POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Evaluating FPGA-Based Convex Optimization Methods for Onboard Low-Thrust Trajectory Guidance

Tesi di Laurea Magistrale in
Space Engineering - Ingegneria Spaziale

Author: **Gonçalo Oliveira Pinho**

Student ID: 10829623
Advisor: Dr. Alessandro Morselli
Co-advisors: Davide Perico, Gianfranco Di Domenico
Academic Year: 2022-23

Abstract

The present work is focused on deploying in an FPGA an onboard guidance algorithm that employs convex approaches to minimum fuel space trajectory problems. An Earth to Mars case with 100 nodes is considered using the interior point solver ECOS. The Zedboard is the selected device.

First, an algorithm is developed to solve the problem. Next, a profile sequence is carried out, from which two functions are elected to integrate the hardware. One is responsible for a numeric factorization of a sparse matrix. The other performs matrix-vector multiplications. These are chosen due to their impact on the application execution time.

The original problem is reduced as a consequence of the limited memory resources of the board. Therefore, depending on the workspace of variables the onboard guidance algorithm presents, a device of appropriate resources shall be selected.

The results from the hardware implementation exhibit a decrease in performance efficiency when compared to its software application. In fact, algorithms developed for software-oriented systems might display nondeterministic and data-dependent behaviors that are incompatible with FPGA optimization techniques. These, contrarily, require deterministic and data-independent cycles to take advantage of the parallelism features of FPGAs. In the ECOS case, significant changes must be made to the functions' structure to ensure compatibility with the concurrent procedures. Moreover, to fully exploit hardware performance, a workaround is to deploy fitting sections of the algorithm that enclose multiple successive operations with high execution time, instead of their separate implementation. This mitigates the additional cycles for the transfer and address of data in the FPGA.

This thesis also highlights the need to apply single-floating point precision in the hardware designs of interior point methods. This is a consequence of the vast dynamic range of numbers these schemes comprehend. Finally, when correctly assembled, the performance gap between the original software application and the FPGA implementation should scale with the problem size.

Keywords: FPGA, Onboard Guidance, Convex Optimization, ECOS, Hardware

Sommario

Il presente lavoro si concentra sull'implementazione in un FPGA di un algoritmo di guida a bordo che utilizza approcci convessi ai problemi di traiettorie nello spazio a minimo consumo di carburante. Viene considerato un caso Terra-Marte con 100 nodi utilizzando il solutore de punto interno ECOS. Il dispositivo selezionato è la Zedboard. A seguito dello svilupo de un algoritmo per risolvere il problema, è stata eseguita una sequenza di scrematura, dalla quale sono stati selezionati due funzioni da integrare nell'hardware. Una è responsabile della fattorizzazione di una matrice sparsa, l'altra esegue moltiplicazioni matrice-vettore, entrambe scelte a causa del loro impatto sul tempo di esecuzione. Il problema originale è stato ridotto a causa delle risorse limitate di memoria della scheda. Pertanto, a seconda dello spazio di lavoro delle variabili presentato dall'algoritmo di guida a bordo, viene selezionato un dispositivo con risorse adeguate.

I risultati dall'implementazione hardware mostrano una diminuzione dell'efficienza delle prestazioni rispetto alla sua applicazione software. Infatti, gli algoritmi sviluppati per sistemi orientati al software potrebbero mostrare comportamenti non deterministici e dipendenti dai dati, che sono incompatibili con le tecniche di ottimizzazione FPGA. Queste richiedono cicli deterministici e indipendenti dai dati per sfruttare la parallelizzazione degli FPGA. Nel caso di ECOS, è necessario apportare modifiche significative alla struttura delle funzioni per garantire la compatibilità con le procedure concorrenti. Inoltre, per sfruttare appieno le prestazioni dell'hardware, una soluzione alternativa sarebbe implementare sezioni dell'algoritmo che includono molteplici operazioni successive con un alto tempo di esecuzione, invece di implementarle separatamente. Questo mitiga i cicli aggiuntivi per il trasferimento e l'indirizzamento dei dati nell'FPGA.

Questa tesi sottolinea anche la necessità di applicare la precisione dei numeri a virgola mobile singola nei progetti hardware dei metodi del punto interno. Questo è una conseguenza della vasta gamma dinamica di numeri compresa in questi schemi. Infine, quando correttamente assemblata, la differenza di prestazioni tra l'applicazione software originale e l'implementazione FPGA dovrebbe essere proporzionale alla dimensione del problema.

Parole chiave: FPGA, Guida a Bordo, Ottimizzazione Convessa, ECOS, Hardware

Acknowledgements

I want to thank my advisor Dr. Morselli for all the time, willingness, attention and help devoted throughout the entire thesis. I also would like to thank both of my co-advisors, Davide and Gianfranco, for all the guidance and corrections that made this work possible. A special gratitude goes to Alessandra and Andrea who were also present throughout the entire process. Particularly, I would like to thank Professor Topputo for allowing me to work with his research group and for his precious insights and kindness during our meetings over the past few months. Thanks to all the aforementioned people, I managed to deepen my knowledge on such a diverse topic from what I was used to. In the end, this experience made me realize how much I still have to learn.

Afterward, I would like to thank my parents José and Maria who never doubted me, or my capacities and were always there by my side. Their simpler look at life helped me in difficult times and made me a better human being. Without them and all their sacrifices, none of this or who I am today would be possible. Then, I would also like to give a special thanks to my brother Tomás for always being there for me. I will always remember our conversations about sports, films, or music over the phone in these last months. I've been learning so much from him and his different viewpoints on life.

A heartfelt thanks goes to Giorgia for all the support, time, and love given over this past year. Always by my side, her caring and warm words have been a source of strength and faith throughout this journey. Thanks for all the stupid, but in a way, beautiful moments we have shared, allowing me to grow so much as a human with someone like her. Hopefully, this is just the beginning chapter of "Our Adventure Book".

Thanks to all the friends that Italy has given me and that have become a pillar of my life. Thanks for all the meals, nights, laughs, trips and, not to forget, debates over non-sense topics which I will delightedly carry with me. Lastly, an enormous thanks goes to all my friends throughout Portugal who always keep care and contact with me. Thanks for the constant support and the memorable experiences from which I've gathered so much. You all have made this an unforgettable one-of-a-kind university and life experience.

Contents

1 | Introduction

Over the past few years, the development of the space market has been towards the miniaturization of satellites and their components, leading to a higher number of launches as a benefit from the reduced costs [1]. Enhancing the level of autonomy and transferring flight-related tasks, such as the guidance design onboard, to automated systems, is a beneficial objective for future missions [2].

The system design of deep-space missions requires thrust actions to perform essential operations such as orbit insertion, correction, maneuvers for trajectory adjustments, and others. For small satellites, such as CubeSats, low-thrust technology is the typical propulsive subsystem of choice due to the high fuel efficiency it presents. However, the design of a low-thrust optimal, or near-optimal, trajectory reveals itself a convoluted task when paired with the need for fuel savings. These require high accuracy and computational efficiency.

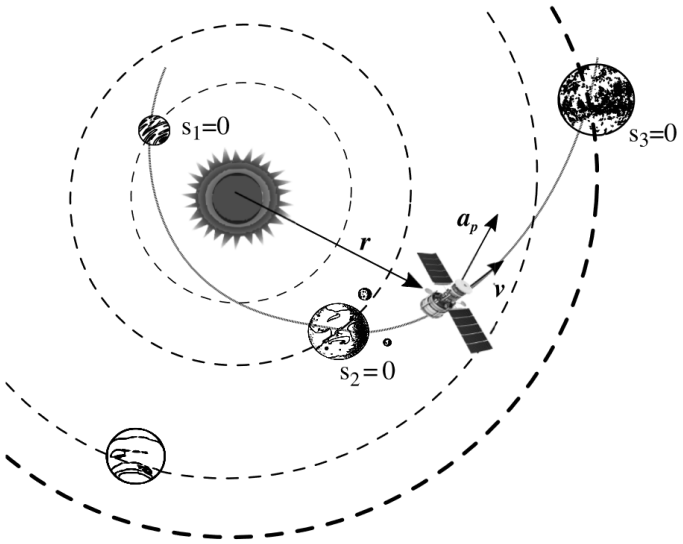


Figure 1.1: Illustrative interplanetary transfer [3].

In fact, when designing solution schemes of optimal control problems, three aspects are taken into account: computational effort, reliability (capability of converging even when

a poor initial guess is provided) and optimality (minimization of a defined objective function) [4]. For the case of autonomous satellites that utilize onboard guidance, the maximization of the available computational power is of extreme importance while preserving the optimality of the problem. In fact, OnBoard Computer (OBC) systems have several key resource constraints compared to ground computers. Specifically, low power consumption, reduced memory capacity, resilience to space environment hazards, long-term reliability, and low volume and weight are some of the vital limitations to consider.

Nowadays, algorithms developed for onboard applications take into consideration the limited number of resources available. Accordingly, to provide higher onboard computational power, many research works have proposed dedicated hardware accelerators through Field Programmable Gate Array (FPGA) implementations [5–8]. This approach combines software and hardware dedicated implementations to optimize the onboard resources.

FPGA has been a topic of interest in the space market in recent years due to the advantages the technology presents [9, 10]. These programmable devices can be specifically designed to integrate different functions and requirements an application imposes. The capacity for onboard reconfiguration, even after deployment, presents a valuable asset for mission objectives that might require in-flight adaptability [9]. With satellite lifetimes expanding far beyond ten years, this has become a stringent requirement in recent times [11]. Then, by limiting the function logic components to the necessary minimum, the implemented hardware design presents minimum power consumption when compared to other fixed-hardware solutions that may include unused components [12]. Afterward, FPGAs are parallel devices that exploit the simultaneous execution of multiple tasks. This concurrency of actions provides higher computational throughput, enabling faster calculations while keeping the required level of accuracy [12]. To ensure reliable operations and performance, radiation-hardened FPGAs, a subset of these devices, are precisely planned to mitigate the harsh space environment effects [13]. Finally, due to the compact form factor, they are suited equipment for space missions where size and weight are two crucial constraints. The ability to support the integration of multiple dedicated functions into a single device corroborates for the needed space efficiency [12].

Table 1.1 highlights the presence of some FPGA devices in state of the art market for OBC and their Commercial-Off-The-Shelf (COTS) availability. The capacity of mitigating Single Event Upsets (SEU) is also shown, an undesired behavior, further discussed in Chapter 3, derived from the interaction between the device and the harsh space environment.

Manufacturer	Product	Processor	Pedigree	SEU mitigation	Vehicle
Innoflight	CFC-300	AMD-Xilinx Zynq-7020 Dual-core ARM Cortex-A9	COTS	No	CubeSat
Xiphos	Q8S	AMD-Xilinx Zynq Ultrascale+ MPSOC Quad-core ARM Cortex-A53	COTS	No	Nano, Micro, and Small Satellites
KP Labs	Antelope onboard computer	DPU – AMD Xilinx Zynq UltraScale+ MPSoC, Quad ARM Cortex-A53 CPU, Dual ARM Cortex-R5	COTS	Yes	CubeSat
GomSpace	Nanomind A3200	Atmel AT32UC3C MCU	COTS	Yes	CubeSat
Unibap	iX10-100	Microchip PolarFire FPGA with RISC-V, AMD V1605b (Ryzen) CPU and GPU	COTS	Yes	Nano, Micro, and Small Satellites

Table 1.1: Sample of highly integrated onboard computing systems [1].

That said, the use of FPGAs for application acceleration purposes is not trivial. Indeed, algorithm-level optimizations done in software implementations can, in some cases, interfere with the hardware requirements. This bottleneck can impact the desired concurrency of tasks. A regular and exemplary concern, in onboard applications, are arithmetic operations such as matrix multiplication or Cholesky factorization. When oriented for software algorithms, software optimization techniques, such as storing formats, are employed to achieve higher levels of computational efficiency. However, these often result in irregular tasks or memory access patterns that interfere with the exigencies of FPGA optimization schemes. Hardware-optimization processes involve a set of requirements to exploit the application throughput maximization the technology can offer.

1.1. Thesis Objective

This thesis aims to investigate the different aspects to consider when deploying an autonomous guidance algorithm based on convex optimization on a reconfigurable computing hardware. To achieve such goal, the main research question this study seeks to resolve is:

What are the key challenges and practices in deploying onboard trajectory optimization guidance algorithms on FPGA platforms?

As a case problem, an Earth to Mars minimum-fuel trajectory optimization is considered. The choice of Mars comes from the renewed interest its exploration has been gaining in recent times [14]. Specifically, an algorithm based on Sequential Convex Programming is exploited to solve the problem. Afterward, the deployment of some of its core parts on an FPGA is targeted.

[1] `https://www.nasa.gov/wp-content/uploads/2023/05/2022-soa-full.pdf` (last accessed: November 1, 2023)

Moreover, this work explores the complexity of designing optimized hardware implementations through the use of high-level code languages such as C. This enables the user to focus on optimization characteristics without deepening into lower-level code.

The proof for performance increase granted by the use of the technology in this field is out of the scope of the thesis. Instead, the knowledge aroused from this study aims to serve as an initial reference for future applications which, undoubtedly, will assist in the procedures for maximizing computational efficiency of onboard guidance algorithms.

1.2. Thesis Structure

This work is organized as follows:

Chapter 2 presents an overview of the convex low-thrust trajectory optimization problem under study and the convex-approach algorithm used to solve it. A brief introduction to the models, assumptions, and mathematical passages used to transform the original non-convex problem into a feasible convex state is done. The chapter ends with a concise scheme of the algorithm to deploy on the board.

Chapter 3 introduces the necessary background information about FPGA. Specifically, the Zynq architecture and functionalities are discussed, going over the different software tools used for hardware design.

Chapter 4 explores, in great detail, the sections of the algorithm selected for deployment in the FPGA. Reasoning behind the selection of functions and interfaces is also presented. The optimization techniques and constraints encountered throughout the development phase are reported.

Chapter 5 shows the results obtained from the deployment of multiple hardware designs. After an in-depth examination and discussion of the workflow and methods used, a set of requirements and valuable insights into the development of future guidance algorithms in FPGA applications is provided.

Chapter 6 concludes the work, including some references for future works in this field.

2 | Background

An optimization problem has to be solved to compute the best achievable trajectory among those feasible. Currently, the approaches taken to solve this optimal control problem can be divided into two main categories: indirect and direct methods [15].

Indirect methods use calculus variations and Pontryagin's minimum principle to attain the desired solution. To achieve it, lengthy mathematical derivations and good initial guesses are required, making this method a not-so-valid option for onboard applications and with the state-of-the-art techniques available [16]. Instead, direct methods do not require the explicit derivation of optimality conditions. The trajectory is discretized into multiple segments such that the continuous-time optimal control problem is converted into a finite-parameter optimization problem, generally solved through Nonlinear Programming (NLP) methods. A challenge occurs for trajectories characterized by low-thrust levels where changes to the orbits may result in long-duration trajectories. In addition, large nonlinear programs with high computational complexities most likely lead to calculations of low accuracy [17]. Indeed, for highly nonlinear problems, the algorithm's convergence can not be guaranteed, making this method a poor choice from a robustness point of view [18].

As a result, in recent times, to solve this type of nonlinear optimal control problems characterized by low-thrust trajectories, a subfield of optimization which targets the minimization of convex functions over convex sets called convex optimization has seen extended developments. Convex problems, due to their low complexity, feature the capability of being solved employing polynomial-time algorithms, resulting in lower computational requirements when compared to the classical methods described above and as explained in [19]. For this approach, the original nonconvex problem has to be transformed into a convex state through a set of techniques explained in Section 2.2.

