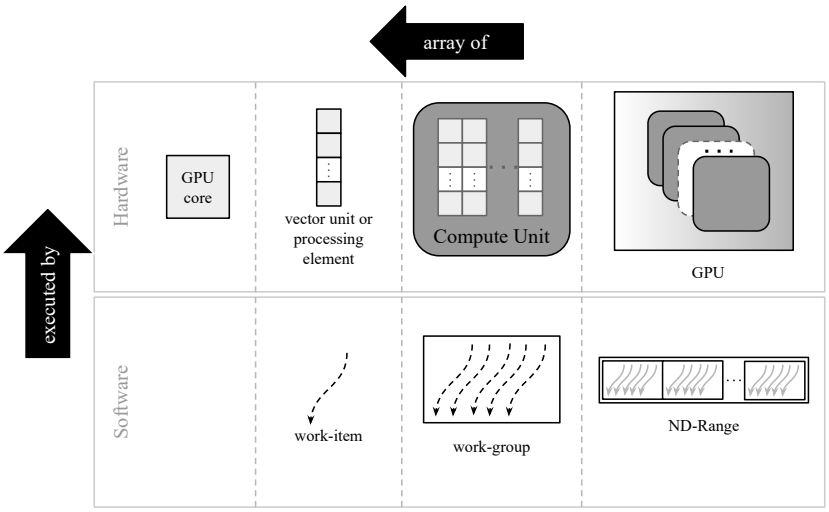


Figure 4.1: Hierarchy of GPU components and OpenCL concepts. Synchronization is possible between work-items of the same work-group. The work-groups that are enqueued on the ND-Range are not warranted to be executed in order or simultaneously, they are attended by the first available Compute Unit (Alberto and Sato, 2015).

vector unit to physically perform a single operation on several pieces of data at once. The number of work-items that are executed in parallel by the same vector unit is usually greater than the actual number of physical cores that the GPU contains because of the pipeline architecture that is used (Gaster et al., 2012). A GPU core is just an ALU, the vector unit is the first component in the hierarchy that actually has a sequential control unit.

A work-group is an array of several work-items that are executed on vector units that are located inside the same Compute Unit which is a physical array of vector units. The maximum size of a work-group is usually a power of 2 that depends on the specific model and architecture of a GPU (Mantor, 2012). Each Compute Unit has high-speed low-latency local data storage which is shared by the local vector units. This local data storage makes synchronization possible, but only between work-items inside the same work-group.

The next level in the hierarchy is the ND-Range which is a logical array of work-groups. The acronym 'ND' in the name means that this array can have one, two or three dimensions which is practical for different rendering operations. All the tasks of the ND-Range are executed using all the computing resources of the GPU. This ND-Range is actually implemented as a queue of work-groups, meaning one work-group is attended as soon as a Compute Unit is available.

Image rendering operations and other simple vector tasks can easily take advantage of GPU parallelization. However, the mapping of work-groups to Compute Units that is performed using the ND-Range gives no guarantee that all the work-groups are attended simultaneously. This creates problems on applications that require global synchronization of all work-groups. A far-fetched solution is to use high-latency global memory to implement a custom global coordination mechanism but this particular problem is still being studied (Orr et al., 2015)

4.1.3 Random Number Generation

A common problem that is encountered by GPU-accelerated computer simulations, is that GPU devices have no native pseudo-random number generators (Manssen et al., 2012). The reason is that GPU devices were not initially intended to be used on such applications. Also, the common GPU instruction set is limited to allow the massive replication of the components. This was a technical challenge during the development of parallel HSA because it uses random numbers to perform operations such as tournament selection, mutation, crossover and initialization.

The solution that was implemented uses the CPU to pre-generate four arrays of random numbers. The first array contains uniformly distributed integers to select individuals by index (Section 2.3.3.2). The second array contains uniformly distributed real numbers used to generate random individuals for initialization or the random offspring (Section 2.3.3.3). The third array contains random numbers generated using the beta distribution for SBX (Section 2.3.3.5). The random numbers of the last array are generated using the delta distribution for Polynomial Mutation (Section 2.3.3.4). Both the beta and delta distribution functions are described thoroughly in the work of Khare et al. (2003).

4.2 Parallel HSA with GPU

As explained by Perez-Cham et al. (2020), the work of each individual is done by a group of e_0 GPU work-items. The maximum number of individuals that can work simultaneously is e_1/e_0, where e_1 is the maximum work-group (Figure 4.2). To avoid increasing the computational costs geometrically, the size of populations μ_e, λ_e, μ_h and λ_h are all set to the same size of e_1/e_0. As commented in the conclusions of the previous chapter (Section 3.5), the order of the exploration, recruitment and foraging phases is respected. Since the exploration and foraging phases do not happen simultaneously, the same groups of e_0 work-items are reused by different individuals.

The parallelization of exploration and foraging phases of HSA (Figure 2.3) is very similar because both are based on the HSA $(\mu+\lambda)$-ES (detailed in Section 2.3.3.1). Each individual of the initial μ population is generated by one of the available e_0 work-items. Individuals of the μ_e population are randomly scattered to be near the last known position (p,q) in the previous frame while the individuals from the μ_f population are randomly scattered around the site marked by the exploration individual that recruited them. The generation of λ offspring populations of mutant sons (α), crossover sons (β) and random sons (γ) is also performed by a single work-item per individual. The selection of the μ best individuals to be part of the population of the next generation requires sorting all the individuals of the μ and λ populations by fitness. The sorting algorithm used in the described implementation is a parallelized merge-sort, that uses one work-item per individual (Davidson et al., 2012).

The conclusions of the previous chapter explain that the evaluation of the fitness function should be the focus of parallelization. So, when the fitness function is evaluated all e_0 work-items that belong to each individual are used. Since the fitness function is ZNCC which is mostly based on summations, this work uses the reduction operations described by Nickolls et al. (2008). The summations of equations (2.2), (2.3), (2.4),

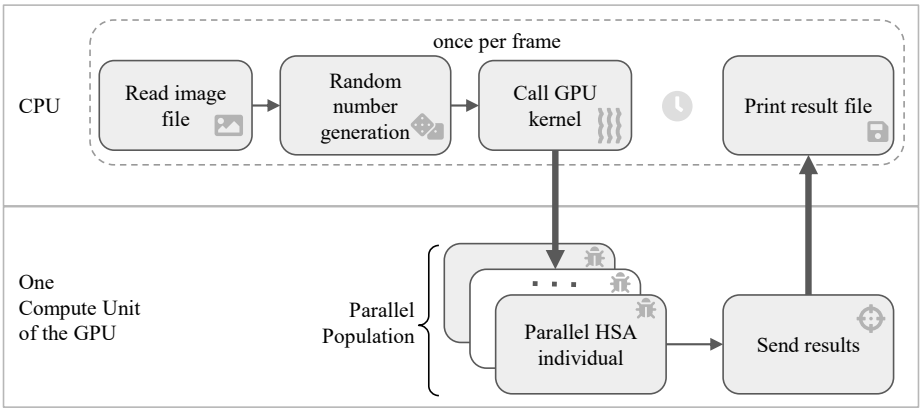


Figure 4.2: General overview of parallel HSA with GPU. Only one Compute Unit is used to allow synchronization. CPU-GPU communication is used only when it is indispensable.

(2.5) and (2.6) are parallelized by distributing the $t_w \times t_h$ elements in smaller groups of $\frac{t_w \cdot t_h}{e_0}$ elements. A preliminary sum is obtained for each group and later used to get the full sum.

The only phase of HSA that is not parallelized is the recruitment phase. Following the advice of Tan and Ding (2015), one work-item is in charge of the assignment of foragers to each μ_{fi} sub population as dictated by Equation (2.16). The boundaries of the search space are reduced based on the cloud of the best μ_e individuals at the end of the exploration phase. To visualize the cloud of μ_e individuals see Figure 3.4.

4.3 Computational Complexity with GPU

Section 3.3 breaks down the analysis that concludes that the computational complexity of exhaustive ZNCC is $O(t_w \cdot t_h \cdot I_w \cdot I_h)$ and that the computational complexity of HSA with ZNCC as fitness function is of $O(a \cdot \lambda_f \cdot t_w \cdot t_h)$. Note that there is no dependency to compute any given ZNCC(p,q) in any order so the naive parallelization of exhaustive ZNCC using the GPU can perform the computation of several ZNCC(p,q) simultaneously. If e_m is the maximum number of simultaneous work-items of the GPU, then the computational complexity of parallel exhaustive ZNCC is naturally bounded as $O(\frac{t_w \cdot t_h \cdot I_w \cdot I_h}{e_m})$. Note that using the GPU to parallelize a data intensive image processing task such as exhaustive ZNCC is relatively straightforward.

Parallel HSA with ZNCC as fitness function is described in Section 4.2. In this case, the evaluation of the fitness function ZNCC(p,q) for each individual is performed by e_0 work-items simultaneously. This means that the computational complexity to evaluate the fitness function is bounded by $O(\frac{t_w \cdot t_h}{e_0})$. Note that all the individuals work simultaneously which means that the computational complexity is no longer tied to the size of the populations μ_e, λ_e, μ_f or λ_f. However, each generation of the iterative process is dependent on the results of the previous generation. Thus, the overall computational complexity of parallel HSA with ZNCC as fitness function is of $O(\frac{a \cdot t_w \cdot t_h}{e_0})$.

Logic dictates that $O(t_w \cdot t_h \cdot I_w \cdot I_h) > O(\frac{t_w \cdot t_h \cdot I_w \cdot I_h}{e_m})$ which means that parallelization improves exhaustive ZNCC. The same is true for $O(a \cdot \lambda_f \cdot t_w \cdot t_h) > O(\frac{a \cdot t_w \cdot t_h}{e_0})$ which evidences that the GPU parallelization of HSA successfully accelerates the process. Nonetheless, the comparison between using exhaustive search or using HSA is not so trivial. Stripping the terms t_w and t_h that come up in the computational complexity of these algorithms reveals that HSA may have an advantage if the size of the image frame $I_w \times I_h$ is significantly greater than the number of generations a.

4.4 Score-Time Efficiency Metric

The work of Perez-Cham et al. (2020) proposes the Score-Time Efficiency metric to complement the comparisons of video tracker implementations based on accuracy and time costs. The F-score F, defined using Equation (2.1), is used as an accuracy indicator. The average time per frame $\bar{\tau}$ is used to measure the time costs using Equation (3.3). Score-Time Efficiency H is defined as a rate that describes how much benefit is obtained in terms of F-score by each second spent to process the frame. If the best tracker

implementation cannot be determined using F or $\bar{\tau}$, then the implementation with the greater Score-Time Efficiency will be considered superior. Equation 4.1 shows how we obtained the Score-Time Efficiency for a given video sequence.

$$H = \frac{F}{\bar{\tau}} \tag{4.1}$$

4.5 Experiments with GPU

This section describes the experiments of (Perez-Cham et al., 2020) with a CPU-GPU heterogeneous system. Most of the experiments presented in this section used an AMD Radeon R9 270 GPU with a clock speed of 925 MHz. This GPU contains 1,280 physical cores that can work on several work-items at once (Mantor, 2012). The cores of this GPU are arranged in vector units of 16 cores. These vector units are also arranged in Compute Units composed of four vector units (64 cores). The maximum number of synchronized work-items or maximum work-group is of 256 parallel work-items (Gaster et al., 2012).

The CPU of this heterogeneous system is an Intel Core i7-4790 with a base frequency of 3.60 GHz. The CPU-GPU communication interface is based on PCI Express 3.0. The maximum work-group e_1 is set to 256 given the capabilities of the GPU. The number of work-items e_0 that each individual uses is set to 4 to allow $e_1/e_0 = 64$ individuals to work simultaneously. According to the initial experiments of Section 3.4.2, the optimal size for the μ_e population is around 50 and 64 is the closest power of 2. Also, 64 is the number of physical cores that are used. As a reminder, $\mu_e = \lambda_e = \mu_f = \lambda_f = e_1/e_0$. The tests that use exhaustive search use a naive parallelization approach by letting each work-item to focus on one ZNCC(p,q) computation and the ND-Range is defined to enqueue all the feasible values for (p,q).

4.5.1 Parallel Exhaustive ZNCC

The GPU accelerated exhaustive ZNCC was compared against the sequential implementation that was used in the experiments of Section 3.4.2 to confirm the speedup effects. As a reminder, sequential exhaustive ZNCC was tested with 100 of 314 videos of the ALOV benchmark. But the average time per frame of all the 314 videos was estimated by processing at least one frame of each video.

As predicted by the computational complexity analysis, the GPU accelerated implementation has lower time costs. The average time per frame of the sequential implementation is of 202.1399 seconds per frame while the GPU implementation takes 1.1823 seconds per frame. Also, there are no important effects in the observed accuracy. The average F-score of the CPU and GPU implementations are 0.5453 and 0.5051, respectively. The average F-score of both implementations shows a small difference of 1% of the overall standard deviation. This small variation comes from the difference in the precision of floating-point values. The CPU uses double precision (8 bytes) and the GPU uses simple precision (4 bytes) to save memory resources.

4.5.2 Efficient Number of Generations

The computational complexity analysis indicates that the number of generations is an important factor that affects the time costs. The initial experiments of Section 3.4.2 showed that the most efficient number of generations of HSA was around 2 generations. This section will use the Score-Time Efficiency to find if that assertion is correct. These experiments use the 314 videos of the ALOV benchmark. Figure 4.3a shows that the average F-score remains almost constant when the number of generations of HSA is between 2 and 10, this is interpreted as an asymptotic behaviour. Note that the trend of the average F-score is not always ascending as the number of generations increases. This happens because F-score is not the fitness function, it is an evaluation of the accuracy of the fitness function.

Figure 4.3b confirms that time costs increase as the number of generations of HSA increases, the trend is clearly linear. In order to stop this increasing time costs the number of generations should be established as the minimal possible value that provides an acceptable level of accuracy. Figure 4.3c shows that the highest average Score-Time Efficiency is achieved with two generations which confirms that the increasing time costs do not provide any benefit in terms of accuracy. Note that the measurement of Score-Time Efficiency allows a straightforward evaluation of different tracker proposals.

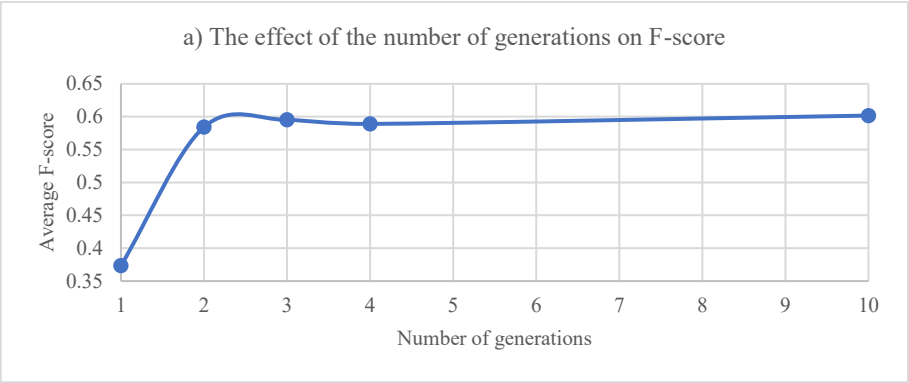
4.5.3 Parallel HSA versus Recent Proposals

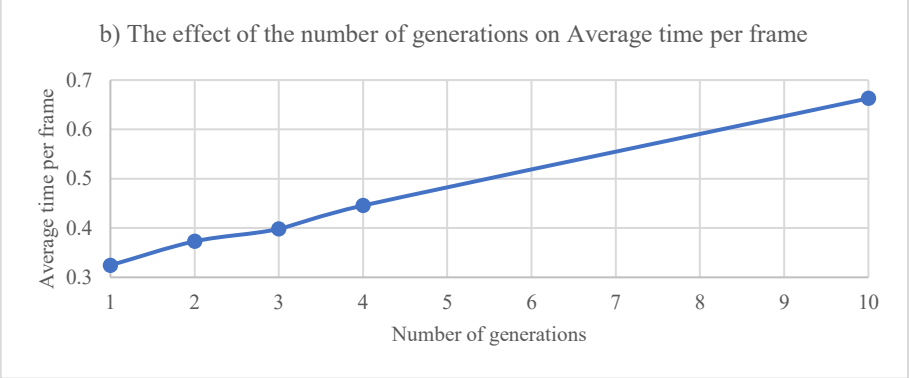
These experiments that test state of the art trackers used an NVIDIA GeForce GTX 1050 GPU with 640 cores and 1354 MHz clock speed. The CPU-GPU communication interface is also based on PCI Express 3.0. As commented in Section 2.2, pure ZNCC has been surpassed in terms of accuracy by recent trackers. Still, this section presents experiments that compare the performance of parallel HSA with ZNCC as fitness function versus recent trackers. The recent trackers that were selected are Struck (Hare et al., 2015) and SiamMask (Liang et al., 2019). SiamMask is based on Convolutional Neural Networks and is currently among the top tier trackers. The Struck tracker remains relevant as one of the most accurate non-trained proposals. The summary of the results is shown in Table 4.1. Only 14 videos of the ALOV benchmark were used to test these trackers, one per category.

Not surprisingly, the accuracy of HSA with ZNCC as fitness function is inferior

Tracker	**Seconds per frame**		**F-score**	
	Average	**Standard deviation**	**Average**	**Standard deviation**
Struck	0.0334	0.0103	0.7070	0.3012
SiamMask	0.0345	0.0011	0.6026	0.3166
Parallel HSA ZNCC	0.0725	0.0099	0.3546	0.2730
Sequential Exhaustive ZNCC	147.5051	297.3442	—	—

Table 4.1: Parallel HSA with ZNCC as fitness function versus recent trackers. This is based on 14 videos of the benchmark, one per category.





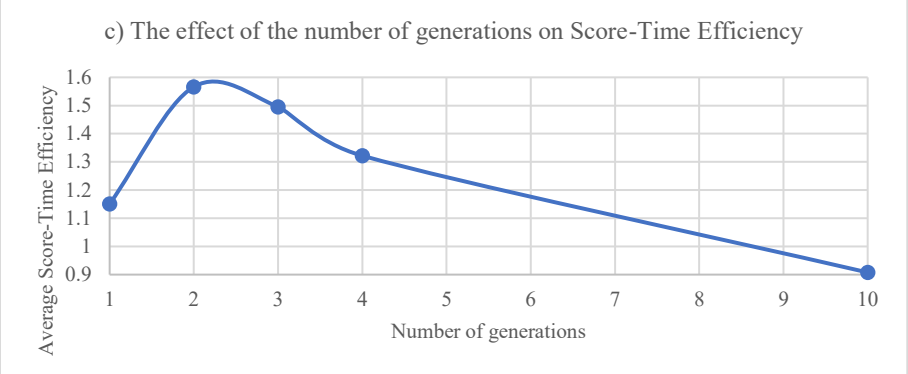


Figure 4.3: Effects of the number of generations of HSA. a) Effect of the number of generations on accuracy. b) Effect of the number of generations on time costs. c) Effect of the number of generations on Score-Time Efficiency.

to that of SiamMask and Struck because the meta-heuristic does not increase the accuracy of the fitness function. However, it is interesting to note that Struck showed a better accuracy than SiamMask. The experiments point out that Struck shows a greater robustness when the resolution of the video is lower and when the objects in the background resemble the target object. A possible explanation is that the SiamMask tracker was not trained to face these specific challenges. These results make it clear that the current tracker proposal of this work is not ready to face recent state of the art trackers on accuracy contests.

Struck and SiamMask have lower time costs than parallel HSA with ZNCC as fitness function, but these time costs are on the same order of magnitude (10^{-2}). It was observed that the combination of HSA as meta-heuristic and parallelization via GPU achieves a notable acceleration effect. This combination of tools reduced the high time costs of sequential exhaustive ZNCC which is greater by 4 orders of magnitude (10^2). This finding shows that the HSA meta-heuristic fulfills its main purpose: acceleration. The results encourage a continued study of the effects of this meta-heuristic applied to video tracking. However, the accuracy measurements indicate that HSA should be adapted to use other more accurate similarity measures as fitness function on future video tracking experiments.

4.5.4 Exhaustive Search versus HSA

The following experiments complement the tests of Section 3.4.2 where only 100 videos of the ALOV benchmark were used to test the accuracy of sequential exhaustive ZNCC. The goal of this section is to study the effects of parallelization and HSA. Figure 4.4a confirms that the effect of HSA on accuracy is negligible. The average F-score of parallel HSA and parallel exhaustive search tests is 0.5841 and 0.5883, respectively. This small variation represents 1.29% of the overall standard deviation. This evidence supports the same conclusion of the last chapter. As expected, the parallelization of HSA has no important effects on accuracy which, in turn, validates the faithfulness of the new version.

It is clear that sequential exhaustive ZNCC (gray line) is by far the most time consuming alternative (Figure 4.4b). As suggested in Section 4.3, the strategy to avoid searching on the full feasible region reduces time costs. The logarithmic scale of Figure 4.4b helps to visualize that the slope of both HSA implementations is lower than the slope of parallel exhaustive search. As the size of the image frame and template grows, all the options take longer to get the results. But, the time costs of parallel and sequential HSA grow at lower rates which is a positive effect. Note that the growing time costs of parallel HSA (red line) are also lower than the growing time costs of sequential HSA (yellow line). The last fact is surprising given the issues related to the CPU-GPU heterogeneous platform that are explained in Section 4.1.1. These difficulties in the implementation of parallel HSA on a CPU-GPU heterogeneous system confirms that the GPU architecture is not the ideal option when it comes to meta-heuristics. However, the results prove that parallel HSA has advantages over sequential HSA and parallel exhaustive search.

Figure 4.4c shows the Score-Time Efficiency survival curves of sequential HSA, parallel HSA, and parallel exhaustive search. Parallel exhaustive search has a marginal

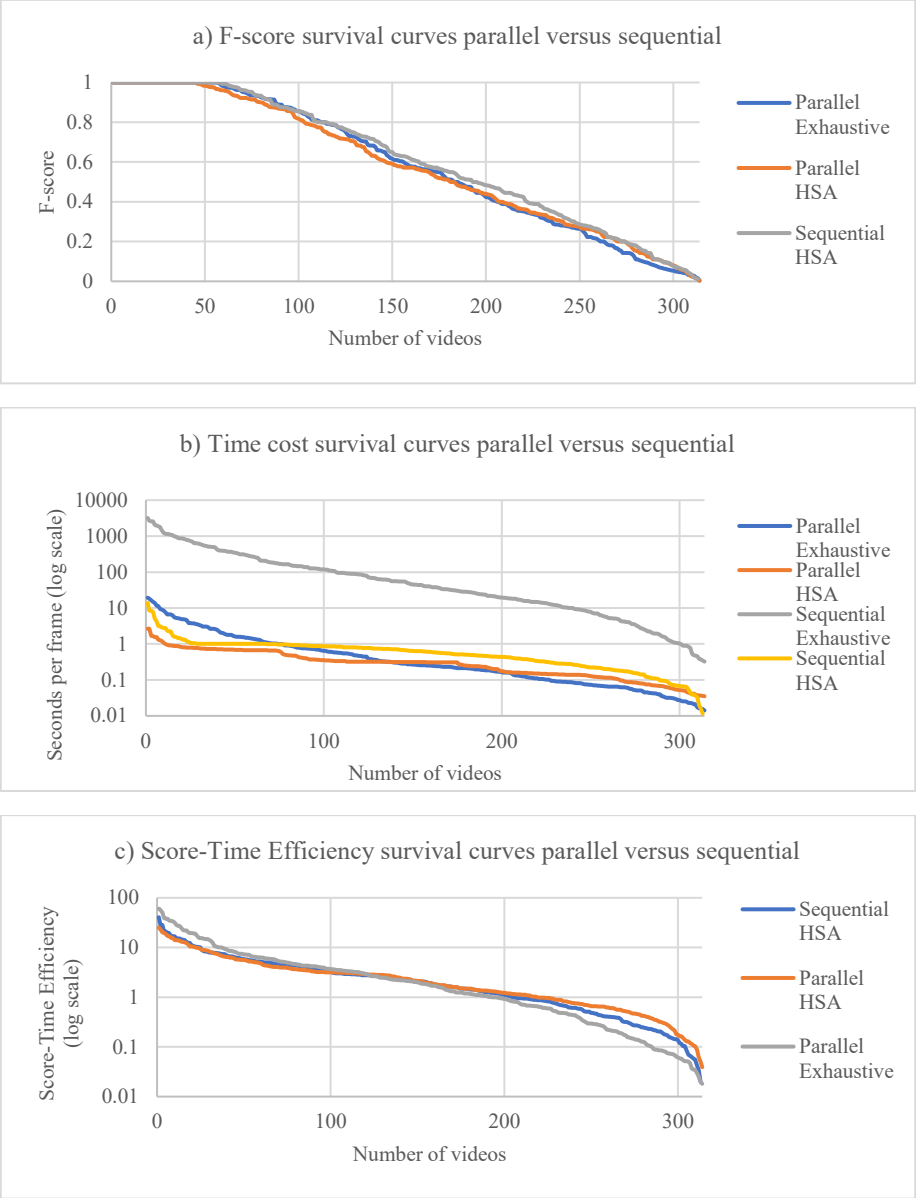


Figure 4.4: Effects of parallelization. a) Effect of parallelization on accuracy. b) Effect parallelization on time costs. c) Effect of parallelization on Score-Time Efficiency.

advantage in terms of Score-Time Efficiency. However, these three lines are all inside the same range (0.01 to 100) which indicates that the efficiency level of these three options is the same in order of magnitude. Also, the parallel exhaustive search implementation uses the full capacity of the GPU (1,280 physical cores) while the parallel HSA implementation uses only a fraction of those resources (64 physical cores). Therefore, this juxtaposition between parallel HSA and parallel exhaustive search is not a fair comparison.