This said, the present chapter states a low-thrust trajectory optimization problem and the convex-approach algorithm used to solve it as in [20]. The goal is to have an overall view of the problem, unveiling the models, hypotheses, and assumptions used. The main steps and processes to transform the original problem into a solvable and feasible state are presented. Lastly, the essence of the algorithm used is detailed.

2.1. Astrodynamic models

Astrodynamic or orbital models are mathematical models used to study the motion of an object in space. After defining the model to use and the initial state vector of the object, it is possible to propagate its trajectory through time and space by sequentially iterating the set of equations. Depending on the selected model, the object will be subject to different types of perturbations and forces.

2.1.1. Problem Formulation

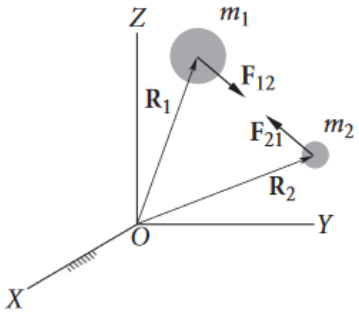


Figure 2.1: Two-body problem [21].

The present work focuses on solving a Space Trajectory Optimization (STO) problem with minimal fuel consumption. In other words, the objective is to determine the ideal path between the specified points which minimizes the propellant expenditure, keeping in mind the need for high computational efficiency and accuracy of the results.

To start, the two-body problem scenario is the chosen dynamical model. It is a classical problem, in orbital mechanics, around the interaction between two bodies due solely to their mutual gravitational attraction. One body has a much larger mass when compared to the second and, as a result, the motion of the first body is not affected by the second one (an example could be a planet and a spacecraft, respectively). Due to the significant masses and interactions between one another, the influence of other celestial bodies is neglected [21].

Considering two-body dynamics, the equations of motion, in spherical coordinates, for the continuous-time optimal control problem, after a simple normalization method is applied, are presented in equation (2.1). Figure 2.2 illustrates the coordinate system.

$$
\begin{aligned}
\dot{r} &= v_r, \\
\dot{\theta} &= \frac{v_\theta}{r\cos\phi}, \\
\dot{\phi} &= \frac{v_\phi}{r}, \\
\dot{v}_r &= \frac{v_\theta^2}{r} + \frac{v_\phi^2}{r} - \frac{1}{r^2} + \frac{cT\cos\alpha_r}{m}, \\
\dot{v}_\theta &= -\frac{v_r v_\theta}{r} + \frac{v_\theta v_\phi \tan\phi}{r} + \frac{cT\sin\alpha_r \sin\alpha_{\phi\theta}}{m}, \\
\dot{v}_\phi &= -\frac{v_r v_\phi}{r} - \frac{v_\theta^2 \tan\phi}{r} + \frac{cT\sin\alpha_r \cos\alpha_{\phi\theta}}{m}, \\
\dot{m} &= -\frac{cT}{v_e},
\end{aligned}
\tag{2.1}
$$

where r is the radial distance from the central body to the spacecraft, θ the azimuth angle in the xy-plane measured from the x-axis and ϕ the elevation angle measured from the xy-plane. Taking into account the spherical coordinate system, v_r, v_θ and v_ϕ are, respectively, the components of the velocity along the three axes. The mass of the spacecraft is represented by m. As for the control variables, the magnitude of the thrust is denoted by T while α_r and $\alpha_{\phi\theta}$ are the thrust direction angles. The angle $\alpha_r \in [0, \pi]$ is defined between the thrust vector and the radial direction $\mathbf{e}_r$ whilst $\alpha_{\phi\theta} \in [0, 2\pi]$ is the angle in the $\mathbf{e}_\phi\mathbf{e}_\theta$-plane measured concerning the $\mathbf{e}_\phi$-axis. Finally, $c = T_{\max}R_0/(m_0V_0^2)$, where $T_{\max}$ is the maximum thrust produced by the engines, $R_0 = 1$ Astronomical Unit (AU) and $V_0 = \sqrt{\mu/R_0}$ (μ being the gravitational constant of the Sun) are distance and velocity units, respectively, used to normalized the problem, m_0 is the initial mass of the spacecraft and v_e is the exhaust velocity.

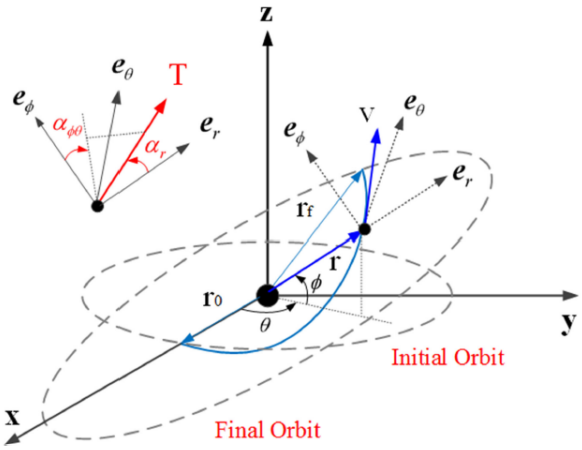


Figure 2.2: Three-dimensional spherical coordinates system [4].

The set of equations defined in (2.1) can be represented in a state space formulation as highlighted in equation (2.2).

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \tag{2.2}$$

where $\mathbf{x} = [r; \theta; \phi; v_r; v_\theta; v_\phi; m]$ is the state vector of the spacecraft and $\mathbf{u} = [T; \alpha_r; \alpha_{\phi\theta}]$ is the control vector.

By defining a set of initial and final state conditions, control constraints (all before mentioned present in [4]) and taking into account the dynamics in (2.1), a general nonlinear optimal control problem is formulated. The goal is to find the optimal control vector profile that minimizes the propellant expenditure. In reality, it is the same as maximizing the final vehicle mass $J = -m(t_f)$ while subject to the dynamics, constraints and conditions stated.

However, the dynamics presented in (2.1) are highly nonlinear with, in addition, the state and control variables also being coupled. This poses a nonlinear and nonconvex problem which brings difficulty to the convergence of NLP algorithms.

Therefore, a series of relaxations have to be applied to the original minimum-fuel STO problem to transform it into a convex one as in [4, 20]. These steps are presented in the upcoming section.

2.2. Problem Transformation and Convexification

To reach a convex optimal control problem, three successive processes are done: change of variables, relaxation of control constraints and linearization of dynamics. This section will mainly show the results obtained from the mentioned processes and key aspects to consider, while the full details and passages can be found in [4, 20].

To decouple the state and control variables, a change of variables is first introduced to replace the mass m and thrust T in equation (2.1), defining $z = \ln m$ and $\tau = T/m$, where τ_r, τ_θ and τ_ϕ are accordingly defined as:

$$\begin{aligned} \tau_r &= \tau \cos \alpha_r, \\ \tau_\theta &= \tau \sin \alpha_r \sin \alpha_{\phi\theta}, \\ \tau_\phi &= \tau \sin \alpha_r \cos \alpha_{\phi\theta}. \end{aligned} \tag{2.3}$$

This step requires the introduction of the constraint

$$\tau_r^2 + \tau_\theta^2 + \tau_\phi^2 = \tau^2, \tag{2.4}$$

which is nonconvex.

For this reason, a relaxation of the control constraint to a convex one is done by expanding its feasible set, seen in equation (2.5).

$$\tau_r^2 + \tau_\theta^2 + \tau_\phi^2 \leq \tau^2. \tag{2.5}$$

The proof that the optimal control of the new defined problem lies on the boundary of the original constraint surface defined by equation (2.4) is shown in [4, 14].

Lastly, the nonlinear dynamics are convexified through a successive small-disturbance-based linearization method. The nonlinear term is linearly approximated with respect to a certain state history $\mathbf{x}^*(t)$. This linearization results in equation (2.6b), where the subscript $(\cdot)^*$ denotes the reference trajectory. Alongside the linearization of dynamics, a trust-region constraint is imposed with equation (2.6g), where R is the radius of the trust region. This constraint keeps the solution close to the reference, and consequently, assures that the linearization is valid [20].

Therefore, the original minimum-fuel STO nonconvex problem is transformed into the convex form presented in equation (2.6), as formulated in [20].

$$\begin{aligned}
\text{minimise} \quad & J_0 = \int_{t_0}^{t_f} \tau(t)\, dt, && \text{(2.6a)}\\
\text{s. t.} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}^*) + \mathbf{A}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + \mathbf{B}\mathbf{u}, && \text{(2.6b)}\\
& \tau_x^2 + \tau_y^2 + \tau_z^2 \leq \tau^2, && \text{(2.6c)}\\
& 0 \leq \tau \leq T_{\max} e^{-z^*}[1 - (z - z^*)], && \text{(2.6d)}\\
& \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u, && \text{(2.6e)}\\
& \mathbf{u}_l \leq \mathbf{u} \leq \mathbf{u}_u, && \text{(2.6f)}\\
& \|\mathbf{x} - \mathbf{x}^*\|_1 \leq R, && \text{(2.6g)}\\
& \mathbf{x}(t_0) = [r(t_0); \theta(t_0); \phi(t_0); v_r(t_0); v_\theta(t_0); v_\phi(t_0); z(t_0)], && \text{(2.6h)}\\
& \mathbf{x}(t_f) = [r(t_f); \theta(t_f); \phi(t_f); v_r(t_f); v_\theta(t_f); v_\phi(t_f)], && \text{(2.6i)}
\end{aligned}$$

with new state vector $\mathbf{x} = [r; \theta; \phi; v_r; v_\theta; v_\phi; z]$ and new control vector $\mathbf{u} = [\tau_x; \tau_y; \tau_z; \tau]$. In equation (2.6b), $\mathbf{f}$ represents the vector of the natural two-body dynamics, defined as

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} v_r \\ v_\theta/(r\cos\phi) \\ v_\phi/r \\ v_\theta^2/r + v_\phi^2/r - 1/r^2 \\ -v_r v_\theta/r + v_\theta v_\phi \tan\phi/r \\ -v_r v_\phi/r - v_\theta^2 \tan\phi/r \\ 0 \end{bmatrix}, \tag{2.7}$$

while $\mathbf{A} = \partial\mathbf{f}/\partial\mathbf{x}$ is the jacobian matrix, and $\mathbf{B}$ is the control matrix, resultant of the change of variables, where

$$\mathbf{B} = \begin{bmatrix} \mathbf{0}_{3\times4} \\ c\,\mathbf{I}_{3\times3} \quad \mathbf{0}_{3\times1} \\ \mathbf{0}_{1\times3} \quad -c/v_e \end{bmatrix}. \tag{2.8}$$

Equation (2.6d) is a convexified control constraint where the upper bound is the result of a first-order Taylor series expansion (more details in [4]). Both lower $(\cdot)_l$ and upper $(\cdot)_u$ bounds of states and controls are given in equations (2.6e) and (2.6f). Lastly, the initial $\mathbf{x}(t_0)$ and final $\mathbf{x}(t_f)$ conditions are defined in equations (2.6h) and (2.6i). As the objective of the problem is to minimize the fuel, the condition $z(t_f)$ is left free.

All the values related to the problem under study can be found in Section 4.1.

Equation (2.6) represents the convex STO problem, which will be referred to as Convex Problem (CXP) throughout this work.

2.3. Sequential Convex Programming

Directly solving the CXP defined in Section 2.2 does not correspond to solving the original nonconvex minimum-fuel STO problem [20]. Instead, a successive approach that considers a sequence of convex problems is established.

A Sequential Convex Programming (SCP) is a local optimization method where a sequence of convex optimal control sub-problems defined by equation (2.6) is formed using the solutions from the previous iteration. However, before exploring the different passages of the SCP, some aspects have to be taken into account.

Each sub-problem, which is an expansion of the CXP defined before, is an infinite-dimensional optimal control problem. To find its numerical solution, an arbitrary-order

Gauss-Lobatto discretization scheme is used, exploiting a nonlinear interpolation of the control variables (for more details, check [20, 22]). This discretization scheme is one among many available, as discussed in [23].

This said, when nonlinear constraints are linearized about a reference solution, one may encounter infeasible convex sub-problems even though the original problem is feasible. This phenomenon is called *artificial infeasibility* [17]. To avoid it, the unconstrained variables $\boldsymbol{\nu}$ and $\eta \geq 0$ are introduced in equations (2.6b) and (2.6d), resulting in

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}^*) + \mathbf{A}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + \mathbf{B}\mathbf{u} + \boldsymbol{\nu}, \tag{2.9}$$

$$0 \leq \tau \leq T_{\max} e^{-z^*}[1 - (z - z^*)] + \eta. \tag{2.10}$$

Although these terms maintain feasibility, they also result in constraint violations which, when active, must be zero at the end of the optimization process so that the constrains are satisfied. Therefore, we incorporate them in the objective function J_0 in equation (2.6a) with sufficiently large penalty parameters μ_i and λ_i

$$J = J_0 + \sum_{i \in I_{\text{eq}}} \mu_i \left\|\boldsymbol{\nu}_i\right\|_1 + \sum_{i \in I_{\text{ineq}}} \lambda_i \max(0, \eta_i), \tag{2.11}$$

where I_{eq} and I_{ineq} denote the set of equality and inequality constrains, respectively.

All these changes are translated into a new version of the minimum-fuel STO problem defined in equation (2.12).

$$\begin{aligned}
\text{minimise} \quad & J = J_0 + \sum_{i \in I_{\text{eq}}} \mu_i \left\|\boldsymbol{\nu}_i\right\|_1 + \sum_{i \in I_{\text{ineq}}} \lambda_i \max(0, \eta_i), && \text{(2.12a)} \\
\text{s. t.} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}^*) + \mathbf{A}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + \mathbf{B}\mathbf{u} + \boldsymbol{\nu}, && \text{(2.12b)} \\
& \tau_x^2 + \tau_y^2 + \tau_z^2 \leq \tau^2, && \text{(2.12c)} \\
& 0 \leq \tau \leq T_{\max} e^{-z^*}[1 - (z - z^*)] + \eta, && \text{(2.12d)} \\
& \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u, && \text{(2.12e)} \\
& \mathbf{u}_l \leq \mathbf{u} \leq \mathbf{u}_u, && \text{(2.12f)} \\
& \left\|\mathbf{x} - \mathbf{x}^*\right\|_1 \leq R, && \text{(2.12g)} \\
& \mathbf{x}(t_0) = [r(t_0); \theta(t_0); \phi(t_0); v_r(t_0); v_\theta(t_0); v_\phi(t_0); z(t_0)], && \text{(2.12h)} \\
& \mathbf{x}(t_f) = [r(t_f); \theta(t_f); \phi(t_f); v_r(t_f); v_\theta(t_f); v_\phi(t_f)]. && \text{(2.12i)}
\end{aligned}$$

At this point, the SCP method can be applied, with the overall flow, illustrated in Figure 2.3, being:

1. Set k = 0. Using the initial and final state conditions defined in equations (2.6h) and (2.6i), setting the time span and number of nodes of the problem, an initial reference trajectory is generated;

2. For k $\geq$ 1, solve the optimal control problem defined in equation (2.12) using the (k-1) trajectory as the reference one and adding a convergence technique imposed through equation (2.13), where γ is a cauchy parameter selected based on the desired level of convergence, further explained in [4, 24];

$$\left\|\mathbf{x}-\mathbf{x}^{(k-1)}\right\|_1 \leq \gamma \left\|\mathbf{x}^{(k-1)}-\mathbf{x}^{(k-2)}\right\|_1 \ , \ \gamma \in (0,1) \text{ and } k>1. \tag{2.13}$$

3. Check the convergence condition (2.14) where ε is a tolerance value for convergence set at the beginning. If this condition is satisfied and the feasibility terms are also zero or under a user-selected threshold, the solution has been found and it is stored. Otherwise, retrieve to step 2, updating all the respective values and functions with the new reference trajectory.

$$\sup_{t_0 \leq t \leq t_f} \left\|\mathbf{x}-\mathbf{x}^{(k-1)}\right\|_1 \leq \varepsilon, \ k>1. \tag{2.14}$$

Notice that, for each $k \geq 1$, the nonlinear optimal control sub-problem defined by equation (2.12) is characterized only by linear time-varying dynamics, affine equality constraints and second-order inequality constraints. For these reasons, each sub-problem can be discretized into a Second-Order Cone Programming (SOCP) problem [4] which can in turn be solved by an Interior Point Method (IPM). In this work, the embedded conic solver (ECOS), an interior-point solver for SOCP, is used to solve each iteration [25].