4.6 Chapter Conclusions

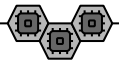
The experimental results of this chapter confirm that using the HSA meta-heuristic reduces the heavy temporal costs of the simple exhaustive search of the ZNCC tracker. The reduction of costs that HSA produces is marginally better than reduction of costs that is obtained with the naive parallelization of the exhaustive search approach. Additionally, the level of accuracy that was obtained using either exhaustive search or HSA is the same, even when HSA does not review the full feasible region. The tests with Struck and SiamMask trackers prove that the accuracy of those trackers surpasses the accuracy of ZNCC as a fitness function. But, HSA was able to reduce the time costs of exhaustive ZNCC, up to the point that this time costs are in the same order of magnitude of the time costs of Struck and SiamMask. These results and the theoretical analysis of Section 4.3 show the potential of the general parallelization approach described in Section 4.2.

However, these experiments also provide evidence that the CPU-GPU heterogeneous system is not the platform that should be used to parallelize a meta-heuristic such as HSA. As commented on Section 2.4.1, the GPU device is designed to perform several tasks at once but excels when those tasks can be performed independently without any external influence. Synchronization is possible but expensive on a common GPU device. These characteristics are not flaws in the design because the common GPU is intended to be used on image rendering. In contrast, population based meta-heuristics are mainly based around the concept of coordination either by collaboration or competition. This is why global synchronization is a must for meta-heuristics. In this specific case, HSA was unavailable to use the full capacity of the GPU which calls into question the fairness of the comparison between parallel HSA and parallel exhaustive search.

This chapter also introduced the idea of the Score-Time Efficiency metric and tested it. This metric was found useful to make decisions based on a single variable that summarizes the effects of accuracy and time cost measurements, particularly when the number of options grows considerably. The comparisons between parallel HSA and sequential HSA in terms of Score-Time Efficiency shows that the observed reduction of time costs is still not enough to cause a significant effect on efficiency. The main limitations of the CPU-GPU heterogeneous system that were observed are the following: the reduced instruction set of the GPU cores, the high-latency of the CPU-GPU communication, and the limited scope of synchronization between processes. This same parallelization approach could be tested on a SoC-FPGA heterogeneous system to overcome the limitations of the CPU-GPU heterogeneous system. Given that the SoC-FPGA heterogeneous system provides the flexibility to use the programmable fabric to design custom processing elements, and the possibility of using low-latency communication interfaces.

Chapter 5

Development of a SoC-FPGA Heterogeneous System



The experiments of the previous chapter revealed that some common characteristics of the CPU-GPU heterogeneous system are not compatible with HSA or other meta-heuristics. This chapter proposes the hypothesis that a SoC-FPGA heterogeneous system is a better suited solution to perform video tracking using HSA and ZNCC. Section 5.1 comments what are the observed limitations of the CPU-GPU heterogeneous system and what are the expected advantages of using a SoC-FPGA heterogeneous system. This chapter uses the codesign methodology that is described in Section 5.2 to design the system. The basic hardware and software materials that were used are discussed in Section 5.3. How the system was built using those materials is explained in Section 5.4. Finally, the designed system is tested with the experiments presented in Section 5.5.

5.1 Justification

The main processing elements of the CPU-GPU heterogeneous system, the CPU and GPU, are separate hardware products while a SoC-FPGA is actually a single chip that contains: the Processing System (PS) and the Programmable Logic (PL). The close proximity of these processing elements allows a faster and energy efficient communication that would be impossible on a CPU-GPU heterogeneous system. The PS of a SoC-FPGA is commonly based on ARM technology which is a reliable, and energy efficient standard RISC architecture that is popular for embedded solutions focused on signal and image processing (Churiwala and Hyderabad, 2017). ARM processor technology is also closely tied to AXI which is a communication interface that was mainly designed for on-chip communication (Sadri et al., 2013). The AXI-based communication of the SoC-FPGA should outperform the speed of CPU-GPU communication based on either USB or PCI express.

The GPU is optimized to work on parallel vector operations that do not require a lot of coordination. In that sense, a GPU is better suited to perform exhaustive search

instead of HSA. However, since the PL of a SoC-FPGA provides a greater malleability to implement a custom coordination mechanism and custom parallel processing units, it is expected that this heterogeneous computing system allows a faster execution of parallel HSA for video tracking, which is this thesis main goal. Nonetheless, there are other expected advantages of using a SoC-FPGA heterogeneous system such as the lower power requirements, the portability of the system, and the possibility of real-time processing using an embedded solution.

5.2 Codesign Methodology

Hardware/software codesign is the design of cooperating hardware components and software components in a single design effort (Wolf, 2003). This kind of methodology emerged because the design and optimization of hardware and software are radically different tasks. While software developers are trained to think in terms of sequential decomposition in time, hardware developers are trained to think about parallel decomposition in space. The codesign methodology integrates traditional top-down and bottom-up methodologies because both are used at some point of the design process. When using a top-down approach the developer starts with a top-level view of the application and investigates how to improve its performance using certain resources (Riesgo et al., 1999). In the opposite case, using the bottom-up view the developer starts with the resources and investigates how to integrate them efficiently into a working system (McFarland and Kowalski, 1990).

However, the terms hardware and software can mean different things in different contexts. This is why some authors redefine this methodology as the partitioning and design of an application in terms of fixed and flexible components (Schaumont, 2012). In this specific case, the system design is implemented using a SoC-FPGA that blurs the line between hardware and software. The FPGA of a SoC-FPGA is a fixed component that can be reconfigured to behave as a custom hardware design. In strict terms, the configuration of the FPGA hardware is made through software, because this configuration is lost once the power source is off.

For the most part, the hardware modules that were designed (Section 5.4) follow the top-down methodology, given that the fitness function (ZNCC) was respected and the fixed components were used to compute that function adequately. However, the bottom-up methodology had some influence in the design process. The fitness function was slightly modified to fit the available resources. The modifications to the fitness function are described with detail in Section 5.4.7.

5.3 Materials

This section identifies the hardware and software resources that are available to be used on different tasks of the system. The experiments of this work use the ZC706 evaluation board that contains a Zynq-7000 SoC-FPGA (Xilinx, 2018) which has been used successfully for image processing (Kechiche, 2021), signal processing (Rabiai et al., 2020), Internet of Things applications (Qureshi and Munir, 2020), etc. The PS

of the Zynq-7000 contains dual-core ARM Cortex-A9 processors while the PL of this SoC has the properties of a Kintex-7 FPGA (Xilinx, 2020). The analysis of Chapter 3 evidences that the evaluation of the fitness function is the part of the algorithm that benefits the most from hardware acceleration. The proposal of this work is that the PL should focus on the acceleration of the fitness function (ZNCC), while other activities of HSA that determine the intelligent behaviour of the system should be executed using the PS.

5.3.1 Programmable Fabric

Section 2.4.2 explains that the CLB, IOB and switch matrix are the top level building blocks of the FPGA. However, a given CLB is actually built from several smaller components that are actually used to implement the custom logic. This section describes the main components that the PL programmable fabric uses as basic building blocks to implement bigger custom logic designs. The custom system design that is presented in this chapter uses several different components that are available in the Zynq-7000 PL but the vast majority of them is either a LUT or a DSP48.

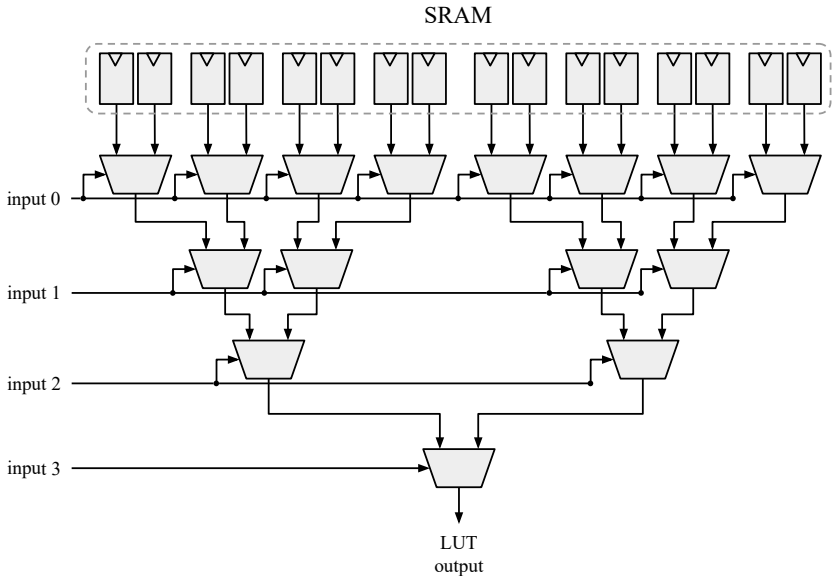


Figure 5.1: The LUT is the core component of any FPGA. All the possible output values can be written on the SRAM which allows the LUT to implement any Boolean function of four input signals. Each module in the figure is a 2×1 multiplexer.

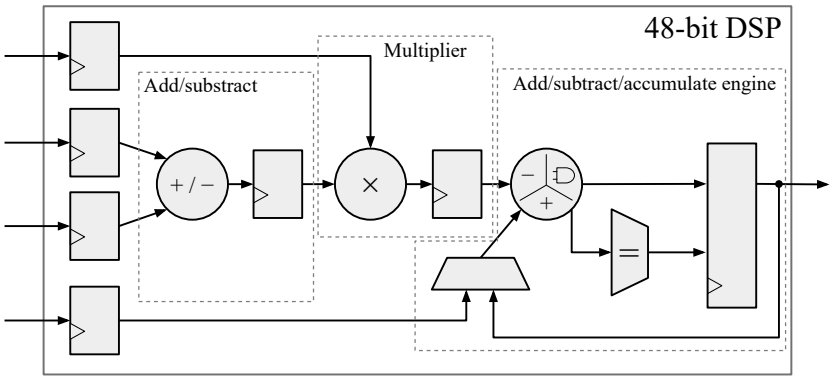


Figure 5.2: The DSP48 block ALU. This block is a chain of three blocks: an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine.

5.3.1.1 LUT

The LUT (LookUp Table) is the core component of the CLB which was introduced in Section 2.4.2. Each LUT (Figure 5.1) is actually an array of 1 bit multiplexers (Jahanirad and Karam, 2017) that has four 1 bit inputs and is capable of implementing any arbitrary Boolean function that has that number of inputs. Also, every LUT has a flip-flop that can be used to introduce sequential behaviour if necessary. The Boolean function of a given LUT is expressed as a truth table that is written as required in SRAM (Static RAM) registers. An FPGA contains a great number of LUT components to allow the implementation of complex combinational logic circuits using the switch matrices. The Zynq-7000 SoC of the ZC706 board has a LUT count of 218,600.