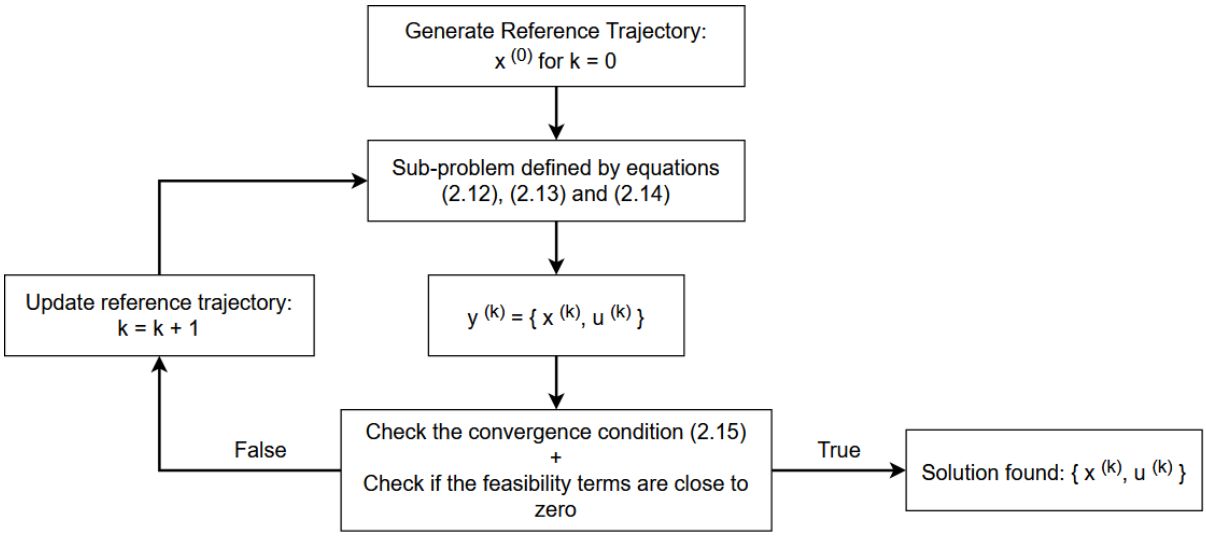


Figure 2.3: SCP method.

Additionally, the parameterized convex optimization problem shown in equation (2.12) can be written in a discretized form as

$$\begin{aligned} \text{minimise} \quad & \mathbf{c}^T \mathbf{x} \\ \text{s. t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{G}\mathbf{x} \leq \mathbf{h}, \end{aligned} \tag{2.15}$$

where $\mathbf{x} \in \mathbf{R^n}$ is the optimization variable, $\mathbf{A}_{n \times n}$, $\mathbf{G}_{n \times n}$, $\mathbf{b} \in \mathbf{R^n}$ and $\mathbf{h} \in \mathbf{R^n}$ are, respectively, equality and inequality constraints matrices and vectors given by the problem.

Writing the convex optimization problem defined in equation (2.12) in the mathematical problem (2.15), such that is compliant with the restrictive standard form of IPM solvers, such as ECOS, would be an arduous and time consuming task. Instead, CVXPY, an open source Python-embedded modeling language for convex optimization problems, is used [26]. This tool enables the problem to be expressed in a user friendly and natural form, such as the one shown in equation (2.12), which is subsequently mapped to the mathematical problem (2.15). This removes the need and time needed to formulate the problem accordingly with the presented ECOS syntax [26]. However, by being written in Python, the generated code would not be compatible with the coding language of the device used in this work. For this reason, another tool called CVXPYgen, which enables the generation of custom C code for a parameterized class of convex optimization problems, is used [27]. This tool uses the Python formulated problem to generate a respective C code version. In Section 2.3, the Pseudo-algorithm 4.1 constructed around

the code attained from CVXPYgen is presented.

Finally, an important remark about a section of the application is made. The C code version of ECOS essentially works with three main functions:

- `Setup`: responsible for allocating memory for ECOS and necessary initialization processes before the solver can start;
- `Solve`: the core interior-pointer solver;
- `Cleanup`: frees memory that has been allocated in the `Setup` stage.

Both the `Setup` and `Cleanup` phases use dynamic memory processes such as `malloc` and `free` [25]. These types of system calls which manage memory allocation are not supported with FPGA implementations. In fact, to synthesize a hardware implementation, the design must be fully self-contained through the use of static memory allocation schemes [28]. Consequently, any function linked with either of these phases of ECOS can not be implemented in the FPGA.

Contrarily, `Solve` only uses static memory allocation, therefore being a perfect fit for this work [25]. In practice, all meaningful candidates to implement in the FPGA will be associated with this step of the algorithm, as described in Section 4.2.1.

3 | FPGA Overview

This chapter aims to give an overall background about FPGAs. In particular, it focus on their architecture and main components, the configuration procedure, potential benefits and current limitations. Moreover, the discussion considers their application in the space domain.

A comprehensive look at Xilinx Zynq devices [1] and their characteristics follows, together with the description of the evaluation board used in this work, the Zedboard [2]. The chapter evolves by presenting the software tools used and the possible framework for IP cores development and deployment. Finally, the typical metrics and optimization techniques developers target in their design process are, likewise, detailed.

3.1. FPGA Definition

An FPGA are integrated circuits intended for custom hardware implementation, being reconfigurable for an infinite number of times. Depending on the specifications and requirements of the application, the user can reconfigure the hardware accordingly to achieve the desired results, even after the manufacturing process. Accordingly, a high flexibility in the implementation is present [29].

To achieve such high versatility and adaptability, FPGAs consist of a web of blocks connected by programmable logic, all of these explained in detail in Section 3.1.2.

With a shorter design time compared to other application-specific environments, the FPGA achieves high-performance computation by exploiting parallel processing [30]. It can be considered as a General Purpose Processor (GPP) combined with an Application-Specific Integrated Circuit (ASIC), each technology being suited for different scenarios depending on the needs and constraints, as seen in Figure 3.1.

[1] `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html` (last accessed: November 10, 2023)

[2] `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/` (last accessed: November 10, 2023)

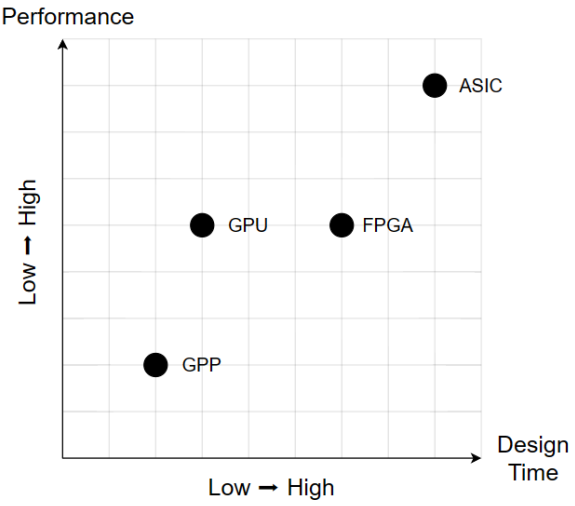


Figure 3.1: Applicable domain of different integrated circuits/processing units.

Due to their programmable nature, FPGAs are a popular choice for different markets and applications such as Aerospace, Defence, Medical, Video and Image Processing, Industrial and others [31].

Within the space sector, FPGAs are commonly used for navigation [32], earth observation [6] or deep-space scenarios [33]. Notably, exemplary cases of past applications are onboard processing such as image processing, filtering algorithms or remote sensing operations. By exploiting the spatial and temporal parallelism that the technology offers, great improvements in the performance of algorithms can be achieved, maximizing the computational power available onboard [34]. In addition, when compared to discrete logic scenarios, FPGAs present reduced weight and volume due to the shortened number of devices required to perform the same operations, increased reliability with reduced solder connections and, last but not least, higher flexibility due to the capability of making design changes even after the board layout is complete [10].

Radiation Protection

When working with these devices in space, special attention has to be given to the possibility of occurrence of Single Event Upsets (SEU), non-destructive errors, such as transient pulses in logic or support circuitry, that might lead to undesired effects in the integrated circuit. These can range from loss or alteration of scientific data to, in extreme scenarios, complete system shutdowns.

Cosmic rays and high-energy protons are two major cause of SEU.

Cosmic rays are high-energy particles, typically characterized by a heavy ion component,

originating from the Sun, outside the Solar System, and other galaxies. They can lead to memory bit flip or transient when the ion particle transverses the device [35]. On the other hand, high-energy protons are high-energy particles usually trapped in Earth's radiation belts or provided from solar flares. They potentially cause direct ionization SEU when impacting the device, leading, once again, to undesired behaviors [36].

Of highlight, an ESA audit made on several FPGA designs included in the Rosetta mission propose countermeasures against two particular undesired SEU, the "bit-flip" and "deadlock state", with more information about the radiation tests and results being available in [9]. Furtherwise, the vulnerability of an FPGA to SEU depends on the type of configuration data storage technology used as Table 3.1 exhibits.

	SRAM	**Flash**	**Antifuse**
Reconfigurable	Yes	Yes	No
Immunity against SEU	Low	Medium to high	Very high

Table 3.1: Configuration data storage technologies and characteristics [13, 37, 38].

The solution to these risks, and fitted for space applications, are space-graded boards characterized by tolerance against the mentioned radiations. Even if at the moment the availability of these types of devices is limited, the tendency is to grow as the space economy expands [39].

Intellectual Property

Lastly, an important term that will be used extensively throughout this thesis is *Intellectual Property* (IP) block or core. These are functional blocks of logic or data designed for specific functionalities, typically integrated in bigger systems or devices. Figure 3.2 depicts a combination of IP cores to form a Video IP core subsystem (the composition of multiple IP cores sets up a subsystem).

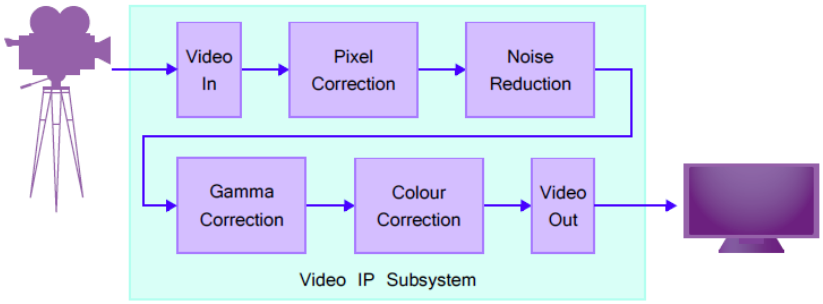


Figure 3.2: Example of an IP subsystem [12].

3.1.1. FPGA Configuration

The configuration of an FPGA is done through Register-Transfer Level (RTL) methods, a representation of notations used to specify the sequence of micro-operations to perform [12]. As RTL code is very low-level, the level of abstraction is raised using higher-level Hardware Description Languages (HDL) such as Verilog or Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) [40]. These programming languages describe the operation and structure of digital electronic circuits. The RTL code is written at an HDL level which, through synthesis tools, gets translated to gate-level description. These are standard cells (like NAND, OR, XOR gates, etc.) that map the functionality specified in RTL. In conclusion, thanks to High-Level Synthesis (HLS) tools, the process configuration of an FPGA, typically done with RTL, is eased. Further details about this workflow and a coding method of a higher level of abstraction are explored in Sections 3.3 and 3.4.1.

Once the design is synthesized, a binary file called Bitstream, which contains the configuration information for a particular FPGA, is generated. It contains all the data to be transferred onto the configuration memory as well as the proper commands to control the chip functionalities. It is responsible for the logic configuration and routing of the FPGA to match the functionality developed at RTL.

To achieve such high versatility and adaptability, FPGAs consist of a web of blocks connected by programmable logic, all of these about to be explained in more detail in the following sections.

3.1.2. The Logic Fabric

An FPGA can be broken down into three main building blocks: Configurable Logic Blocks (CLB), Input or Output Blocks (IOB) and Interconnections. A scheme of these elements and their disposition can be seen in Figure 3.3.

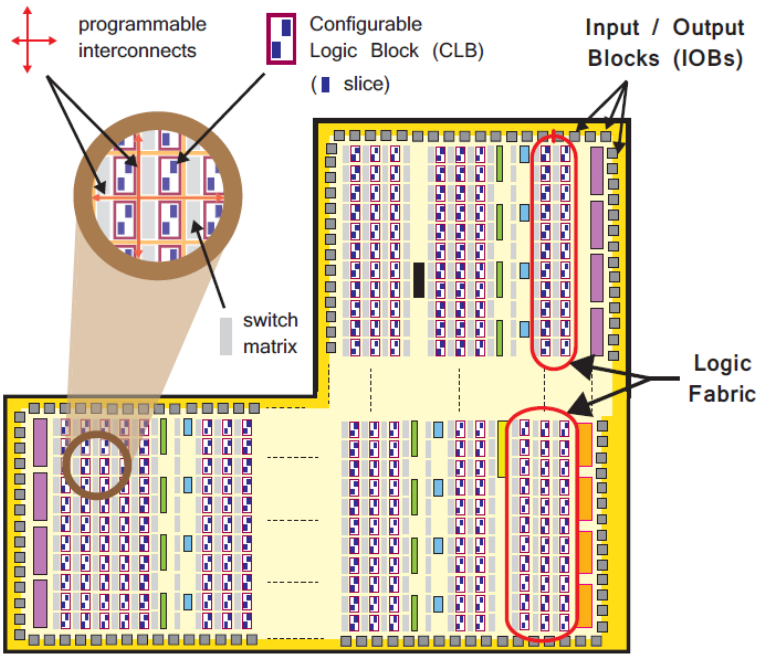


Figure 3.3: The logic fabric and its elements [12].

CLB

CLB is a basic building block used to create logic operations, making up a large part of the FPGA [12]. They are connected to other similar resources through programmable interconnections. Each CLB contains two logic slices and is positioned next to a switch matrix as in Figure 3.4.

A switch matrix provides a flexible routing facility to make connections. This is both between elements within a CLB and between the CLB and other resources of the FPGA.

Logic slices are a sub-unit of a CLB which contains resources to implement combinatorial and sequential logic circuits. They are composed of Lookup Tables (LUT) and Flip-Flops (FF). LUT is a set of memory cells responsible for implementing small logic functions. When combined, they can form larger logic functions, memories, or shift registers as required. FF are simple circuit elements capable of implementing a 1-bit register, with reset functionality.