5.3.1.2 DSP48

The DSP48 block is a physical ALU embedded into the fabric of Xilinx FPGA devices and is composed of a chain of three different blocks. This chain contains an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain (Figure 5.2) allows a single DSP48 unit to implement arithmetic functions that require a combination of such operations. The multiplier block can handle one input value of up to 25 bits and a second input of 18 bits. Given that multipliers are components that use a great deal of resources from the programmable logic fabric (LUT components), using a DSP can generate savings in area and power consumption while allowing the designer to use the programmable fabric in other tasks. The Zynq-7000 SoC of the ZC706 board has 900 DPS48 blocks to be used as required by the designer.

5.3.2 Soft Intellectual Property Cores

An Intellectual Property (IP) Core is a reusable unit of logic, cell, or integrated circuit layout design that is the intellectual property of one party (like Xilinx). Soft IP cores are commonly offered as HDL files written with Verilog or VHDL. These allow designers

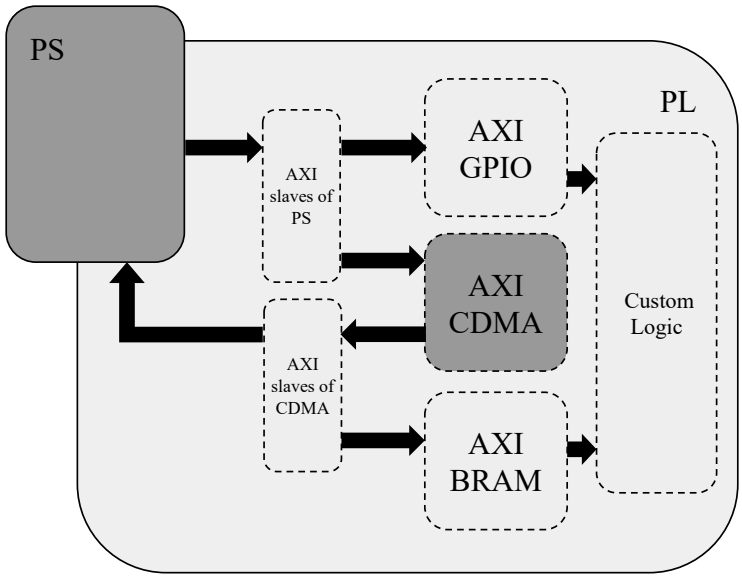


Figure 5.3: A general overview of the AXI channel. The PS acts as master and slave of the AXI CDMA block to send orders but also to allow direct access to the PS memory.

to easily implement those same cores on any FPGA to accelerate the design of a system. This section will describe some IP Cores that were used in the design of the system that is described in this chapter. The IDE that was used in the development process of the system was Xilinx Vivado 2017.3 which provided access to the soft IP cores that are described in following sections.

5.3.2.1 AXI and DMA

AXI defines a protocol where several master and slave components interact using parallel, high-performance, synchronous, high-frequency transfer operations. Xilinx has adopted AXI4, AXI4-Lite and AXI4-Stream (ARM, 2011) as the main communication interfaces that are used by their products. The PS can act as an AXI master or an AXI slave depending on the specific AXI peripherals that are implemented using the logic fabric. The simplest Xilinx AXI peripheral is the AXI GPIO (General Purpose Input Output) that simply reads or writes a register of 32 bits that are directly connected to physical hardware such as an output LED array, an input switch array, or other circuits implemented on the PL.

However, in situations where the data that is being transferred between PS and PL is greater than a few bytes, GPIO is not a viable alternative. In such cases, the AXI CDMA (Central DMA) peripheral can be used to act as the AXI master of the PS (Figure 5.3) in order to read the data directly from PS memory and write it directly to the memory of other AXI slave peripherals. The main benefit of using DMA is that a dedicated processor performs the data transfer in just a fraction of the time while the

PS is free to continue working on other tasks. To provide the PS with control over the data that is transferred using DMA, the PS acts as the AXI master of the AXI CDMA peripheral, but the only data that the PS sends to the AXI CDMA is a request to move a given number of bytes from a given memory address to a given destiny address.

5.3.2.2 CORDIC block

The CORDIC algorithm, or Volder's algorithm (Volder, 1959), is a method that uses simple shift-add operations to perform a series of two-by-two matrix multiplications in order to compute trigonometric functions, hyperbolic functions, or square roots that can be implemented using low level software or hardware. Xilinx allows the user to implement this efficient algorithm on the programmable fabric of their FPGA devices by providing a reproducible Intellectual Property core. However, some important limitations of this resource are that it can not use the DPS48 blocks that are available in the SoC; and that the maximum size of the input is 48 bits. The system that is presented in this chapter uses the CORDIC core to compute square roots as required to calculate a given ZNCC(p,q) using Equation (2.7)

5.3.2.3 Divider block

The Divider soft IP core provided by Xilinx can use three different division algorithms: LUTMult, Radix-2 and High Radix (Xilinx, 2021). Of these three available algorithms the Radix-2 algorithm was selected because it delivers the actual integer quotient of the division. LUTMult is considered a fast solution but it uses memory resources that should instead be used to store image data. The High Radix algorithm was also considered because it can be accelerated using the DSP48 blocks. However, using the High Radix algorithm produces an estimated integer output with error (Chen et al., 2017). That error can be greater than 1 which could have the negative effect of reducing the definition in the evaluation of ZNCC.

5.4 System Design

This section describes the parts of the system that were built using the fixed components that were described in previous sections. Figure 5.4 shows an abstract overview of the designed system. The main blocks that are used to perform arithmetic and logic operations are the Average blocks (Section 5.4.5), SSE blocks (Section 5.4.6) and RSSE blocks (Section 5.4.7). These modules perform the operations that are required but the sequential flow is controlled using finite state machines implemented as Control Units. Several instances of these components can be implemented using the PL which allows to compute several ZNCC values simultaneously. Since the designed system performs the same operations but on different input data, it can be classified as a SIMD device. The final iteration used in the experiments implements five Average blocks, one SSE_I block, four SSE_I blocks, and four RSSE blocks to compute four ZNCC values in parallel. A higher degree of parallelization is theoretically possible however the computer that was

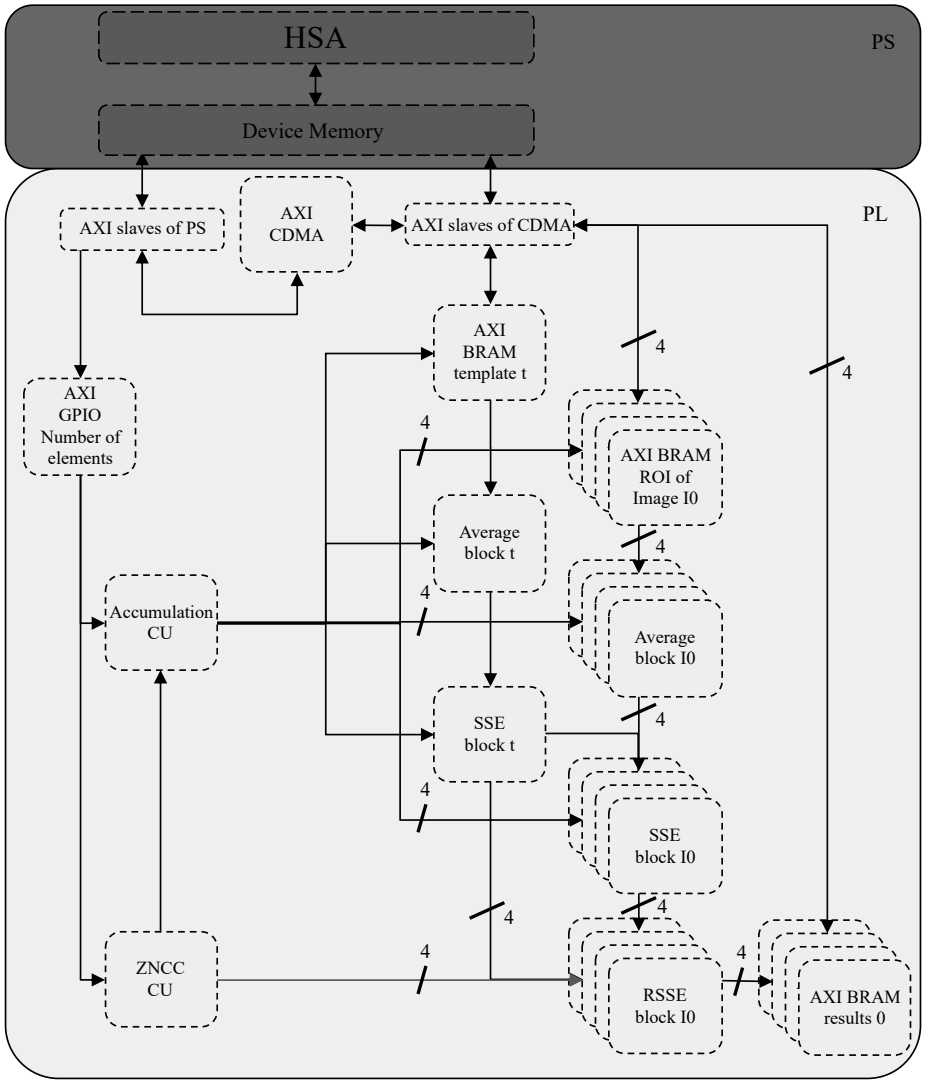


Figure 5.4: Top level blocks of the ZNCC processing unit. The main blocks that were implemented using the PL are: Average blocks, SSE blocks, RSSE blocks, the Accumulation CU and the ZNCC CU.

used to convert the design into a configuration file for the FPGA was not able to perform the process with higher requirements.

The Control Unit that guides the overall procedure to get a ZNCC value is the ZNCC CU (Section 5.4.3), it follows the right order to compute equations (2.2), (2.3), (2.4), (2.5), (2.6) and (2.7) but allows some of this operations to be simultaneous. Equations (2.2) and (2.3) are computed simultaneously using the Average blocks. Then, equations (2.4), (2.5) and (2.6) are computed simultaneously using SSE blocks. Finally, Equation (2.7) is computed using RSSE blocks.

The second Control Unit, the Accumulation CU (Section 5.4.4) was designed because several of the sequential operations to compute a ZNCC value are summations, this same Control Unit is used to guide two different summation phases that are considered in the overall flow that is given by the ZNCC CU.

Section 5.4.1 describes the setup of the PS which is in charge of running the HSA meta-heuristic. The communication between PS and PL (Section 5.4.2) is done using AXI interfaces. The communication of image data uses DMA to allow a higher throughput. However, the signal to start working on the evaluation of the fitness function (ZNCC) is provided using a simple AXI GPIO that also provides the number of elements that are written on the input arrays; whenever this 4 byte integer is zero a reset signal is fired.

5.4.1 PS Development Environment

The dual-core processor of the Zynq-7000 SoC can be programmed directly using assembly or low level languages but this project uses a lightweight Linux Operating System installation to allow the PS to serve as a development environment along with compilation tools and useful software libraries such as OpenCV to easily manipulate image files. Also, the Operating System builds a logic layer that allows the use of board interfaces such as UART, USB, Ethernet, etc. A program running on Linux can interact with the physical memory that is directly used by the computer hardware, which is different to the virtual memory that is commonly used by programs. The interaction of a program with physical device memory is achieved through a file that is an image of the main memory which can be located following the path /dev/mem, using the common root folder structure. In this case, the /dev/mem file is used to read and write data that is also manipulated by the PL, creating a bridge between PS and PL. As a reminder, the PS will focus on the activities of HSA that determine the intelligent behaviour of the system while the PL will focus on the acceleration of the fitness function (ZNCC).

5.4.2 Image Data Communication

ALOV image files are stored in JPEG format, however they are read and converted to a one-dimensional array of grayscale unsigned integer values of 8 bits using software. These values are then transferred from the PS to the PL and stored using standard BRAM resources. The word size of the used BRAM is 32 bits, meaning that 4 grayscale values (of 1 byte each) can be written or read at once. The maximum capacity of BRAM resources is 8,192 bytes (2^{13}) or 8 KiB and the data transfer is made using CDMA

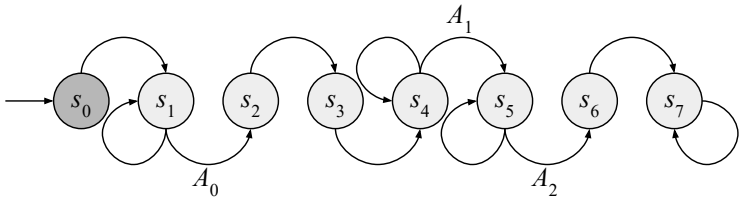


Figure 5.5: State diagram of the State Machine of the ZNCC Control Unit

blocks that read the image values from the PS and write them to the corresponding BRAM resource in the PL.

5.4.3 State Machine of the ZNCC Control Unit

This CU controls the overall flow of the custom ZNCC processor. It has control over the Accumulation CU and the elements of the datapath. In this case, the Moore Machine used to describe the CU has eight states $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$. The tasks that are performed in each of these states are: clear the average accumulators to start from zero (s_0); work to get averages $\bar{t}$ and $\bar{I}(p,q)$ using the Accumulation CU (s_1); save the averages to use them later (s_2); clear the SSE and ZCC accumulators to start from zero (s_3); work to get the SSE and ZCC values using the Accumulation CU (s_4); work to compute Equation (2.7) to get the ZNCC value (s_5); save the ZNCC value (s_6); standby and wait for a hard reset (s_7).

The input alphabet A has three input characters: a signal that indicates that the average is ready (A_0), a signal that indicates that the SSE is ready (A_1) and a signal that indicates that the ZNCC value is ready (A_2). The transition function T (Figure 5.5) is defined so that s_0 transitions to s_1 in all cases; s_1 transitions to s_2 when the A_0 input character is received and stays in s_1 otherwise; s_2 transitions to s_3 in all cases; s_3 transitions to s_4 in all cases; s_4 transitions to s_5 when the A_1 input character is received and stays in s_4 otherwise; s_5 transitions to s_6 when the A_2 input character is received and stays in s_5 otherwise; s_5 transitions to s_6 in all cases; and s_7 stays in s_7 in all cases.

The implementation of this CU also uses the sequence register and decoder method. Where the sequence register is a binary counter of 3 bits, given that the behaviour is linear. Input characters are interpreted as three binary signals that can be either true or false. The transition function is implemented as a Boolean function that returns true only when the binary counter should start counting based on the state and input. This transition function is only false when: the state is s_1 and the average done signal A_0 is not received; the state is s_4 and the SSE done signal A_1 is not received; the state is s_5 and the average done signal A_2 is not received; the state is s_7.

5.4.4 State Machine of the Accumulation Control Unit

The algorithm to get ZNCC requires the evaluation of several summations described in Equations (2.2), (2.3), (2.4), (2.5) and (2.6). This CU (Control Unit) is used to perform

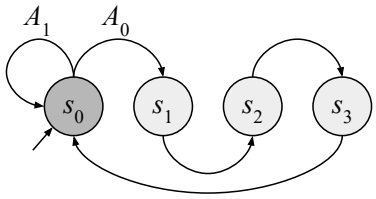


Figure 5.6: State diagram of the State Machine of the Accumulation Control Unit

the sequential accumulation of integers in two cases: to get the sums that will be used to evaluate $\bar{t}$ and $\bar{I}(p,q)$; and to get SSE_t and $\text{SSE}_I(p,q)$ and $\text{ZCC}(p,q)$. Note that $\bar{t}$ and $\bar{I}(p,q)$ can be evaluated in parallel, but SSE_t and $\text{SSE}_I(p,q)$ and $\text{ZCC}(p,q)$ need those values to be computed previously. This is why the design considers two accumulation phases.

The Moore Machine that describes this CU has four states $S = \{s_0, s_1, s_2, s_3\}$. Where s_0 is the state in which the stop condition is checked, if the full array of values has been iterated the stop condition should be true. Also, during state s_0 the CU waits for the BRAM to deliver the piece of data that is being currently requested. Then, in state s_1 the data that has been received from the BRAM is multiplied to obtain the square, only if needed. The next step is to accumulate the value or the squared value while the CU is in the s_2 state. Finally, state s_3 is where a signal is provided to indicate that the index of the process should be incremented.

The input alphabet A only has two input characters, the not done signal A_0 and the done signal A_1. These input characters are used to indicate when the stop condition has been met, which is that the index is greater than the length of the array. The output alphabet B has four characters, one for every state, making the definition of this alphabet and the output function G a trivial matter. The transition function T (Figure 5.6) is also quite simple. The state s_0 transitions to s_1 only if the stop condition is not met $T(s_0, A_0) = s_1$, and stays in s_0 if the stop condition is met $T(s_0, A_1) = s_0$. The rest of the transitions are always from s_1 to s_2, s_2 to s_3 and s_3 to s_0 no matter the input.

In general terms, the behaviour of this CU is very similar to a 'for loop' in common programming. The implementation of this CU uses the sequence register and decoder method. Where the sequence register is a binary counter of 2 bits, given that the behaviour is cyclical. Both input characters are summarized into a single binary signal that can be either true or false. The transition function is implemented as a Boolean function that returns true only when the binary counter should start counting. This transition function is only false when on the s_0 state and receiving the done signal A_1.

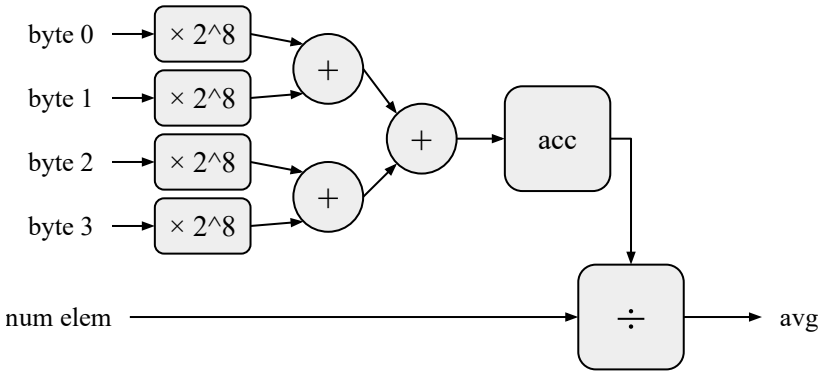


Figure 5.7: Average block.

5.4.5 Average Block

This block is used to get the average of a set of 8 bit positive integers (Figure 5.7). Four input bytes are received at once to adapt to the BRAM component that is used to store the array of input data. Each of these bytes is multiplied by 2^8 by using a hardwired shift left of 8 bits. These 16 bit unsigned integers are then added using 3 simple binary adders. The sum of those 4 integer numbers is the input of the main sequential accumulator that follows the instructions of the Accumulation CU and ZNCC CU. The 32 bit output of the accumulator is the dividend input of a sequential divider that uses the Radix-2 algorithm. The divisor input of the same divider block is the length of the array of numbers. The signal to start working on the division is given by the accumulation stop condition and the ZNCC CU. Once the division is over, an output signal is given, this is the average done signal. The average is a 16 bit unsigned integer number that is stored in a register. All the components of this block are implemented using LUT programmable fabric. The system should have at least two of these blocks, one to get $\bar{t}$ and one to get $\bar{I}(p,q)$. However it is possible to use more blocks to compute several $\bar{I}(p,q)$ values in parallel.

5.4.6 SSE Blocks

This block has two variations: one used to get SSE_t, and another one that gets both $\text{SSE}_I(p,q)$ and $\text{ZCC}(p,q)$ in parallel (Figure 5.8). The four input bytes receive the same treatment of the Average block to generate four 16 bit unsigned integers. The corresponding average that was previously obtained is subtracted from each of the integer values to get the individual error values. In the case of the SSE_t block, error values are provided as output so they can be used by other blocks to get $\text{ZCC}(p,q)$. The block that computes both $\text{SSE}_I(p,q)$ and $\text{ZCC}(p,q)$ has eighth 17 bit multipliers, four of them are used to get the squared errors, and the rest are used to multiply the errors of t with the corresponding errors of I. The block that gets SSE_t only has four multipliers to get the squared errors. Both variations have 48 bit accumulators that are controlled by the Accumulation CU and the ZNCC CU. However the SSE_t has only one while the

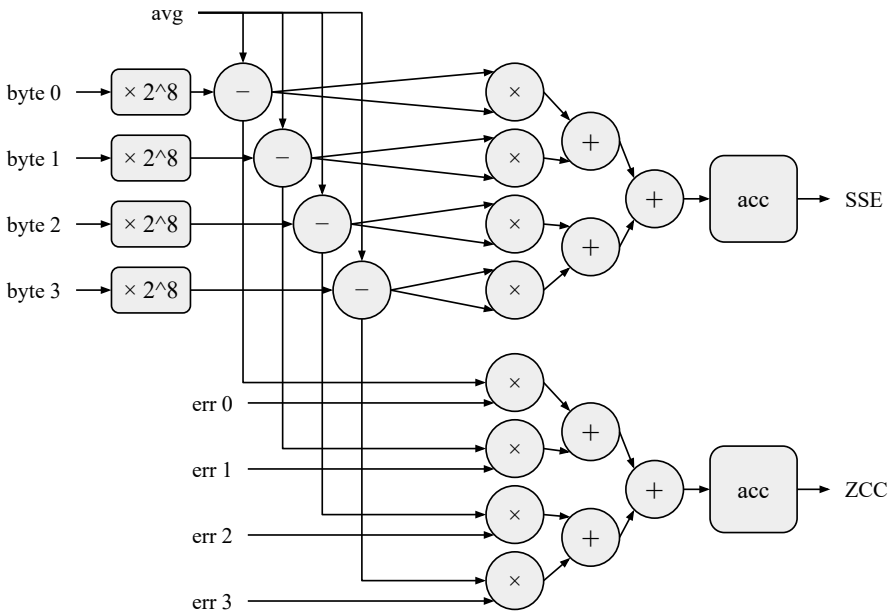


Figure 5.8: SSE block.

second variant has two accumulators, one for $\text{SSE}_I(p,q)$ and the second for $\text{ZCC}(p,q)$. In order to work, the system requires at least one block of the first variant and one of the second variant. To work in several $\text{SSE}_I(p,q)$ and $\text{ZCC}(p,q)$ values in parallel, several blocks of the second variant are required. The multipliers are implemented using the DSP48 components while the rest of the blocks are LUT-based.

5.4.7 RSSE Block

Working with integers introduces some complications such as the loss of significant figures when the result of an operation has a decimal part. However, floating point operations use a great deal of resources and are avoided in this design. This is why ZNCC is adapted to work with integers only. The CORDIC block is used to obtain the truncated integer square root of a 48 bit input integer. In order to compute Equation (2.7) faster without losing too much detail, this work proposes to use some simple algebraic manipulations. Equation (5.1) shows how to obtain the squared $\text{ZNCC}(p,q)$ value using two parallel divisions and one multiplication (Figure 5.9). However, the $\text{ZCC}(p,q)$ value can be either positive or negative. To avoid confusion and save resources, this work also proposes to ignore the negative $\text{ZCC}(p,q)$ values which are considered undesirable in the fitness function because they indicate a negative correlation of the compared images. This is implemented using a simple multiplexer that outputs zero whenever a negative ZCC value is detected.