The characteristics of the blocks and the amount available in each FPGA vary with the device, with the specifications for the board used in this work being presented at Section 3.2.3.

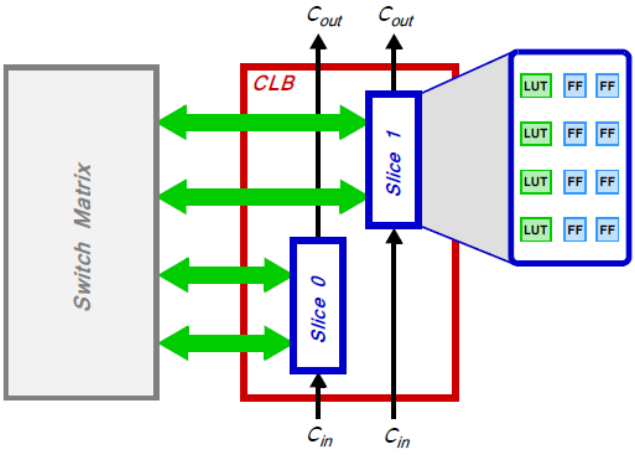


Figure 3.4: Composition of a configurable logic block [12].

IOB

The IOB is a programmable unidirectional or bidirectional interface between the FPGA's internal logic and a package pin used to connect to external circuitry [12]. IOBs are usually located around the perimeter of the device and are used to complete the matching requirements for input or output signals under different electrical characteristics.

Interconnections

Lastly, the aforementioned array of programmable logic blocks are connected through a network of programmable interconnects which, then, serve as the bridge responsible for transferring signals across the device.

3.1.3. DSP and BRAM

In addition to the previously defined elements, modern FPGA architectures include two useful components: Digital Signal Processor (DSP) and Block Random Access Memory (BRAM).

DSP slices are dedicated elements used to improve the throughput and latency of arithmetic operations. This block reduces the amount of CLBs that would be needed to produce similar arithmetic computations, shortening the overall number of resources used. Specifically, DSP48E1s [41], the ones present in the Zedboard, are dedicated silicon resources

used for high-speed arithmetic operations on signals with medium to long word lengths. More details about DSP48E1s can be found in [41].

BRAM is useful for memory requirements such as storing data. In the Zynq-7000 series, discussed later in Section 3.2, these blocks are capable of Random Access Memory (RAM), Read Only Memory (ROM) and First In First Out (FIFO) buffers. Each block can store up to 36Kb of information, with reshape capability for different data formats. The alternative would be to combine multiple LUTs to reach a similar result, this being called Distributed RAM. This approach has two main limitations, them being the higher number of resources it requires and the implementation restriction to the routing delays between the logic used, impacting the timing results. For further details about the Block RAM check [42].

Finally, due to the operations these two resources perform, they often interact with one another. To maximize timing throughput, they are integrated in proximity to each other into the logic array. Figure 3.5 highlights the column arrangement that fulfills this requisite.

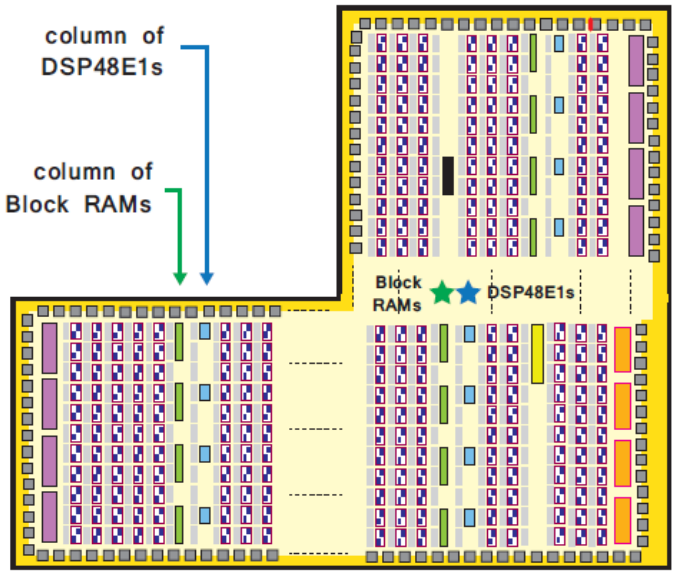


Figure 3.5: DSP and RAM blocks in the logic fabric [12].

3.2. Zynq Architecture

A System-on-Chip (SoC) is a single chip capable of implementing multiple system functionalities instead of using multiple physical chips to achieve the same purpose. It presents itself as a cheaper solution, with fast data transfer between the different elements present,

smaller physical size, low power consumption and reliable results [12]. Yet, current SoC designs lack in the flexibility and modularity needed by certain markets and applications. With SoC, the upgrade or replacement of a specific component would often require the replacement of the entire chip.

This need for a more flexible solution led to Xilinx designing a SoC implemented on a programmable, reconfigurable instrument. Therefore, the Zynq device is an All-Programmable System-on-Chip (APSoC) as it combines a dual-core ARM Cortex-A9 processor with the traditional FPGA logic fabric discussed in the previous Section 3.1.2.

3.2.1. Processing System and Programmable Logic

All things considered, the architecture of a Zynq device can be split into two components: a Processing System (PS) formed around the dual-core ARM Cortex-A9 processor, and a Programmable Logic (PL), an FPGA. To better understand the architecture, a simple diagram is displayed in Figure 3.6.

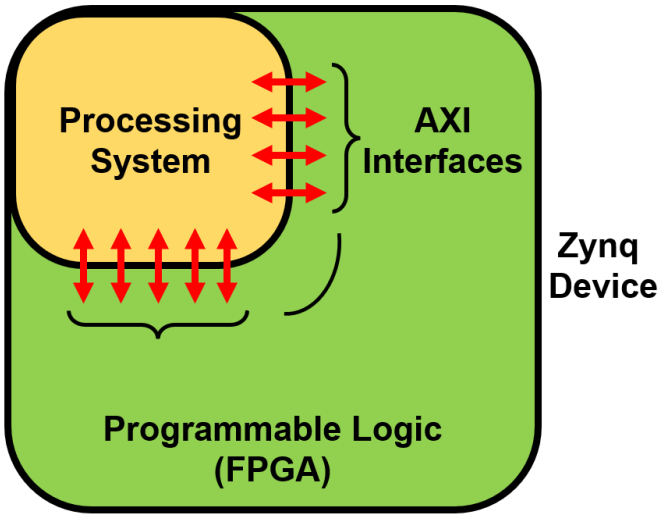


Figure 3.6: Simple scheme of the Zynq architecture.

PS

The PS part is responsible for running software routines or operating systems (for e.g. Linux [3]) or even both. Inside the PS, the Cortex-A9 processor is called the "hard processor" as it physically exists as an optimized silicon element on the device, achieving high performance. However, as an alternative, the Xilinx MicroBlaze is a "soft processor"

[3] `https://www.linux.org/` (last accessed: November 11, 2023)

formed through a combination of elements of the programmable logic fabric, being equivalent to an IP block design deployed on an FPGA. This processor has the advantage of the flexibility it can offer depending on the implementation, despite not reaching the same levels of performance when compared to a hard processor. The PS is further comprised of a list of complex components described in depth in [43], also highlighted in Figure 3.7.

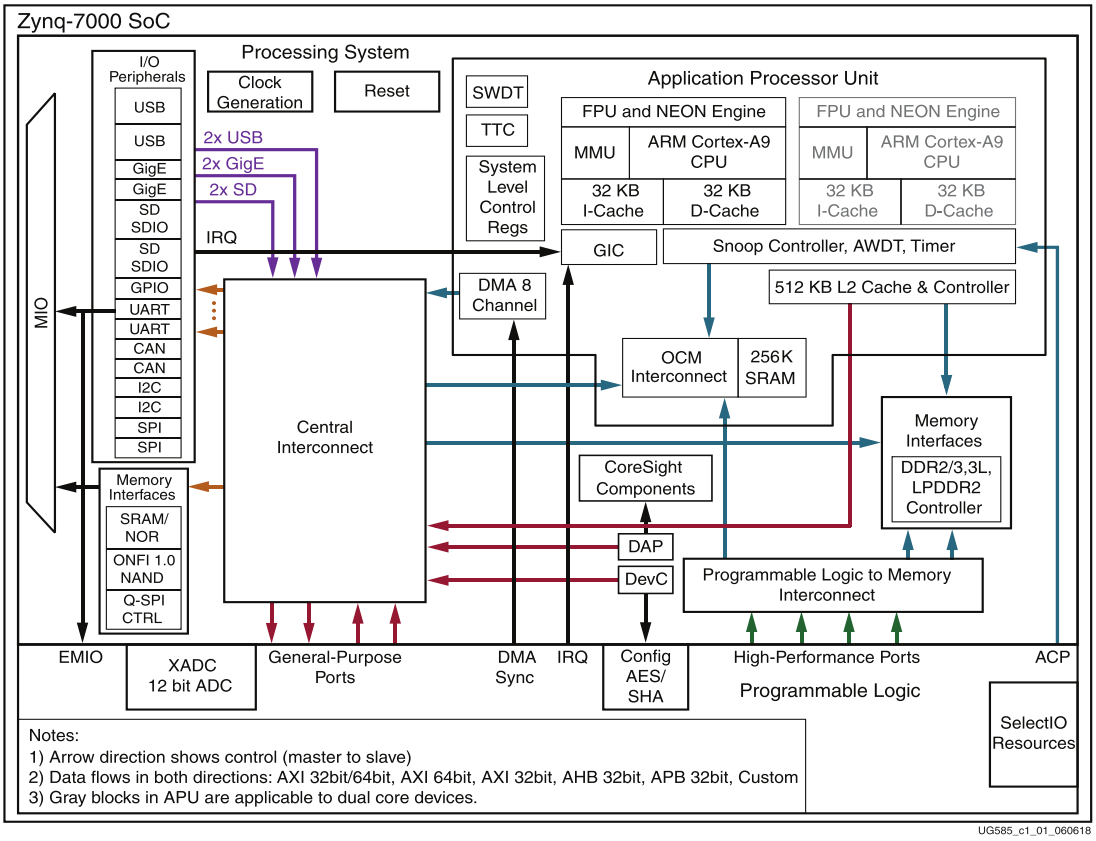


Figure 3.7: Zynq 7000 SoC overview [44].

PL

The PL part is responsible for the implementation of high-speed logic, arithmetic and data flow subsystems, being associated with the hardware component. It behaves as a typical FPGA, embedding all the elements and characteristics discussed in Section 3.1.

For the correct use of the device, the two parts are interfaced through the standard Advance eXtensilbe Interface (AXI) connections. This can be seen in Figure 3.7. A more detailed description and features of these connections is done in Section 3.2.2.

3.2.2. Processing System - Programmable Logic Interfaces

This section aims to list the different interfaces and interconnects available, the characteristics of each and explain in detail the AMBA AXI protocol.

AXI Interconnects and Interfaces

The primary link between the PS and the PL is done through a set of nine, for the case of Zynq-7000, available AXI interfaces.

Interface and interconnect are two different systems. Specifically, an interface is a "point-to-point connection for passing data, addresses, and hand-shaking signals between master and slave clients within the system" [12]. An interconnect, on the other hand, is "effectively a switch which manages and directs traffic between attached AXI interfaces" [12].

Of relevance for this work and by convention, the term master is related to the one who is in control of the bus and initiates transactions, while the slave responds.

This said, the different types of AXI interfaces available in the device, with a respective schematic being shown in Figure 3.8, are [44]:

- Accelerator Coherent Port (ACP): a single asynchronous interface with a bus width of 64 bits - used to achieve coherency access between the L2 cache and DDR memory present in the PS and elements within the PL.
- General Purpose (GP) AXI: four general purpose communication asynchronous interfaces with 32-bit data bus - suitable for low and medium rate communications. The PS is the master of two ports while the PL is the master of the other two.
- High Performance (HP) Ports: four high-performance interfaces with data width of either 32 or 64 bits - Each interface accommodates two FIFO buffers, supporting burst read and write behaviors. A burst mode is characterized by higher data throughput since the creation of a separate transaction for each data piece is avoided. Therefore it is typically used for high-rate communications between the PL and PS, the latter being always the master.

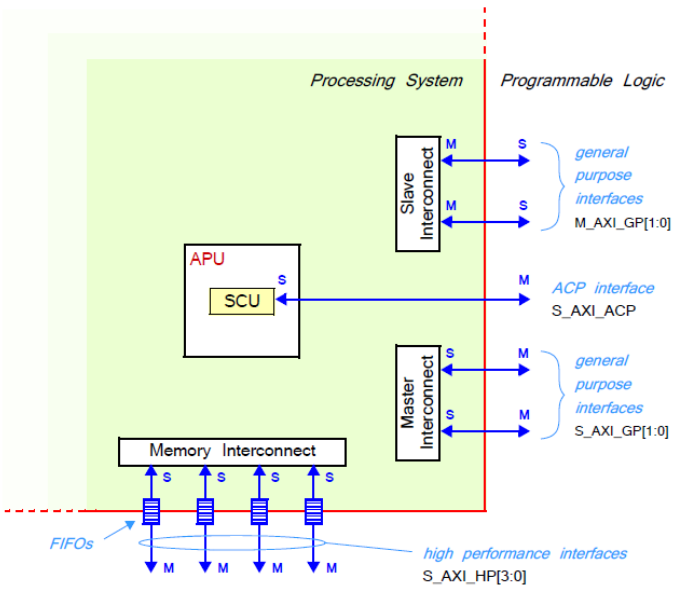


Figure 3.8: AXI interconnects and interfaces between the PS and PL [12].

A scheme of the interconnections of the Zynq PS is shown in Appendix A.0.1.

There are other types of connections (such as EMIO) that, for the scope of this work, are not exploited, with additional info available at [44].

The AXI Standard

AXI is a protocol belonging to the ARM Advanced Microcontroller Bus Architecture (AMBA), an open standard on-chip interconnect specification that allows the connection and management of IP functional blocks. In particular, the AMBA AXI specification defines the protocols optimized for FPGA implementation, targeting high-speed, high-frequency systems designs [4].

Considering the current AXI4 standard available, three types of application-dependent interfaces exist [45]:

- AXI4: a high-performance interface suited for memory-mapped links. It supports bursts of up to 256 data transfer cycles per address.
- AXI4-Lite: a lightweight variant of the interface, used in simple low throughput memory transactions. For this reason, it does not support burst data, only providing a single data transfer per transaction.

[4] `https://developer.arm.com/Architectures/AMBA` (last accessed: September 20, 2023)

- AXI4-Stream: used for high-speed streaming data scenarios, supporting an unlimited data burst size. In the need of bidirectional transfers, both peripherals must be of type master or slave as the connection is from master to slave type only.

The AXI protocol establishes an interface between a single AXI master and AXI slave, these two representing IP cores that exchange data with each other. However, thanks to an IP block called AXI interconnect, several AXI masters can be connected to several AXI slaves.

First, an in-depth analysis of the AXI4 and AXI4-Lite interfaces is done, as their flow architecture is distinct from that of AXI4-Stream. The first two interfaces consist of five different channels: Write Address Channel, Write Data Channel, Write Response Channel, Read Address Channel and Read Data Channel.

Figure 3.9 shows the write architecture. First, the address and control data is passed from the master to the slave, followed by a burst transfer of data. Lastly, the slave informs the master that the write operation was successful or not. Additional information about the different signals that compose a burst transaction can be found in [45].