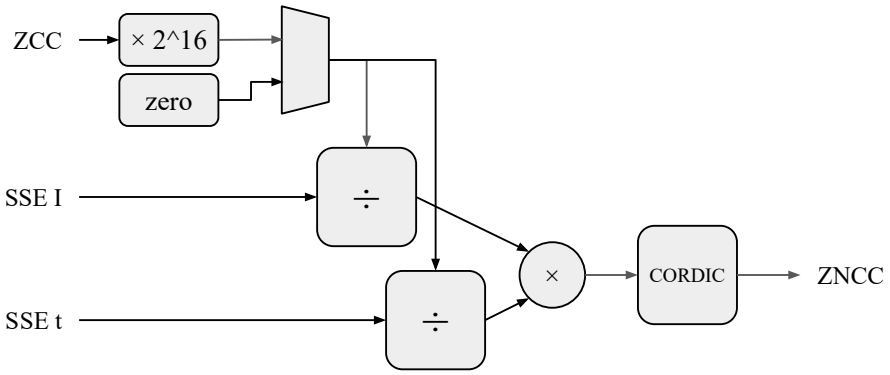


Figure 5.9: RSSE block.

$$\mathrm{ZNCC}^2(p,q) = \frac{\mathrm{ZCC}(p,q)\cdot\mathrm{ZCC}(p,q)}{\mathrm{SSE}_I(p,q)\cdot\mathrm{SSE}_t} = \frac{\mathrm{ZCC}(p,q)}{\mathrm{SSE}_I(p,q)}\cdot\frac{\mathrm{ZCC}(p,q)}{\mathrm{SSE}_t} \tag{5.1}$$

Since this design uses integer values, the output of this block is easier to handle if the value ranges from 0 to 2^{16}, or from 00000 to 10000 using hexadecimal numbers. Then, the output value is transformed to $\lfloor \mathrm{ZNCC}(p,q)\cdot 2^{16} \rfloor$ when $\mathrm{ZNCC}(p,q)$ is positive and 0 otherwise. The input ZCC value of this block is multiplied by 2^{16} before division to avoid the loss of resolution when the result is truncated. There are two divider blocks with 64 bit dividends and 48 bit divisors. The quotients of the divider blocks are the inputs of a 24 bit multiplier that is used to compute the square of the ZNCC value. Once the dividers provide a positive done signal, a single CORDIC block starts working on the square root of the squared ZNCC value. The system needs at least one of these blocks, but several can be used to estimate a number of $\mathrm{ZNCC}(p,q)$ values in parallel.

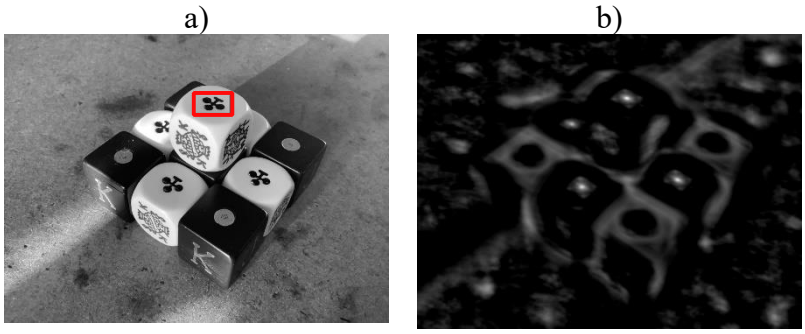


Figure 5.10: Static image used for calibration. a) The image in grayscale with the object (ace of clover) inside a bounding box. b) A visual representation of ZNCC values, brighter spots mark the greatest ZNCC values, note that similar objects are spotted.

5.5 Experiments with SoC-FPGA

The system that was used in these experiments is configured to use the blocks that are defined in previous sections of the chapter. The implemented blocks are: five Average blocks, one SSE_t block, four SSE_I blocks, and four RSSE blocks which are interconnected as illustrated in Figure 5.4. The clock frequency of the custom logic blocks is of 225 MHz, 450 MHz for the AXI peripherals, and 800 MHz for the PS. The implemented design uses 40% of the LUT programmable fabric, and 7% of the available DSP blocks.

5.5.1 Calibration with a Static Image

This section describes a simple test that was performed to check if the proposed design behaves as expected. The range of ZNCC values includes real numbers from -1 to 1 (Section 2.2.2). However the custom hardware design proposed in this chapter modifies this range to use the resources effectively. The modifications are: negative numbers are ignored, the range extends from 0 to 2^{16}, and the values are discrete integer numbers. This experiment is intended to confirm that these modifications have no negative effects on the accuracy of the ZNCC tracker.

This test used the custom ZNCC unit to perform an exhaustive search of a simple template within the same image frame where it is originally located. The static image that was used is shown in Figure 5.10a. Given that the only point with 100% similarity is the same spot where the initial object is located, the test is considered successful. A closer inspection of the ZNCC values (Figure 5.10b) is convincingly accurate from a qualitative standpoint because the other similar instances of the object are also marked as bright spots.

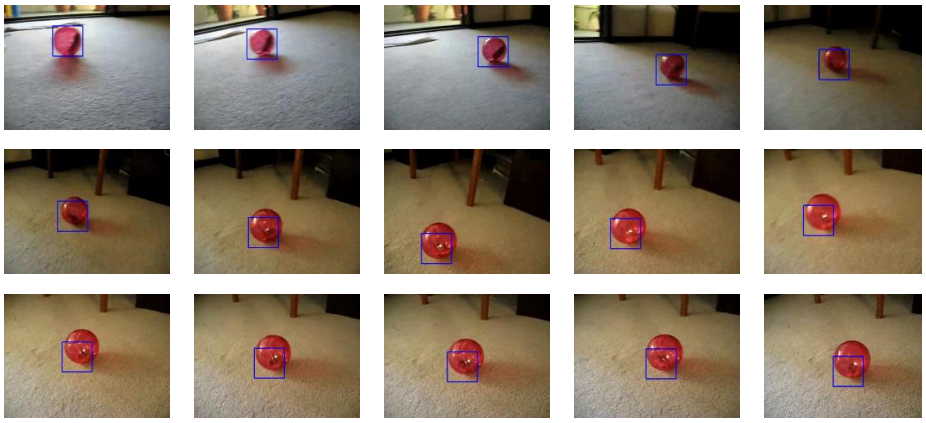


Figure 5.11: Exhaustive search with video. Some frames of the video to illustrate how the tracker follows the object of interest.

5.5.2 One Video with Exhaustive Search

Given that the previous test proves that the proposed ZNCC unit can find an object with a 100% similarity, this test confirms that the video tracking results are as accurate as the ones obtained with the CPU-GPU heterogeneous system (Section 4.5.1). The test uses only one video given the high time costs that are observed using exhaustive search. The selected video is the twenty third video of the Light category from the ALOV benchmark, which follows a hamster ball as it rolls on the floor of the scene (Figure 5.11). The attained F-score is of 0.975, which is the exact same result that was obtained using the CPU-GPU heterogeneous system. This shows that the proposed ZNCC unit can recognise a visually similar object even if the similarity is not perfect in the scale of ZNCC.

However, it is also observed that the execution times of exhaustive search are lower when using a GPU. The average time per frame of processing this video using the CPU-GPU heterogeneous system is 0.0503 seconds per frame. While the time cost using the SoC-FPGA with the proposed configuration is 5.0004 seconds per frame. The superiority of the CPU-GPU heterogeneous system in this setting is expected, given that the GPU is the ideal architecture for exhaustive search and that the full capacity of the GPU is used in that case. Following test will focus on HSA.

5.5.3 CPU-GPU versus SoC-FPGA Using HSA

The SoC-FPGA heterogeneous system was used to process 309 videos of the ALOV benchmark (Section 2.2.1), the PL computes ZNCC to be used as fitness function and the PS is in charge of the HSA meta-heuristic. The comparison in terms of accuracy using the F-score (Figure 5.12a) shows that there is a small difference between using the SoC-FPGA heterogeneous system or the CPU-GPU heterogeneous system. However, that difference is only the 6.6% of the overall standard deviation, which is negligible. This means that the implemented design does not have important effects on accuracy.

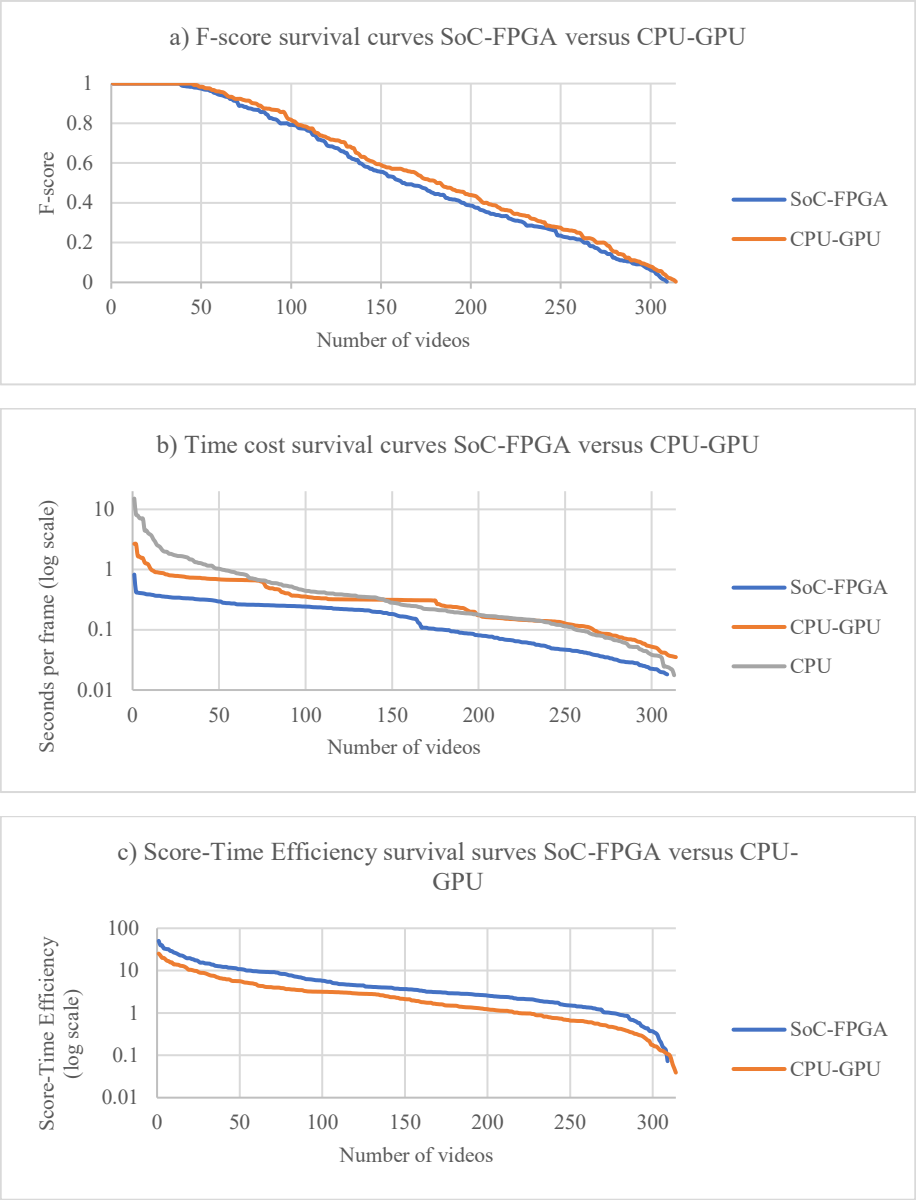


Figure 5.12: SoC-FPGA versus other architectures. a) SoC-FPGA versus other architectures on accuracy. b) SoC-FPGA versus other architectures on time costs. c) SoC-FPGA versus other architectures on Score-Time Efficiency.

The previous computational complexity analysis of Section 3.3 and Section 4.3 evidences that the size in pixels of the object of interest has an important effect on time costs, this fact influences the following statements about the observed time costs. In terms of time costs, the SoC-FPGA shows an advantage over CPU-GPU and CPU (Figure 5.12b). The average time per frame that was obtained using the SoC-FPGA is 0.1697, lower than the average of 0.3729 that was obtained using the CPU-GPU heterogeneous system. Also, the standard deviation of the time per frame measurements obtained using the SoC-FPGA is 0.1199 while the CPU-GPU heterogeneous system is 0.3461 which shows that the SoC-FPGA is about 2.8 times more stable. Note that CPU-GPU provides lower time costs than CPU only with large frame sizes, while the SoC-FPGA provides lower time costs consistently.

Using the Score-Time Efficiency metric shows that the improvements that were achieved using the SoC-FPGA heterogeneous system efficiently uses the resource of time (Figure 5.12c). These results validate that the SoC-FPGA architecture is better suited to implement a parallelized meta-heuristic like HSA. However, the evidence also suggests that the GPU is the ideal architecture to conduct an exhaustive search.

Using the SoC-FPGA allows to increase the speed of this video tracker proposal and allows real-time processing with certain frame sizes (Figure 5.13a). As a reminder, the speed of 30 frames per second was defined as the frontier of real-time video tracking. Furthermore, the SoC-FPGA is more efficient in terms of power consumption (Figure 5.13b). The CPU-GPU heterogeneous system consumes approximately 40 times more power that the SoC-FPGA heterogeneous system in this particular setting. This indicates that the SoC-FPGA heterogeneous system in combination with HSA is an efficient alternative to accelerate a video tracker.