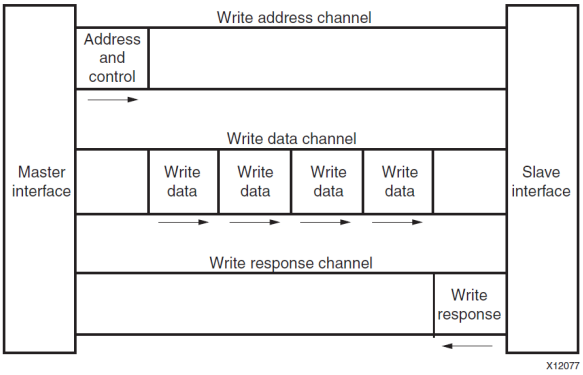


Figure 3.9: Channel architecture of write [45].

On the other hand, Figure 3.10 describes the read transaction. In this case, the master sends the address and control data to be written to the slave before a burst of read data is sent from the latter.

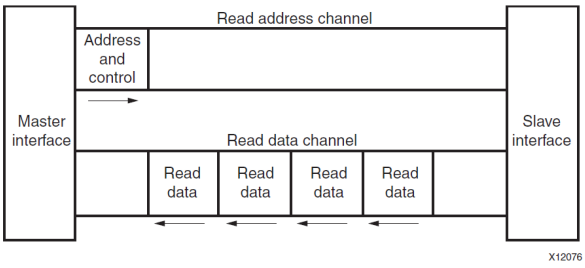


Figure 3.10: Channel architecture of read [45].

These write and read flows are present in both AXI4 and AXI4-lite interfaces. To emphasize, as separate write and read channels are offered, the capability for bidirectional transfers is supported.

Finally, the AXI4-Stream only has a single channel for the transmission of streaming data, modeling the write data channel of AXI4. This makes it ideal for applications where the concept of address is not required. It is composed of three different necessary signals: `TREADY`, `TVALID` and `TLAST`. The transfer starts once the producer sends the `TVALID` signal and the consumer responds by sending the `TREADY` signal. Then, the producer starts sending `TDATA` until `TLAST` is asserted. This signal alerts for the last byte of the stream [28]. Figure 3.11 highlights the explained behavior.

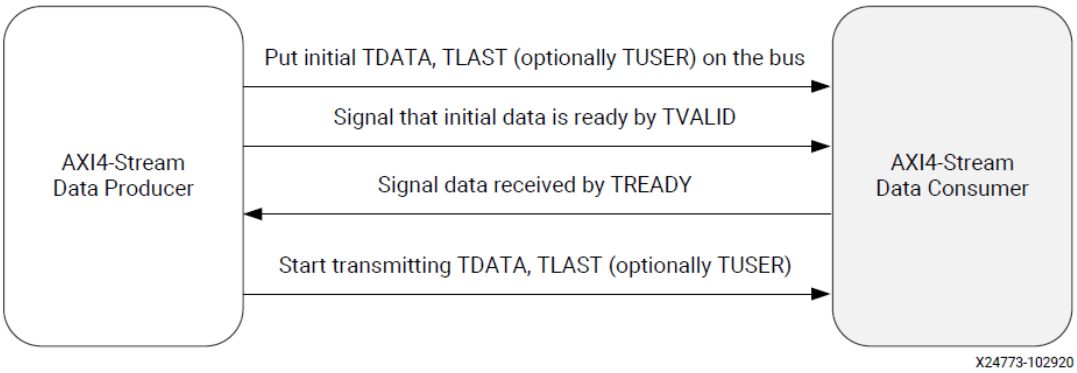


Figure 3.11: AXI4-Stream handshake [28].

This is the basic method to use an AXI4-Stream titled "AXI4-Stream without side-channels". A second one called "AXI4-Stream with side-channels" is available, presenting five additional control signals to be exploited. More information about the second method is available at [28, 45].

These different protocols are utilized and discussed in Section 4.3.2.

3.2.3. Zedboard

Within this thesis, a low-cost Xilinx Zynq ZC7Z020-1CLG484 APSoC [5], also known as Zedboard, development kit has been used. This board has already been used in other computational experiments such as video processing [8], embedded systems [7], control software [5], and others. Despite not being targeted as a space-graded or radiation-tolerant device, it is aligned with the needs and experiments conducted in this work. In fact, previously used by National Aeronautics and Space Administration (NASA) in an experiment for ionosphere readings [6], the board, for the scope of this thesis, can be seen as a starting point to measure the feasibility of the deployment of algorithms such as the ones explored here.

The technical characteristics of the board are reported in Table 3.2. Based on the Artix-7 logic fabric, Table 3.3 highlights the resources available on the PL side.

	Flash Memory [Mbit]	**DDR3 Memory [MB]**	**Clock - PS [MHz]**	**Clock - PL [MHz]**
Quantity	256	512	33.33	100

Table 3.2: Characteristics of the Zedboard.

	Logic Slices	**DSP48E1s**	**Block RAMs**
Quantity	13300	220	140 (36 Kb each)

Table 3.3: Resources available in the PL.

Moreover, the peripheral interfaces are highlighted in Figure 3.12.

The board offers different programming, debugging and booting methods, with the full list being available in Appendix A.0.2. In this work, JTAG-based procedures were used as identified as the most convenient. This is due to the simple configuration (single cable connection) and the real-time debugging capabilities it offers.

[5]Zedboard Datasheet (last accessed: September 21, 2023)

[6]NASA Team Miniaturizes Century-Old Technology for Use on CubeSats (last accessed: September 21, 2023)

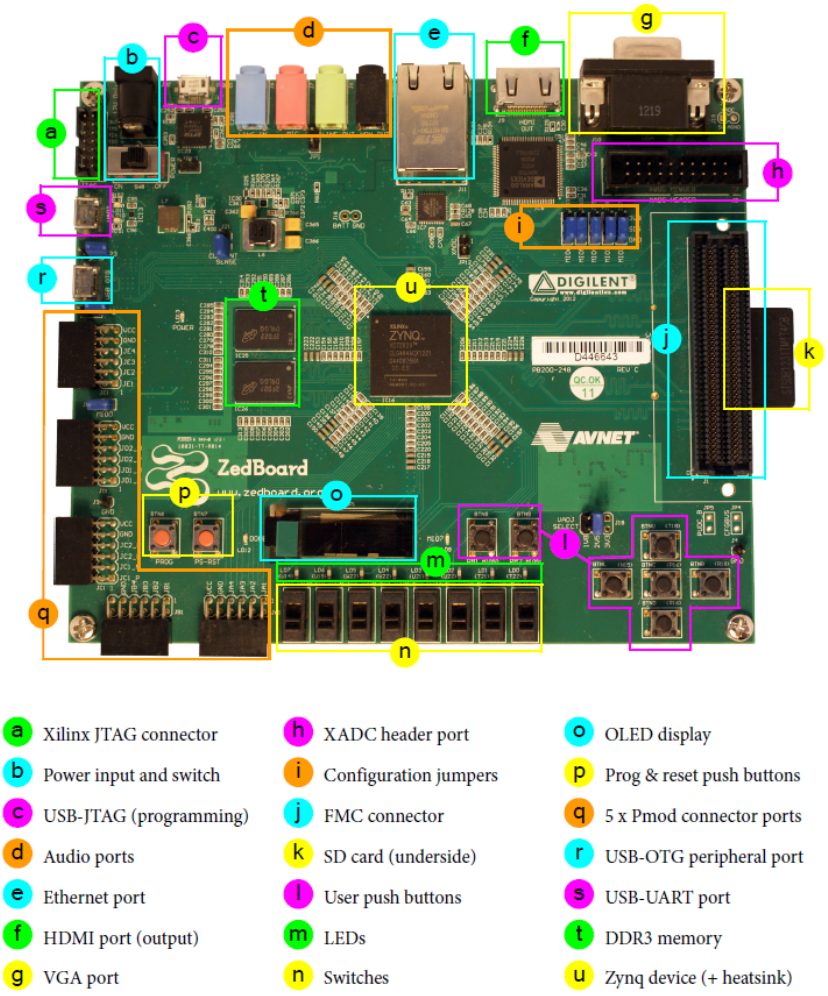


Figure 3.12: Zedboard and the different interfaces [12].

3.3. Development Tools

From the vast number of design tools offered by Xilinx [7], of relevance for this work, and that were used extensively, there are:

- Vivado: an integrated development graphical tool for creating the hardware system part of the SoC design. Besides, it provides synthesis, implementation for placing and routing, RTL design using HDL and bitstream generation capability. In addition, it enables the integration and packaging of other IPs, enhancing reusability [46]. This tool is essential for translating the desired HDL functionalities into gate-level descriptions.
- Vitis Integrated Design Environment (IDE): a software design used for the devel-

[7] `https://www.xilinx.com/products/boards-and-kits.html` (last accessed: November 11, 2023)

opment of embedded software applications that target Xilinx embedded processors. Based on the Eclipse platform, it includes driver support for all Xilinx IPs and GCC library support for ARM extensions using C or C++ languages. Essentially, it is responsible for the software aspect of the hardware design created and exported from Vivado [47].

- Vitis HLS: used for creating, testing and managing IP which, later on, is included in the hardware system. It is a design tool that synthesizes a C or C++ function into RTL for implementation in the PL region of the Zynq. This method is particularly powerful as it enables the creation of an optimized hardware subsystem from a high level of abstraction (such as C or C++) without the need to meddle with RTL code [28]. This particular tool will be subject to further discussion in Section 3.4.1 as it comprises a set of functionalities of interest for the intent of this work.

Figure 3.13 presents a guiding summary of the functionality of each software, showing how they are related and also the usual flow of an application development and deployment.

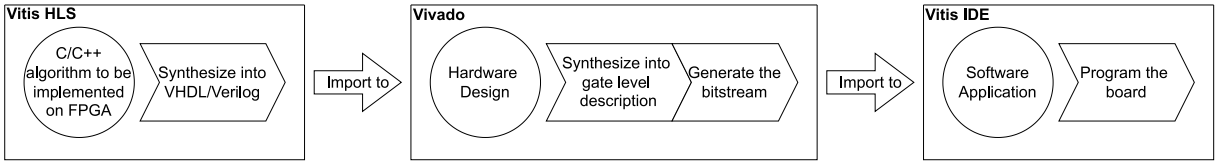


Figure 3.13: Fundamental diagram of the different software and their functionalities.

3.4. Development and Implementation of IP cores

After introducing in Section 3.1 what an IP core is, the present section presents the different workflows available, reasoning also for the chosen method for this work. The set of methods which enable the creation of IPs are:

- HDL: specialized programming languages that allow the user to have maximum control over the functionality of his peripheral, as seen before. If the solution requires tight hardware or timing constraints, this method is, usually, the best one [12]. The biggest disadvantage is in the complexity of the coding which, consequently, leads to a demanding process of development and testing, requiring a specialized user to work with it.

- Vitis Model Composer: software tool, provided by Xilinx, that utilizes the Mathworks Simulink design platform for the generation of synthesisable HDL code which can be integrated in hardware designs [48]. Specifically targeted for Digital Signal

Processing (DSP) applications, it presents the capability of simulating the design before moving it into the hardware. The actual deployment is easily achieved, with automatic test bench generation and verification being part of this process. Capable of receiving custom IP, it also offers a catalog of blocks for usual operations frequently integrated into designs to achieve higher performances.

- HDL Coder [8]: MATLAB add-on capable of generating synthesisable HDL code from both MATLAB functions and Simulink models. It automatically analyzes the function or model under analysis, converting it from floating-point to fixed-point, an important concept discussed in Section 4.5.
- Vitis HLS: software tool, provided by Xilinx and presented in Section 3.3, capable of synthesizing RTL implementations from C or C++ code. Being at a higher level of abstraction when compared to the previous methods, it enables the developer to focus on the functionality and performance of the algorithm to be deployed.

Considering the aforementioned characteristics, Vitis HLS has been the selected tool to employ. Firstly, it is a robust solution capable of generating high-performance designs, even when compared to RTL solutions. However, while both can present similar performances, the development time is immensely different as seen by Figure 3.14. This enables the user to focus on the IP core design and optimization, leading to improved hardware designs and removing the need to think at low-level implementation code. Secondly, as the algorithm presented in Section 2.3 is developed in C code, its compatibility with Vitis HLS is granted. This promotes easy integration between the algorithm code to deploy in the FPGA and the IP development tool.

[8] `https://www.mathworks.com/products/hdl-coder.html` (last accessed: November 11, 2023)

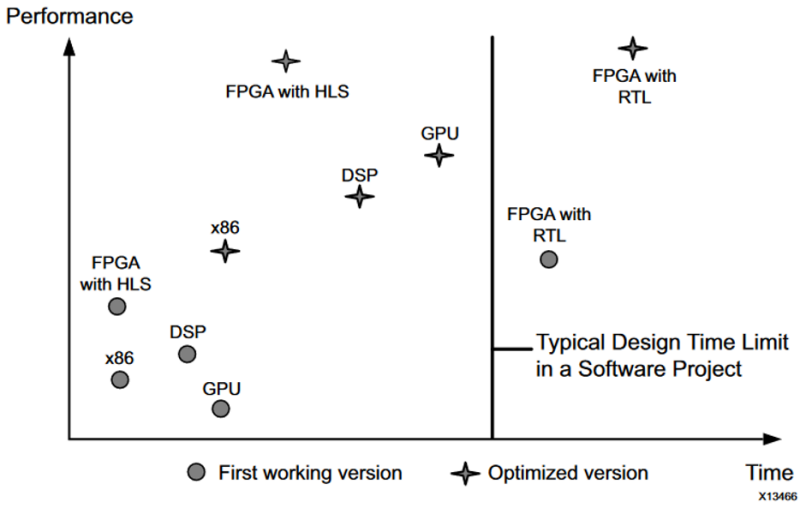


Figure 3.14: Design time vs application performance with RTL and Vitis HLS compiler [28].

3.4.1. Vitis HLS Deep Dive

After selecting Vitis HLS as the chosen tool to generate the IP cores used in this work, their selection and reasoning being explained in detail in Section 4.2, it is important to, first, go over which aspects of the design should be optimized and how to do it. Then, an overall view of the different steps to be made for the correct use of the software is done. This will be useful in Section 4.3 to understand the reasoning behind certain decisions.

Primary aspects of the design

When considering the optimization of a design, the two most important metrics to be highlighted are:

- Resource cost: the amount of hardware required by the application to perform the desired functionality.
- Throughput: the rate at which the circuit can process data. Explicitly, the latency (either in number of cycles or ns) required by the design.

A trade-off between these two has to be made, being up to the designer to choose which one to prioritize depending on the specific application and requirements of his design.

Then, an important distinction between two aspects of any HLS design are: its interfaces (how the IP core will communicate with the PS) and the functionality of the design itself (the algorithm to be accelerated using the FPGA).

To control the aforementioned characteristics and the behavior of the HLS process, the following two techniques are available:

- `Pragmas`: directives which guide the tool's behavior over aspects of the RTL implementation, enabling the designer to dictate how certain sections of the code should be treated to match the desired requirements. They also play a vital role in the selection and customization of the interfaces to be used in the design, being further explored in Section 4.3.2. All the different types of `Pragmas` available on Vitis HLS, each tailored for specific optimizations, are listed in [28].
- Constraints: limits specified by the developer on some aspect of the design. These can either be number of resources or timing restrictions, with the application providing a report that tells if the implementation meets the requirements of the bigger system into which will be later integrated.