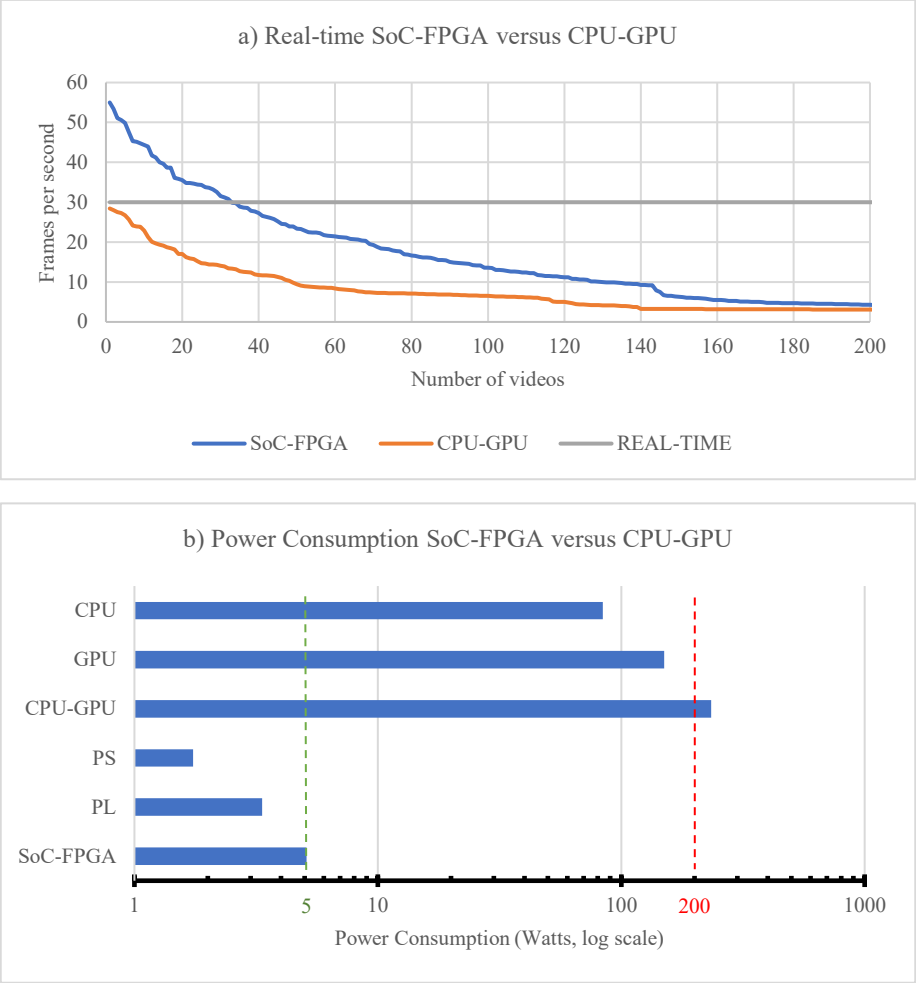
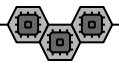


Figure 5.13: SoC-FPGA power consumption and real-time video tracking. a) SoC-FPGA versus CPU-GPU in terms of real-time video tracking. b) SoC-FPGA versus CPU-GPU in terms of power consumption.

Chapter 6

Discussion and Conclusions



The HSA meta-heuristic was selected to be used in this work because it is oriented to CV applications. The experiments prove that HSA can successfully be used to accelerate ZNCC for video tracking using CPU, CPU-GPU and SoC-FPGA. The results of the experiments with CPU and HSA are comparable to the results of the experiments with the CPU-GPU heterogeneous system and exhaustive search. Using HSA has advantages over CPU-GPU because it can be easier to use and prevent the heavy power consumption costs of a GPU. A recommendation of this work is to consider using a meta-heuristic over a GPU whenever possible.

However, the combination of HSA with different heterogeneous computing systems shows benefits in terms of time costs. The greatest reduction of time costs was observed when HSA was used in combination with SoC-FPGA. Even though the CPU-GPU heterogeneous system can be used in combination with HSA, the resources are not used efficiently. Certain limitations of the internal coordination mechanisms of GPU devices are not compatible with meta-heuristics in general. Those limitations (Section 5.1) are: the high time costs of CPU-GPU communication; the limited coordination of GPU components; the fixed nature of the CPU-GPU architecture. The implemented solution based on CPU-GPU and HSA used only a fraction of the available GPU cores while most of them remained idle. The CPU-GPU heterogeneous system was created to accelerate massive rendering operations. According to the experiments of this work, CPU-GPU is better suited to perform exhaustive search. Another general recommendation of this work is to use CPU-GPU over SoC-FPGA only if the problem requires exhaustive search and the solution does not require portability.

Several benefits were observed using HSA in combination with SoC-FPGA for video tracking. The SoC-FPGA allows a greater control of the datapath. The modules of the atomic operations can be used as required by the specific problem. In the case of the fitness function, the problem was reassessed to fit the available resources in a bottom-up fashion. Time costs are lower using a SoC-FPGA which makes real-time processing possible. The speed of 30 fps can be achieved using HSA and SoC-FPGA. However, it is only possible when the object of interest is contained on a rectangle that contains at most 3,800 pixels. Still, SoC-FPGA makes it possible to process a greater frame size in real-time. Additionally, SoC-FPGA allows lower power consumption

and a greater portability. Also, the power consumption of SoC-FPGA is noticeably lower. Furthermore, the SoC-FPGA heterogeneous system provides a greater portability. One of the few disadvantages of using SoC-FPGA is that complex iterative codesign is required. However, introducing HSA simplifies the hardware design process. Given that HSA does not explore the full search space, it is not strictly necessary to transfer the full frame from PS to PL. In that sense, HSA reduces the demand of memory resources. As it can be observed, both tools fit together and the general recommendation of this work is to use SoC-FPGA over CPU-GPU to work with meta-heuristics.

To provide a direct answer to the main research question, a given video tracker does increase its speed and keeps its accuracy using a multiprocessing system based on HSA and an heterogeneous architecture platform (such as SoC-FPGA or CPU-GPU). More specifically, the SoC-FPGA platform is better in this case.

6.1 Contributions

The main contributions that have derived from this work are:

1. The journal articles of Perez-Cham et al. (2020) and Perez-Cham et al. (2021).
2. The international congress article of Perez-Cham et al. (2018).
3. An automaton that describes the behaviour of a honeybee that searches for food which is defined and studied in Chapter 3.
4. The design, implementation and evaluation of a CPU-GPU heterogeneous computing system that performs video tracking using HSA which is defined and studied in Chapter 4. This system is proposed following the restrictions of the rigid GPU architecture.
5. The design, implementation and evaluation of a SoC-FPGA heterogeneous computing system that performs video tracking using HSA which is defined and studied in Chapter 5. This system is designed from scratch using the codesign methodology.
6. A new metric that helps to compare video trackers in terms of accuracy and speed which is defined in Section 4.4.

6.2 Future Work

- Use HSA in other CV applications. The results of using HSA in the applications of three-dimensional scene reconstruction and video tracking show positive results. This motivates the proposal to study the effect of HSA when used on other CV applications or on specific variations of the tested problems. For example, face tracking and detection, or tracking based on infrared image data.

- Use the individual reallocation mechanism (Section 3.2) in other CV applications. The experiments with the applications of three-dimensional scene reconstruction showed that the individual reallocation mechanism can reduce resource consumption in certain applications. This should be studied further on compatible applications.

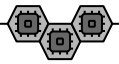
- Test if HSA can be implemented using the PL of the SoC-FPGA. The current proposal uses the PL to compute the fitness function, but the general decision making process is executed using the PS. Further experiments should be performed to find the advantages and disadvantages of using the PL to run HSA. Those experiments should be based on the Moore Machine that was proposed in Section 3.1 to study HSA.

- Improve the system that was designed to use the full capacity of the SoC-FPGA. The current proposal uses 40% of the LUT components, and 7% of the available DSP blocks. This configuration was the most complex design that could be generated by the computer equipment used for design and simulation. Also, the possibility of reconfiguration allows the designer to propose many different designs that solve the same problem but with varying degrees of sequential and parallel behaviour.

- Use other video trackers as fitness functions to replace or complement ZNCC. The canonical ZNCC tracker is currently not a viable contender against state-of-the-art trackers in terms of accuracy. However, it served as a starting point to study the advantages of using HSA and different heterogeneous systems for video tracking given its relative simplicity in comparison to newer proposals like Struck (Hare et al., 2015) or SiamMask (Liang et al., 2019).

- Consider the CPU-GPU-FPGA heterogeneous system. The experiments of this thesis focused on CPU-GPU and SoC-FPGA. However, the CPU-GPU-FPGA platform should also be studied.

References



Al-Zoubi, A., Tatas, K., and Kyriacou, C. (2018). Towards dynamic multi-task schedulling of OpenCL programs on emerging CPU-GPU-FPGA heterogeneous platforms: A fuzzy logic approach. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 247–250. IEEE.

Alberto, C. M. M. and Sato, H. (2015). Linear performance-breakdown model: A framework for GPU kernel programs performance analysis. *International Journal of Networking and Computing*, 5(1):86–104.

Anderson, J. A. (2006). *Automata theory with modern applications*. Cambridge University Press.

ARM (2011). *AMBA AXI and ACE Protocol Specification*.

Asano, S., Maruyama, T., and Yamaguchi, Y. (2009). Performance comparison of FPGA, GPU and CPU in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE.

Asgarizadeh, M., Pourghassem, H., and Shahgholian, G. (2012). Robust object tracking using regional mutual information and normalized cross correlation. In *2012 Fourth International Conference on Computational Intelligence and Communication Networks*, pages 411–415. IEEE.

Bätz, M., Richter, T., Garbas, J.-U., Papst, A., Seiler, J., and Kaup, A. (2014). High dynamic range video reconstruction from a stereo camera setup. *Signal Processing: Image Communication*, 29(2):191–202.

Bean, A. (2016). *Improving memory access performance for irregular algorithms in heterogeneous CPU/FPGA systems*. PhD thesis, Imperial College of Science.

Bitam, S., Batouche, M., and Talbi, E.-G. (2010). A survey on bee colony algorithms. In *2010 IEEE international symposium on parallel & distributed processing, workshops and phd forum (IPDPSW)*, pages 1–8. IEEE.

Blickle, T. and Thiele, L. (1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394.

Blum, C. and Merkle, D. (2008). *Swarm intelligence: introduction and applications*. Springer Science & Business Media.

Boumaza, A. M. and Louchet, J. (2001). Dynamic flies: Using real-time parisian evolution in robotics. In *Workshops on Applications of Evolutionary Computation*, pages 288–297. Springer.

Chen, L., Lombardi, F., Montuschi, P., Han, J., and Liu, W. (2017). Design of approximate high-radix dividers by inexact binary signed-digit addition. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 293–298.

Chen, Y. and Lin, W. (2009). An improved bacterial foraging optimization. In *2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2057–2062. IEEE.

Churiwala, S. and Hyderabad, I. (2017). *Designing with Xilinx® FPGAs*. Springer.

Clerc, M. (2010). *Particle swarm optimization*, volume 93. John Wiley & Sons.

Collett, D. (2015). *Modelling survival data in medical research*. CRC press.

Comaniciu, D., Ramesh, V., and Meer, P. (2000). Real-time tracking of non-rigid objects using mean shift. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000*, volume 2, pages 142–149. IEEE.

Crist, E. (2004). Can an insect speak? the case of the honeybee dance language. *Social Studies of Science*, 34(1):7–43.

Davidson, A., Tarjan, D., Garland, M., and Owens, J. D. (2012). *Efficient parallel merge sort for fixed and variable length keys*. IEEE.

Deb, K. (2011). Multi-objective optimisation using evolutionary algorithms: an introduction. In *Multi-objective evolutionary optimisation for product design and manufacturing*, pages 3–34. Springer.

Deb, K. and Beyer, H.-G. (2001). Self-adaptive genetic algorithms with simulated binary crossover. *Evolutionary computation*, 9(2):197–221.

Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39.

Elkhani, N., Muniyandi, R. C., and Zhang, G. (2018). Multi-objective binary PSO with kernel P system on GPU. *International Journal of Computers Communications & Control*, 13(3):323–336.

Forsyth, D. A. and Ponce, J. (2012). *Computer vision: a modern approach*. Pearson,.

Galoogahi, H. K., Fagg, A., Huang, C., Ramanan, D., and Lucey, S. (2017). Need for speed: A benchmark for higher frame rate object tracking. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1125–1134.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2012). *Heterogeneous computing with OpenCL: revised OpenCL 1*. Newnes.

Golberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36.

Goldberg, D. E. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum.

Haddad, O. B., Afshar, A., and Marino, M. A. (2006). Honey-bees mating optimization (hbmo) algorithm: a new heuristic approach for water resources optimization. *water resources management*, 20(5):661–680.

Hamblen, J. O., Hall, T. S., and Furman, M. D. (2007). *Rapid prototyping of digital systems: SOPC edition*. Springer Science & Business Media.

Haralick, R. M. (1979). Statistical and structural approaches to texture. *Proceedings of the IEEE*, 67(5):786–804.

Hare, S., Golodetz, S., Saffari, A., Vineet, V., Cheng, M.-M., Hicks, S. L., and Torr,

P. H. (2015). Struck: Structured output tracking with kernels. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):2096–2109.

Held, D., Thrun, S., and Savarese, S. (2016). Learning to track at 100 fps with deep regression networks. In *European conference on computer vision*, pages 749–765. Springer.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Hoiem, D., Divvala, S. K., and Hays, J. H. (2009). Pascal VOC 2008 challenge. *World Literature Today*.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65.

Hrúz, B. and Zhou, M. (2007). Finite automata. *Modeling and Control of Discrete-event Dynamic Systems: with Petri Nets and Other Tools*, pages 49–62.

Hu, J.-L., Tang, X.-W., and Qiu, J.-N. (2017). Analysis of the influences of sampling bias and class imbalance on performances of probabilistic liquefaction models. *International Journal of Geomechanics*, 17(6):04016134.

Jahanirad, H. and Karam, H. (2017). BIST-based testing and diagnosis of LUTs in SRAM-based FPGAs. *Emerging Science Journal*, 1(4):216–225.

Kaeli, D. R., Mistry, P., Schaa, D., and Zhang, D. P. (2015). *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann.

Karaboga, D. and Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. *Journal of global optimization*, 39(3):459–471.

Kechiche, L. (2021). Hardware acceleration for deep learning of image classification. In *2021 International Conference of Women in Data Science at Taif University (WiDSTaif)*, pages 1–5. IEEE.

Khare, V., Yao, X., and Deb, K. (2003). Performance scaling of multi-objective evolutionary algorithms. In *International conference on evolutionary multi-criterion optimization*, pages 376–390. Springer.

Kristan, M., Leonardis, A., Matas, J., Felsberg, M., Pflugfelder, R., Kämäräinen, J.-K., Danelljan, M., Zajc, L. Č., Lukežič, A., Drbohlav, O., et al. (2020). The eighth visual object tracking VOT2020 challenge results. In *European Conference on Computer Vision*, pages 547–601. Springer.

Li, X., Hu, Y., Shen, T., Zhang, S., Cao, J., and Hao, Q. (2020). A comparative study of several template matching algorithms oriented to visual navigation. In *Opto-electronic Imaging and Multimedia Technology VII*, volume 11550. International Society for Optics and Photonics.

Liang, Z., Liang, C., Zhang, Y., Mu, H., and Li, G. (2019). Tracking of moving target based on SiamMask for video SAR system. In *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*, pages 1–4. IEEE.

Liberti, L., Lavor, C., Maculan, N., and Mucherino, A. (2014). Euclidean distance geometry and applications. *SIAM review*, 56(1):3–69.

Lin, C., Li, Y., Xu, G., and Cao, Y. (2017). Optimizing ZNCC calculation in binocular stereo matching. *Signal Processing: Image Communication*, 52:64–73.

Liu, Y. and Zhu, H. (2010). A survey of the research on power management techniques

for high-performance systems. *Software: Practice and Experience*, 40(11):943–964.

Luenberger, D. G. and Ye, Y. (1984). *Linear and nonlinear programming*, volume 2. Springer.

Maggio, E. and Cavallaro, A. (2011). *Video tracking: theory and practice*. John Wiley & Sons.

Mano, M. M. (2003). *Computer system architecture*. Prentice-Hall of India.

Manssen, M., Weigel, M., and Hartmann, A. K. (2012). Random number generators for massively parallel simulations on GPU. *The European Physical Journal Special Topics*, 210(1):53–71.

Mantor, M. (2012). AMD radeon HD 7970 with graphics core next (GCN) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–35. IEEE.

Martin, G. and Chang, H. (2001). System-on-Chip design. In *ASICON 2001. 2001 4th International Conference on ASIC Proceedings*, pages 12–17. IEEE.

McFarland, M. C. and Kowalski, T. J. (1990). Incorporating bottom-up design into hardware synthesis. *IEEE transactions on computer-aided design of integrated circuits and systems*, 9(9):938–950.

Morsi, N. N., Abdelhalim, M. B., and Shehata, K. A. (2013). Efficient hardware implementation of PSO-based object tracking system. In *2013 International Conference on Electronics, Computer and Computation (ICECCO)*, pages 155–158. IEEE.

Mozaffari, A., Gorji-Bandpy, M., and Gorji, T. B. (2012). Optimal design of constraint engineering systems: application of mutable smart bee algorithm. *International Journal of Bio-Inspired Computation*, 4(3):167–180.

Nakhmani, A. and Tannenbaum, A. (2013). A new distance measure based on generalized image normalized cross-correlation for robust video tracking and image recognition. *Pattern recognition letters*, 34(3):315–321.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53.

Nogueira, B. and Barboza, E. (2020). A FPGA-based accelerated architecture for the continuous GRASP. *Computing*, pages 1–20.

NVIDIA (2013). CUDA C programming guide.

Olague, G. and Puente, C. (2006a). The honeybee search algorithm for three-dimensional reconstruction. In *Workshops on Applications of Evolutionary Computation*, pages 427–437. Springer.

Olague, G. and Puente, C. (2006b). Parisian evolution with honeybees for three-dimensional reconstruction. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 191–198.

Orr, M. S., Che, S., Yilmazer, A., Beckmann, B. M., Hill, M. D., and Wood, D. A. (2015). Synchronization using remote-scope promotion. *ACM SIGARCH Computer Architecture News*, 43(1):73–86.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.

Özgül, M., Deeg, F., and Sattler, S. M. (2017). Mealy-to-Moore transformation. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic*

Circuits & Systems (DDECS), pages 22–27. IEEE.

Palermo, G., Silvano, C., and Zaccaria, V. (2008). Discrete particle swarm optimization for multi-objective design space exploration. In *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 641–644. IEEE.

Patterson, D. A. and Hennessy, J. L. (2016). *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

Peng, Y., Li, B., Liu, D., and Peng, X. (2009). A high speed DMA transaction method for PCI express devices. In *2009 IEEE Circuits and Systems International Conference on Testing and Diagnosis*, pages 1–4. IEEE.

Perez-Cham, O. E., Puente, C., Soubervielle-Montalvo, C., Olague, G., Aguirre-Salado, C. A., and Nuñez-Varela, A. S. (2020). Parallelization of the honeybee search algorithm for object tracking. *Applied Sciences*, 10(6):2122.

Perez-Cham, O. E., Puente, C., Soubervielle-Montalvo, C., Olague, G., Castillo-Barrera, F.-E., Nunez-Varela, J., and Limon-Romero, J. (2021). Automata design for honeybee search algorithm and its applications to 3D scene reconstruction and video tracking. *Swarm and Evolutionary Computation*, 61:100817.

Perez-Cham, O. E., Soubervielle-Montalvo, C., Nunez-Varela, A. S., Puente, C., and Ontanon-Garcia, L. J. (2018). Source code metrics to predict the properties of FPGA/VHDL-based synthesized products. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 93–98. IEEE.

Prongnuch, S. and Wiangtong, T. (2016). Heterogeneous computing platform for data processing. In *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 1–4. IEEE.

Quijano, N. and Passino, K. M. (2010). Honey bee social foraging algorithms for resource allocation: Theory and application. *Engineering Applications of Artificial Intelligence*, 23(6):845–861.

Qureshi, M. A. and Munir, A. (2020). PUF-IPA: A PUF-based identity preserving protocol for internet of things authentication. In *2020 IEEE 17th annual consumer communications & networking conference (CCNC)*, pages 1–7. IEEE.

Rabiai, M., Senouci, M. R., Senouci, A., Busawon, K., and Laurent, D. (2020). A hardware solution to overcome the bandwidth limitation of drone jamming platforms. In *2020 12th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 1–4. IEEE.

Riesgo, T., Torroja, Y., and De la Torre, E. (1999). Design methodologies based on hardware description languages. *IEEE Transactions on Industrial electronics*, 46(1):3–12.

Rodriguez, A. and Moreno, F. (2015). Evolutionary computing and particle filtering: a hardware-based motion estimation system. *IEEE Transactions on Computers*, 64(11):3140–3152.

Sadri, M., Weis, C., Wehn, N., and Benini, L. (2013). Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *Proceedings of the 10th FPGAworld Conference*, pages 1–8.

Sangiovanni-Vincentelli, A., Zeng, H., Di Natale, M., and Marwedel, P. (2015). *Embedded systems development*. Springer.

Scharstein, D., Hirschmüller, H., Kitajima, Y., Krathwohl, G., Nešić, N., Wang, X., and Westling, P. (2014). High-resolution stereo datasets with subpixel-accurate ground truth. In *German conference on pattern recognition*, pages 31–42. Springer.

Schaumont, P. R. (2012). *A practical introduction to hardware/software codesign*. Springer Science & Business Media.

Sklyarov, V. (2000). Synthesis and implementation of RAM-based finite state machines in FPGAs. In *International Workshop on Field Programmable Logic and Applications*, pages 718–727. Springer.

Smeulders, A. W., Chu, D. M., Cucchiara, R., Calderara, S., Dehghan, A., and Shah, M. (2013). Visual tracking: An experimental survey. *IEEE transactions on pattern analysis and machine intelligence*, 36(7):1442–1468.

Smith, D. J. (1998). *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone publications.

Sobel, I. (1990). An isotropic 3×3 image gradient operater. *Machine vision for three-dimensional scenes*, pages 376–379.

Spanjersberg, H. (2012). Micro programming. In *Reflections on the History of Computing*, pages 62–68. Springer.

Szeliski, R. (2010). *Computer vision: algorithms and applications*. Springer Science & Business Media.

Tan, Y. and Ding, K. (2015). A survey on GPU-based implementation of swarm intelligence algorithms. *IEEE transactions on cybernetics*, 46(9):2028–2041.

Tomassini, M. (1996). Evolutionary algorithms. In *Towards Evolvable Hardware*, pages 19–47. Springer.

Toumi, L., Ugur, A., and Azzi, Y. (2019). GPU-based PSO for bitmap join indexes selection problem in data warehouses. In *2019 International Conference on Advanced Electrical Engineering (ICAEE)*, pages 1–6. IEEE.

Tsai, C.-C., Huang, H.-C., and Lin, S.-C. (2010). FPGA-based parallel DNA algorithm for optimal configurations of an omnidirectional mobile service robot performing fire extinguishment. *IEEE Transactions on Industrial Electronics*, 58(3):1016–1026.

Vahid, F. (2010). *Digital design with RTL design, VHDL, and Verilog*. John Wiley & Sons.

Volder, J. (1959). The CORDIC computing technique. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 257–261.

Wolf, W. (2003). A decade of hardware/software codesign. *Computer*, 36(4):38–43.

Wu, Y., Lim, J., and Yang, M.-H. (2013). Online object tracking: A benchmark. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xiao, Y., Pun, C.-M., and Liu, B. (2020). Adversarial example generation with adaptive gradient search for single and ensemble deep neural network. *Information Sciences*, 528:147–167.

Xilinx (2018). *Zynq-7000 SoC Data Sheet: Overview*. v1.11.1.

Xilinx (2020). *7 Series FPGAs Data Sheet: Overview*. v2.6.1.

Xilinx (2021). *Divider Generator*. v5.1.

Yang, X.-S. (2012). Flower pollination algorithm for global optimization. In *International conference on unconventional computing and natural computation*, pages 240–249. Springer.

Zhang, D. and Zheng, Z. (2020). Joint representation learning with deep quadruplet network for real-time visual tracking. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.