Optimization techniques

There are two paradigms when searching for optimization of IP core designs: Pipelining and Data flow.

Pipelining is a common term in hardware designs referring to the minimization of the critical path, the "longest combinatorial logic path between clocked elements" [12]. The goal is to reduce the Initiation Interval (II), so the number of clock cycles between the launch of successive operations within a function or loop, therefore increasing the system's throughput. This can be achieved by overlapping independent operations as seen in Figure 3.15, where the II is reduced from 60 clock cycles to 20. Important to notice that pipelining does not decrease the latency, so, the time required for one item to go through the entire system. However, by decreasing the II between iterations, it does increase the total latency, therefore resulting in a higher system throughput.

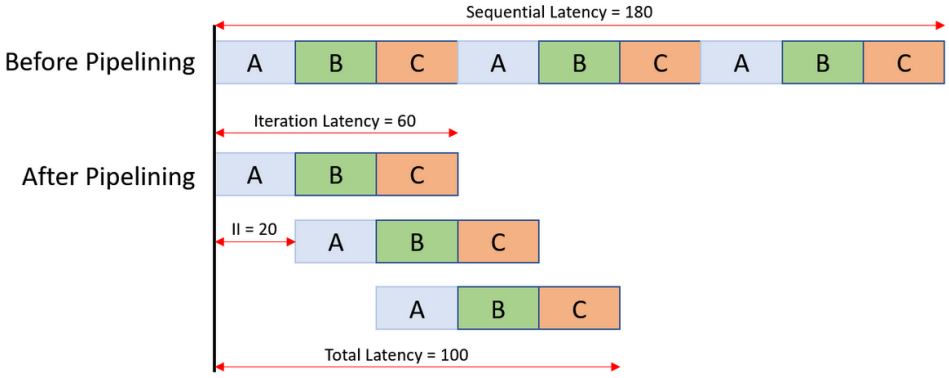


Figure 3.15: Pipelining example [28].

Dataflow (also known as dataflow pipelining) is a similar concept, yet, it targets the concurrency between functions or loops. While pipelining explored an increase in the hardware throughput through software operations overlap, dataflow targets the parallelization of function and loops by, physically, increasing the number of dedicated resources. This parallelization paradigm is one of the main advantages FPGAs can offer. Of course, this process leads to an increment of resource costs. Also, a critical aspect of dataflow is the data dependencies between operations. Figure 3.16 perfectly illustrates this by showing that the 3 functions can not be parallelized to start at the same clock cycle but are, indeed, delayed to comply with data dependencies.

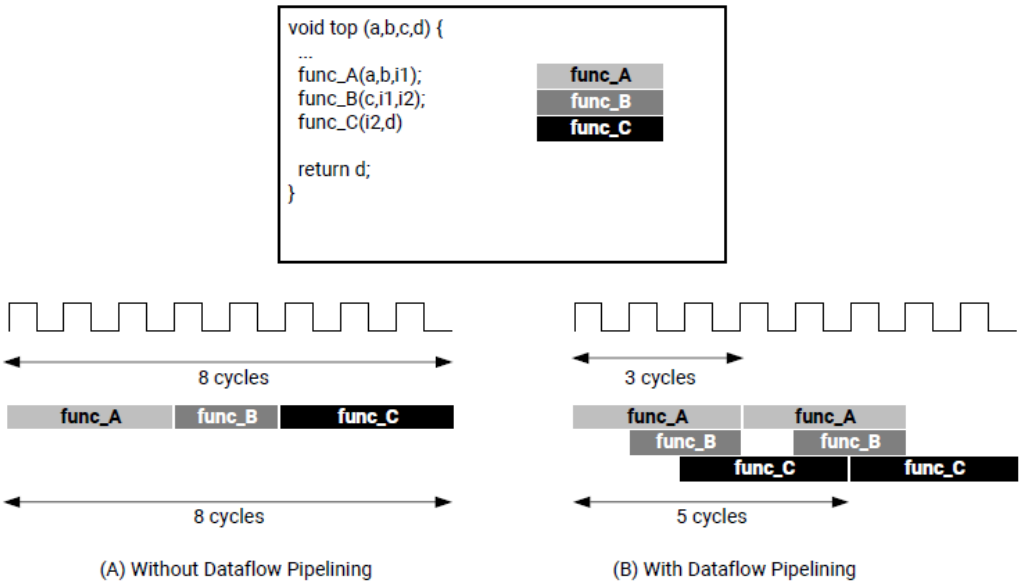


Figure 3.16: Dataflow example [28].

The approaches presented before can be explored in Vitis HLS through the use of `Pragmas`, such as *Pragma HLS pipeline* for pipeline purposes and *Pragma HLS dataflow* or *Pragma HLS unroll* for dataflow intents, with many others being available at [28]. This topic is further discussed in Section 4.3.3.

Design Flow

After a brief introduction about the different aspects to take into account during an HLS design, as well as exploring two possible sources of optimizations, a general view of the different steps for the correct use of the software tool is done in this section.

Figure 3.17 illustrates an overview of a typical design flow. There are four different main stages: C Simulation, C Synthesis, C/RTL Cosimulation and Implementation.

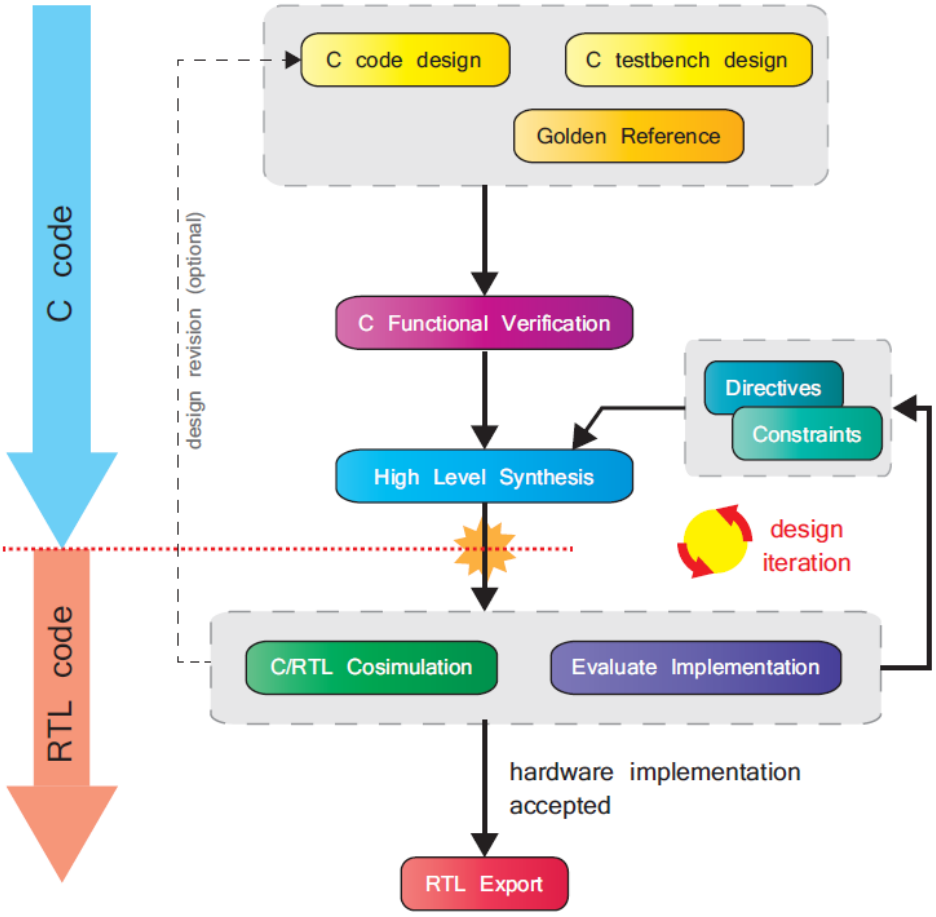


Figure 3.17: Overview of the Vitis HLS workflow [12].

Setup

For simplicity purposes, let's take, as an example, the development of a matrix multiplication IP core. The first step would be to develop the C or C++ function that takes two input matrices, performs the multiplication and then retrieves as output the result. This will be the target function we want to translate into RTL code and implement on the FPGA. Specification on the type of interfaces to use is done at this level by the use of `Pragmas`. After, a test bench file is written, this being a practical example that validates the developed function by comparing its results against previously known results (normally named Golden Reference). This validation, previous to any generation of RTL code, guarantees the correct behavior of the C or C++ functions, decreasing subsequent errors. In this example, this file would initialize two matrices with known values, call the function we want to implement in the hardware and, lastly, compare its output with the expected matrix result.

C simulation

After creating all the before mentioned files, the "C Simulation" step can be run. The objective is to validate the logic of the source code to be implemented on the FPGA by using the provided test bench file and golden data. If the results are correct, the test bench returns a value of zero, otherwise, it returns any non-zero value [28].

C Synthesis

The next phase entitled "C Synthesis" analyzes and processes the C or C++ code, previously developed and validated, to create the equivalent RTL description of the circuit. At the end, a Synthesis Report is generated, giving an estimate on the timing, performance and number of resources of the design [28]. With this report, the user can have a preliminary idea of the performance of the design and compare it with the established requirements. If unsatisfied with the results, `Pragmas` and Constraints can be added following the metrics to be achieved.

C/RTL Cosimulation

After producing the equivalent RTL model, its functionality is checked by comparing it to the original C or C++ code through the "C/RTL Cosimulation". This process reuses the previously defined test bench file to supply inputs to the generated RTL design and, subsequently, check if the produced outputs match the expected values (the golden reference file). It is noteworthy to mention that this stage presents timing behaviors that are more coherent and accurate with the actual performance the hardware will achieve, allowing for a more detailed analysis of the metrics to be attained [28].

RTL Export

Finally, once the design has been validated and the implementation has been iterated to achieve the required criteria, it is time to export it as an RTL IP core. This will, later on, be integrated into a larger system, as explained in the next section. It is pertinent to note that Vitis HLS supports both VHDL and Verilog as the HDL options to export the design.

3.4.2. IP Block Integration

Now that the IP core has been developed and synthesized into a RTL, it needs to be integrated into the system. This is the transition point between the Vitis HLS tool and

Vivado, as illustrated before in Figure 3.13.

The Vivado IP Integrator is a powerful feature within Vivado that enables the configuration and integration of IP cores with larger system designs. By providing a graphical and Tool Command Language (TCL) development environment, it offers an automated development flow that automatically connects the fundamental IP interfaces needed to grant the right integration of the IP core in the whole system [12].

The tool also provides features such as IP core configuration and customization or even debugging capabilities, which will be of relevance for later stages of this work.

3.5. Fixed-Point Arithmetic

When working with FPGAs, the specification of the data type to use is an important aspect that directly impacts the integrity and key metrics of the design. For numerical data, under-specifying the word length can lead to accuracy compromises while over-specifying can introduce unnecessary amounts of resources or sub-optimal maximum clock frequencies. For optimal hardware designs, no more bits than the necessary ones should be used. Additionally, FPGA designers employ fixed representations whenever possible due to the speed reduction that floating-point deployments present in hardware [49]. For this, an important distinction between floating-point and fixed-point precision is made.

Floating-point precision represents numbers through a fixed set of significant digits scaled using an exponent in some fixed base. This representation is composed by a sign bit S, several exponent bits E and several mantissa bits M. Figure 3.18 shows, as an example, the particular floating-point representation of double and float types.

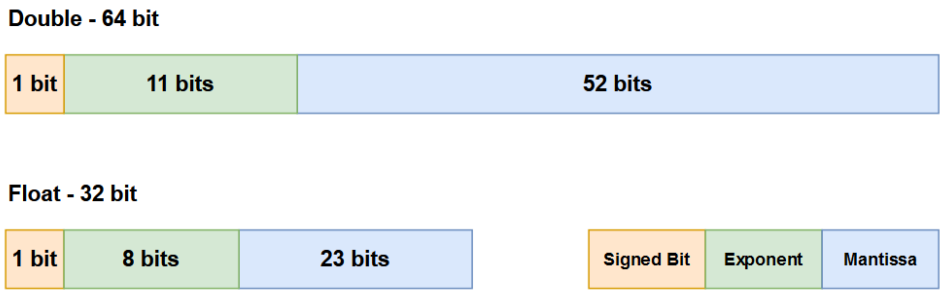


Figure 3.18: Floating-point representation example.

Moreover, every data type can represent a range of values, collected in Table 3.4 for the three data types of relevance for this work.

Type	Number of bits	Range
int (IEEE 754 [50])	32	-2,147,483,648 to 2,147,483,647
double (IEEE 754 [50])	64	$\pm$ 2.225e^{-308} to $\pm$ 1.798e^{308}
float (IEEE 754 [50])	32	$\pm$ 1.175e^{-38} to $\pm$ 3.403e^{38}

Table 3.4: Floating-point range for presented data types.

On the other side, fixed-point precision represents numbers through a fixed number of digits before and after the radix point. This enables the choice of the exact quantity of fixed and integer places to consider, following the application requirements, allowing for arbitrary amounts of bits. For FPGAs, this conducts in fewer resources used to represent numbers and perform operations. Its particular definition and implementation in Vitis HLS is explained in Section 4.5.

Concerning the arithmetic operations, an element to recognize is the resultant amount of bits certain operations produce. For example, the multiplication of two 18-bit numbers gives a 36-bit results. If not handled correctly, this can lead to bit overflow scenarios. Therefore, an important outlook to consider when developing an application with fixed-point precision is the agreement of the number of bits of each data type with the operations to be carried out.

A last regard is the performance difference certain operations between the two precision types present. For floating-point, additional computation to handle the exponent and mantissa part of a number is required. The wide range of magnitudes and variable precision available is counteracted by additional processing and storage operations. On the other hand, fixed-point precision presents greater efficiency for certain operations since they can directly operate with the same units [49].

A wise choice of bits for fixed cases, respecting the sizes required by the application, results in higher quality hardware implementations characterized by fewer resources and better performance while preserving accuracy. In [49], a study of the difference in performance and resource usage for several arithmetic operations using both data types in FPGA devices is presented.

4 | Implementation

This chapter delves into the deployment of the SCP algorithm presented in Section 2.3. First, the solution of the Earth to Mars minimum-fuel trajectory optimization problem is presented. Subsequently, the deployment of the application in the PS of the Zedboard and a profiling sequence is done. This stage conducts the functions or sections to target the FPGA. An overview of the operations of each function is laid out. Following this, the workflow for elaborating and implementing the selected IP cores into the whole system is shown, discussing the reasoning for the interfaces used in this work. Due to memory and resource constraints, a reduction of the original problem is justified and evaluated. Lastly, a discussion about fixed-point arithmetic and its implementation for this work is done.

4.1. Sequential Convex Programming Algorithm

For this work, a minimum-fuel STO problem between Earth and Mars is solved considering the algorithm and concepts presented in Chapter 2. The set of conditions considered for the problem, normalized and referred to 100 nodes, are:

$$\mathbf{x}_l = [0;\ 1;\ 0;\ -\pi;\ -10;\ -10;\ -10;\ \ln(0.1)], \tag{4.1a}$$

$$\mathbf{x}_u = [10;\ 10\pi;\ \pi;\ 10;\ 10;\ 10;\ 0], \tag{4.1b}$$

$$\mathbf{u}_l = [-10;\ -10;\ -10], \tag{4.1c}$$

$$\mathbf{u}_u = [10;\ 10;\ 10], \tag{4.1d}$$

$$R = [1;\ 5\pi;\ 10\pi;\ 0.4;\ 0.4;\ 0.4;\ 10], \tag{4.1e}$$

$$\boldsymbol{\varepsilon} = [10^{-6};\ 10^{-4}], \tag{4.1f}$$

$$\gamma = 0.7, \tag{4.1g}$$

$$\mathbf{x}(t_0) = [1;\ 0;\ 0;\ 0;\ 1;\ 0;\ 0], \tag{4.1h}$$

$$\mathbf{x}(t_f) = [1.5236;\ 3.1415;\ 0.0322;\ 0;\ 0.8101;\ 0], \tag{4.1i}$$

where the convergence variable $\boldsymbol{\varepsilon}$ is split into two values, the first one related to the

first six components of the state, and the second one to the last component of the state. Moreover, the engine technical data used can be found in [4].

The Pseudo-algorithm 4.1 presents a scheme and logic for the SCP algorithm constructed around the code attained from CVXPYgen, being coherent with the concepts presented in Section 2.3 and Figure 2.3.

Algorithm 4.1 Sequential Convex Programming Algorithm

Inputs: Initial and final state vector $\mathbf{x}(t_0)$ and $\mathbf{x}(t_f)$, initial and final time instants t_0 and t_f, number of nodes N, convergence criteria ε, trust region R and cauchy parameter γ

Generate reference trajectory $\mathbf{x}^{(0)}$ ▷ 3rd order polynomial

Set k = 1

for $(k < k_{max})$ **do**

  Solve Problem 2.12 to find $\mathbf{y}^{(k)} = \left\{\mathbf{x}^{(k)}, \mathbf{u}^{(k)}\right\}$ ▷ ECOS

  Check the convergence condition:

  if $sup \ \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_1 \leq \varepsilon$ **then**

    Optimal solution found: $\left\{\mathbf{x}^{(k)}, \mathbf{u}^{(k)}\right\}$

    Check minimum fuel mass: $m(t_0) - m(t_f)$

    Check objective function: J

    break

  end if

  Update reference trajectory $\mathbf{x}^* = \mathbf{x}^{(k)}$

  Update cauchy parameter γ

end for

The solution to the problem, after running the C code of the Pseudo-algorithm 4.1 with the conditions stated above, is displayed in Section 5.3.

4.1.1. Deployment in the PS

After developing, compiling and executing the C code algorithm for the problem under study on a desktop computer, the next step is to deploy it in the PS of the Zedboard. Once verified, a profiling technique can be applied to understand which functions to deploy in the FPGA as Section 4.2 presents.

The deployment in the PS is straightforward with the use of Vitis, with only some small

remarks to take into account. First, a conversion of the POSIX/UNIX routines present in the original C implementation (such as `CLOCK_MONOTONIC`) to routines which are compatible with a bare metal implementation on the Zedboard is carried out. This leads to the need to implement compatible timing mechanisms, a point explored in Section 5.2.1. Similarly, ECOS includes a set of files which, despite not being used by the developed algorithm, will cause conflicts in the compilation act ("multiple definition of main error"), therefore being removed. Lastly, a correct linker script structure is needed to grant the appropriate execution of the code. A linker script is a configuration file responsible for specifying the memory layout and address mapping of the target device. It defines the memory regions to be used and how the code and data are placed in memory. The user should stipulate two fields [12]:

- Stack: area of computer's memory which stores temporary variables declared, stored and initialized during the runtime of a function.
- Heap size: section responsible for storing global variables.

If not correctly fixed, the program stack can continuously grow until a non-accessible memory region is accessed (stack-overflow).

Following the correct setting of the problem, its deployment was done into the board through the USB-JTAG programming mode. The results are, once again, presented in Section 5.3.

4.2. IP Cores Selection

A benchmark of the algorithm was performed with the intent of identifying possible hardware implementations through IP Cores. These aim the improvement of the application throughput.

4.2.1. Profiling

In many embedded systems, the processing bandwidth or throughput of an application is limited by the processor's performance. Therefore, an application computational time profiling was executed to identify its execution-time bottlenecks.

To not alter the program execution flow, the Vitis' non-intrusive Target Communication Framework (TCF) profiler has been adopted to sample the Program Counter through the debug interface. However, when the stack trace is also enabled, the program execution speed decreases due to the presence of code memory probing [47]. In this case, this is not

a problem, as the interest is to identify the relative execution time and not the absolute atomic one.

Having said that, Figure 4.1 shows the profiling information for the application under study deployed in the PS as exhibited in Section 4.1.1. The first four columns provide, respectively, the address for the function, its exclusive percentage, its inclusive percentage and its name. The fifth and sixth columns provide the file where that function is located and the corresponding line.

The important metrics to take into account are the % exclusive and % inclusive columns. These are defined as [47]:

- % exclusive: the percentage of samples for that function only, excluding samples of any child functions. A child function is a function executed within the current function [47].
- % inclusive: the percentage of samples of a function, including samples collected during the execution of any child functions [47].

As anticipated, the functions that consume the greatest execution time and, consequently, impact the performance of the application, are all associated with ECOS processes. This is expected since, from the Pseudo-algorithm 4.1, it is reasonable to assume the program spends most of its time inside the interior point solver. Particularly, the `Solve` step of ECOS highly impacts the execution time. This is because an STO problem heavily relies on the execution of the chosen solver. Moreover, all of the candidates below work with static memory allocation and are suited for possible FPGA implementations.

Address	% Exclusive	% Inclusive	Function	File	Line	Details
0016b6a0	.000	100	_start	xil-crt0.S	61	[+]
00153d08	.000	100	main	main.c	44	[+]
00153020	.000	99.9	SCP	functions.c	94	[+]
00150d38	.173	99.9	cpg_solve	cpg_solve.c	6409	[+]
0015ae24	2.00	98.2	ECOS_solve	ecos.c	1076	[+]
0015ef5c	7.11	67.9	kkt_solve	kkt.c	88	[+]
0015ee40	.000	16.4	kkt_factor	kkt.c	43	[+]
0016488c	16.4	16.4	ldl_numeric2	ldl.c	289	[+]
00162ea8	15.0	15.0	sparseMV	spla.c	42	[+]
00164e7c	14.4	14.4	ldl_lsolve2	ldl.c	393	[+]
00165084	12.9	12.9	ldl_ltsolve	ldl.c	433	[+]
00163154	11.6	11.6	sparseMtVm	spla.c	82	[+]
00158660	.000	5.11	computeResiduals	ecos.c	456	[+]
001635cc	3.83	3.83	norminf	spla.c	144	[+]
00156c54	2.73	2.73	unstretch	cone.c	542	[+]
00157b48	.069	2.59	init	ecos.c	261	[+]
001593dc	.795	2.10	RHS_combined	ecos.c	689	[+]

Figure 4.1: Total execution time percentage profiling for 100-nodes application.

For this work, it was decided to optimize the functions with higher values of exclusive percentage. This logic is because while a high inclusive percentage indicates that the function executed for a long time, the sampling information of its child functions is also included in it. This could mean that a lot of the processing activity may not result from the function itself but, indeed, from its child functions. In contrast, functions with a high exclusive percentage point out the operations, through the entire application, that took the highest amounts of processing.

Analyzing Figure 4.1, the functions selected for IP core development, and subsequent deployment in the PL, are `kkt_factor` and `sparseMV`.

`kkt_factor` is a perfect example of a function with a high inclusive percentage but a low exclusive percentage. It contains `ldl _numeric2` as its child function responsible for all the processing. The first function serves essentially as a wrapper for the second one. Thus, by implementing `kkt_factor`, the actual implementation and focus is on the `ldl_numeric2` function. Being the first function to be implemented and deployed as an IP core in the board, its choice was uniquely due to the high execution time it takes over the application.

Then, the `sparseMV` function was chosen for the simplicity of operations and arithmetic it presents, providing a straightforward integration. This point is further discussed in Section 4.5.

4.3. IP Design

4.3.1. Functional Analysis

The functions previously selected in Section 4.2.1 have to be analyzed in terms of inputs and outputs, their data structures, and functionality. The aim is to produce IP cores that could replicate the original purpose of each function in a hardware oriented environment.

As general remark, the minimum amount of inputs and outputs needed for the function operation should be passed to the FPGA. Likewise, its performance should be maximized through, for example, the optimization techniques discussed in Section 3.4.1. This, nonetheless, will not be possible due to aspects further discussed in Section 4.3.3.

As a common practice in C implementations, the C version of ECOS greatly explores the use of pointers and structures throughout the code. As a result, in most cases, functions used in the algorithm have pointers to their input or output variables. Instead, due to the type of interfaces chosen for the IPs, a different approach, explained in detail in Section

4.3.2, is considered. The upcoming paragraphs explain some of the critical elements taken into account for each function.

Numeric Factorization Function

The purpose of `kkt_factor` is to compute the numeric factorization $\mathbf{A} = \mathbf{LDL^T}$, with $\mathbf{A}$ being a symmetric sparse matrix. It is a variant of the classical Cholesky decomposition [51]. ECOS implements it through the LDL software package [52], a set of concise routines for factorizing symmetric positive-definite sparse matrices.

Let $\mathbf{Ax} = \mathbf{b}$ be a linear system with $\mathbf{A}$ symmetric and let $\mathbf{PAP^T} = \mathbf{LDL^T}$ be a factorization of matrix $\mathbf{A}$, where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a unit lower triangular matrix and $\mathbf{D}$ is a block diagonal matrix. The solution to the original linear system can be computed iteratively, at reduced computational cost, by the sequence of equations shown in (4.2) which involve permutation, diagonal and lower triangular matrices.

$$\begin{aligned} \mathbf{Lu} &= \mathbf{Pb}, \\ \mathbf{Dv} &= \mathbf{u}, \\ \mathbf{L^T w} &= \mathbf{v}, \\ \mathbf{x} &= \mathbf{P^T w}, \end{aligned} \tag{4.2}$$

where, $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are intermediate vectors [53]. Therefore, the factorization process of matrix $\mathbf{A}$ is a pivotal part in the execution of iterative solvers, such as ECOS, as it enables computationally efficient and numerically stable operations [25, 54].

The Pseudo-algorithm 4.2 is a simplified version of the function operation, where l_{kk}, d_{kk}, a_{kk} are, respectively, the kk entry of matrices $\mathbf{L}$, $\mathbf{D}$ and $\mathbf{A}$.

Algorithm 4.2 $\mathbf{LDL^T}$ factorization of a n-by-n symmetric matrix $\mathbf{A}$

Inputs : Matrix $\mathbf{A}$ and nonzero pattern of matrix $\mathbf{L}$

for ($k = 0$ to n) **do**
  Find $\mathbf{y}$ by solving $\mathbf{L}_{1:k-1,1:k-1}\mathbf{y} = \mathbf{A}_{1:k-1,k}$
  $\mathbf{L}_{k,1:k-1} = (\mathbf{D}^{-1}_{1:k-1,1:k-1}\mathbf{y})^T$
  $l_{kk} = 1$
  $d_{kk} = a_{kk} - \mathbf{L}_{k,1:k-1}\mathbf{y}$
end for

In this case, the symmetric matrix to factorize is $\mathbf{KKT}$, shown in equation (4.3), result

of a composition of the SOCP matrices $\mathbf{A}$ and $\mathbf{G}$, and a primal dual-scaling $\mathbf{W}$ [53]. All passages and algebraic manipulations to obtain this matrix are demonstrated in [25].

$$\mathbf{KKT} \equiv \begin{bmatrix} 0 & \mathbf{A^T} & \mathbf{G^T} \\ \mathbf{A} & 0 & 0 \\ \mathbf{G} & 0 & -\mathbf{W}^2 \end{bmatrix}. \tag{4.3}$$

For clarity purposes, Listing 1 shows and contextualize the inputs and outputs of the `ldl_numeric2` mentioned. Some of the variables are strictly related to the nomenclature used by the software package.

Listing 1 Inputs and outputs of `ldl_numeric2` function

```
int kkt_factor(kkt* KKT, double eps, double delta) {
    int nd;

    nd = LDL_numeric2(
        KKT->PKPt->n,    // size of matrix PKPt
        KKT->PKPt->jc,   // index of elements in PKPt->pr which
                         // start a column of matrix PKPt, size n+1
        KKT->PKPt->ir,   // stores the row index of each entry of matrix
                         // PKPt, size lnz=Lp[n]
        KKT->PKPt->pr,   // input of size nz=Kjc[n]
        KKT->Parent,     // elimination tree, size n
        KKT->Sign,       // input, permuted sign vector for regularization,
                         // size n
        eps,             // value of inverse permutation vector
        delta,           // value of dynamic regularization
        KKT->Lnz,        // # of nonzeros below the diagonal of L, size n
        KKT->L->jc,      // index of elements in L->pr which start a
                         // column of matrix L, size n+1
        KKT->L->ir,      // stores the row index of each entry of matrix L,
                         // size lnz=Lp[n]
        KKT->L->pr,      // non-zero numerical values of matrix L,
                         // size lnz=Lp[n]
        KKT->D,          // diagonal matrix, size n
        KKT->work1       // workspace, size n
        KKT->Pattern     // workspace, size n
        KKT->Flag        // workspace, size n
    );

   return nd == KKT->PKPt->n ? KKT_OK : KKT_PROBLEM;
};
```

Moreover, the size of the inputs and outputs is reported in Table 4.1. The data is transmitted through the interface described later in Section 4.3.2.

The `LDL` package works, throughout its different routines, with the n-by-n sparse matrix **A** in a Compressed Column Storage (CCS) format described hereafter [55]. Considering three different arrays **Ax**, **Ai** and **Ap**, exemplified in equation (4.4), **Ax** stores the non-zero numerical values of matrix **A**, **Ai** stores the row indices of each entry and **Ap** stores the index of the elements in **Ax** which start a column of matrix **A**. At the end of this last array, the total number of non-zero values is added.

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 8 & 0 & 0 & 19 \\ 0 & 44 & 0 & 0 \\ 0 & 0 & 30 & 15 \\ 23 & 0 & 6 & 2 \end{bmatrix}, \\ \mathbf{A_x} &= [8, 23, 44, 30, 6, 19, 15, 2], \\ \mathbf{A_i} &= [0, 3, 1, 2, 3, 0, 2, 3], \\ \mathbf{A_p} &= [0, 2, 3, 5, 8]. \end{aligned} \tag{4.4}$$

This format, however, leads to algorithms characterized by non-fixed size iteration loops, that is, the indices are not defined a priori. An example is appreciated through lines 7-9 and 13-15 of the Pseudo-algorithm 4.3. Moreover, `kkt_factor` presents `while loops`.

These cases of function loops with nondeterministic routines cause conflicts in the optimization techniques and Vitis HLS synthesis and implementation tools to carry out. The impact they have on the overall FPGA performance is analyzed in Section 4.3.3.

Matrix-Vector Multiplication Function

The `sparseMV` function performs the sparse matrix-vector multiplication operations presented in (4.5).

$$\begin{aligned} \mathbf{y} &= \mathbf{Ax}, \quad \text{if } a > 0 \text{ and } \text{newVector} = 1, \\ \mathbf{y} &+= \mathbf{Ax}, \quad \text{if } a > 0 \text{ and } \text{newVector} = 0, \\ \mathbf{y} &= -\mathbf{Ax}, \quad \text{if } a < 0 \text{ and } \text{newVector} = 1, \\ \mathbf{y} &-= \mathbf{Ax}, \quad \text{if } a < 0 \text{ and } \text{newVector} = 0. \end{aligned} \tag{4.5}$$

The Pseudo-algorithm 4.3 shows a simplified implementation of the operations above, being coherent with the CCS format presented before.

Algorithm 4.3 Sparse matrix-vector multiplication using CCS format

```
1: Inputs : Matrix A, vectors x and y and values newVector and add
2:
3: if  newVector > 0 then
4:    Set y = 0
5: end if
6:
7: if add > 0 then
8:    for (j = 0 to n) do
9:       for (i = Ap[j] to Ap[j + 1]) do
10:         y[Ai[i]] += Ax[i] × x[j]
11:      end for
12:   end for
13: else
14:   for (j = 0 to n) do
15:      for (i = Ap[j] to Ap[j + 1]) do
16:         y[Ai[i]] −= Ax[i] × x[j]
17:      end for
18:   end for
19: end if
```

Listing 2 points out the different inputs and output vector **y** of the function. The structure `spmat` contains pointers to the arrays for the CCS format previously introduced. Once again, the dimensions of all the different arrays to use were determined, as specified in Table 4.2.

Listing 2 Inputs and outputs of `sparseMV` function

```
void    nd = sparseMV(
            int* A->jc,     // index of elements in A->pr which start
                            // a column of matrix A, size n+1
            int* A->ir,     // stores the row index of each entry of
                            // matrix A, size nnz
            double* A->pr,  // non-zero numerical values of matrix A,
                            // size nnz
            int n,          // size of matrix A
            int m,          // # of nonzeros below the diagonal of A
            int nnz,        // # of nonzeros on diagonal and upper
                            // triangular part of A
            double* x,      // input vector, size n
            double* y,      // output vector, size m
            int add,        // condition to add or subtract operation,
                            // size 1
            int newVector,  // condition to reset vector y, size 1
        );
};
```

4.3.2. Interfaces

After carefully selecting the functions to implement as IP cores and understanding their operations, a choice of which type of interfaces to use is made. The FPGA interacts with the PS through the AXI Protocol and a set of ports, as explained in Section 3.2.2. In general scenarios, the interfaces of an IP core are split into two components: control and data signals. Control signals refer to typical IP core commands such as `start`, `pause`, `resume`, `reset` and others. Data signals are responsible for the necessary exchange of data between the PL and PS for the IP core operation.

Selection

Regarding control signals, the AXI4-Lite is the recommended choice since it is a lightweight and simple interface to implement that, once used in Vitis HLS, produces an associated set of C driver files when exporting the RTL. These files include a set of straightforward functions and structures useful in the Vitis SW control of the IP core actions.

As for data signals, considering the available choices, AXI4 or AXI4-Stream, the second

interface type is selected. While both interfaces present data burst capability, the AXI4-Stream implementation requires less configuration on the Vitis HLS side, which, however, introduces the setup of an additional IP core, as discussed in Section 4.3.4.

Data Structures Manipulation

Referencing to Section 4.3.1, a simplification of the input and output data structures was required as, in their original form, they contain pointers to pointers. This construct is not supported, as top-level interfaces, by Vitis HLS [43]. The inputs and outputs of the IP cores have to be simple pointers to arrays. Consequently, the main structure should be divided into separate arrays. However, this could lead to an excessive increase in the number of pointers (and therefore buffer memory) involved in the PS-PL transfers. Furthermore, Vitis HLS advises the avoidance of pointers in cases where multiple accesses (read or write) are done [28].

Therefore, the data exchange between the PL and PS is done through Transmitters (TXs) and Receivers (RXs) arrays, lowering to a minimum the quantity of inputs and outputs each IP core presents. The exact method and details are described in the following section.

PS-PL Interface Implementation

First, an AXI4-Stream interface is specified through the *Pragma HLS interface* option. It guarantees the interface implementation mimics the AXIS interface style mentioned in Section 3.2.2. Moreover, `Pragmas` also enable the customization of the interface to use. Specifically, the `depth` option was used to specify the maximum number of samples for the test bench to process.

Secondly, each AXI4-Stream interface must have an associated data type. Since the selected functions use integer and double data types arrays as inputs and outputs, a need to create two distinct stream interfaces arose, as exemplified in Figure 4.2.

Thirdly, `stream` objects have to be incorporated into the C code by using the `hls::stream` object definitions. More in detail, this datatype-dependent object encapsulates the requirements of the streaming interface and, importantly, includes a set of useful functionalities. Moreover, `read` and `write` methods allow to sequentially read and write elements from and into the interface. Figure 4.2 presents a basic scheme of the methodology to exchange data between the PS and PL. The way the streams are packed in the PS has to be coherent with how they are unpacked in the PL and vice-versa.

As anticipated, and referring to Figure 4.2, the implementation of the AXI4-Stream re-

quired the creation of four different arrays, two TXs and two RXs, each with an associated data type (step 1). Each TX is sequentially loaded with all the data to pass to the PL (step 2). Inevitably, for each function, this leads the need to find the size of the arrays the pointers, used in the C application, are directing to. After the computation in the FPGA is complete (step 4), the RXs unload the data, coherently, to the respective variables (step 6).

While the discussed workflow is applicable for both functions, `sparseMV` only needs one RX since it only has a single output.

For reasons discussed in Section 4.3.4, when the last element of the transaction is written, the `TLAST` signal, defined in the C code of the IP core, is set to high.

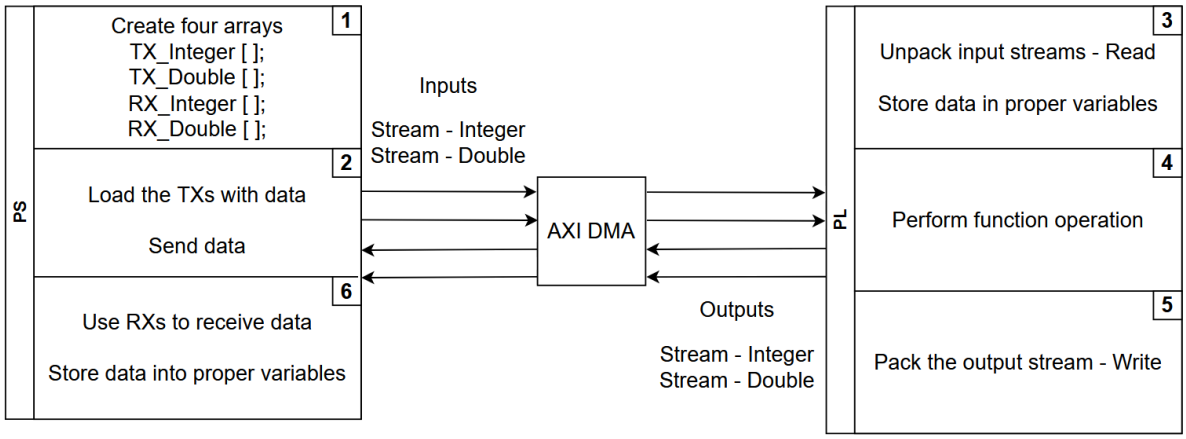


Figure 4.2: Scheme of stream use between PS and PL.

Finally, two AXI HP ports and one AXI GP port were selected, for the data and the control signals, respectively.

The connection between the IP core, the AXI Direct Memory Access (DMA) block and the Central Processing Unit (CPU) was done through the IP Integrator, discussed in Section 3.4.2, in parallel with the *Run Connection Automation* option Vivado grants. Besides providing the right link between the different components, it also creates additional necessary connections for the operation of the whole system.

Section 5.2 presents and explores the integration of each IP core into the system, with Figures 5.1 and 5.2 exhibiting the respective diagrams.

4.3.3. Optimization

Referring to Figure 4.2, each one of the IP cores developed is characterized by the same high-level operational flow:

1. Read from input streams and store;
2. Perform computations;
3. Write to the output stream.

To obtain a higher throughput, the optimization techniques detailed in Section 3.4.1 were used. Therefore steps 1 and 3 were pipelined, using the *Pragma HLS pipeline*, to reduce its II to the minimum. Hence their latency, in terms of cycles, was reduced to the respective sizes of inputs and outputs to work with plus two additional cycles. These arise from the faded operations present at the beginning and each of each `for loop` to optimize. Figure 4.3 exemplifies this condition with an input stream with 3 elements, summing up to a total of 5 cycles, where `READ` and `STORE` are processes.

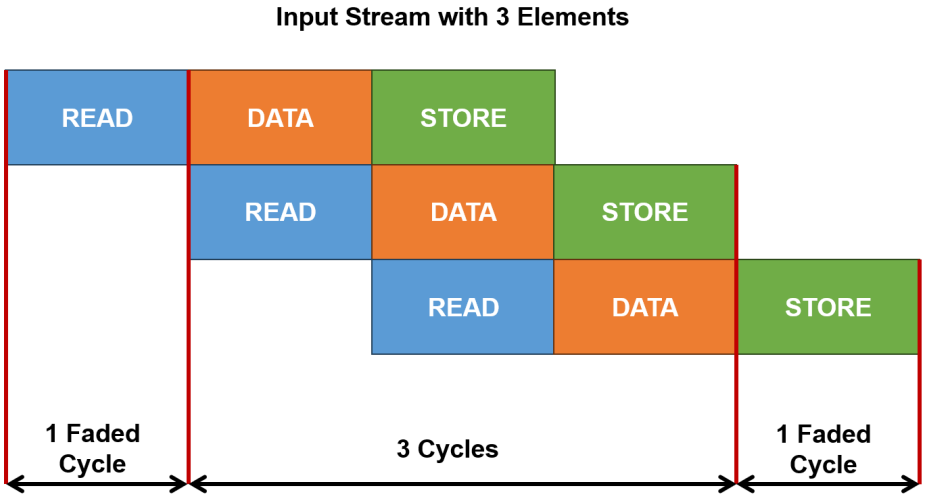


Figure 4.3: Example with input stream to demonstrate faded behavior.

Within step 2, parallelization was explored but not implemented due to the presence of nondeterministic loops and nested data dependencies.

In fact, both functions revolve around multiplication loops which vary considering the number of non-zero values a matrix row presents. This is a typical workflow explored by compact matrix storage methods such as CCS [55]. As an example, let's consider the matrix $\mathbf{A}$ and its CCS format shown in equation (4.6), an input vector $\mathbf{x} \in \mathbf{R^4}$ and an output vector $\mathbf{y} \in \mathbf{R^4}$.

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 0 \\ 4 & 5 & 0 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\ \mathbf{A_x} &= [1, 4, 2, 1, 5, 3, 4, 5], \\ \mathbf{A_i} &= [0, 2, 0, 1, 2, 0, 0, 2], \\ \mathbf{A_p} &= [0, 2, 5, 6, 8]. \end{aligned} \tag{4.6}$$

Reasoning in the number of iterations for each multiplication:

- SW-oriented application: the Pseudo-algorithm 4.4 presents a process where, thanks to the CCS format, the multiplication loop only utilizes the non-zero values stored in vector $\mathbf{A_x}$. Therefore, the `for loop` of line 4 is characterized, for each outer iteration j, by 2, 3, 1, and 1 iterations, respectively, summing up for a total of 8 (the number of non-zero entries).

Algorithm 4.4 Software-oriented matrix-vector multiplication example

```
1: Inputs : Matrix A, vectors x and y
2:
3: for (j = 0 to 4) do
4:    for (i = Ap[j] to Ap[j + 1]) do
5:       y[Ai[i]] = Ax[i] × x[j]
6:    end for
7: end for
```

- Hardware-oriented application: a standard matrix-vector multiplication algorithm is considered where all values of the matrix are used, both zero and non-zero. Consequently, an n-by-n matrix multiplied by an n-by-1 vector leads to a `for loop` composed of n-by-n iterations. For the presented example, this makes up to 16 iterations. Looking at Figure 4.4, in the left case, an arbitrary hardware design is created to contain the 16 iterations. However, as exemplified in the right case of Figure 4.4, parallelization can be explored through a *Pragma HLS unroll* optimization of factor 4, resulting in distinct hardware blocks, each responsible for a row. This process is possible as each row-wise operation is data-independent and deterministic.

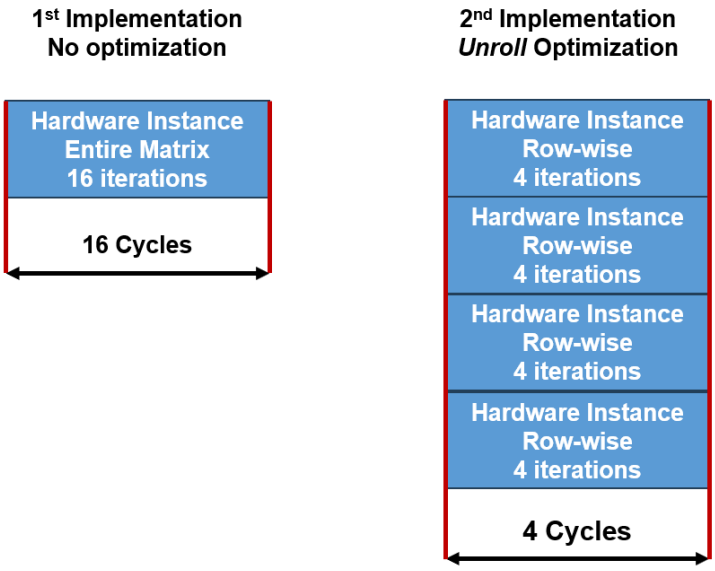


Figure 4.4: Example for matrix-multiplication unroll.

The software-oriented application presents 8 iterations while the first implementation in the hardware 16. However, as data-independent and deterministic operations are used, a parallelization process can be done to combine multiple concurrent hardware instances which, when combined, reduce the number of cycles from 16 to 4.

Nevertheless, when trying to implement this framework to the deployment of the Pseudo-algorithm 4.4 in hardware, a problem arise in the `for loop` of line 4. At each outer iteration j, depending on the number and disposition of non-zero entries the matrix presents, the bounds of the loop change. Therefore, Vitis HLS creates a single arbitrary hardware design which, to accommodate all operations, inevitably leads to a sequential workflow (not exploring the concurrency capabilities).

To exemplify, using the matrix in equation (4.6), the bounds of the `for loop` of line 4 of the Pseudo-algorithm 4.4 are 2 and 3 when j=0 and j=1, respectively. Vitis HLS creates a single arbitrary hardware design to support the 2 iterations of j=0. Consequently, when j=1, the hardware is not suited for the application. The hardware instance is called to perform the first 2 iterations which, once done, is called again to perform the final 3rd iteration.

This non-deterministic framework makes it not possible to explore the concurrency advantages that FPGAs present.

Additionally, if the computation of an iteration of the loop is dependent on previous results, the concurrency and pipeline of the function are not possible. This is observed in

the Pseudo-algorithm 4.2 where the computation of the kth row of L is processed after the previous L-1 rows are set.

To conclude, cases of variable-bound loops with data dependencies affect the possible loop unrolls or pipeline. As exemplified, a single arbitrary hardware instance is created such that, to accommodate all operations, inefficient code sequences are performed. Therefore, the concurrency advantage FPGAs present is not explored, leading to worse hardware implementations.

Both functions under analysis have nondeterministic and data-dependent behaviors. Therefore, problems in the synthesis report of each IP core are noticed, as discussed in Section 5.1.

Nonetheless, for the appropriate code sections of each, the *Pragma HLS unroll* was used to allow certain operations to run simultaneously. Pipelining was also applied when applicable.

4.3.4. AXI DMA

For this work, the AXI DMA IP block, shown in Figure 4.5, was used to transfer data from DDR memory to the IP block and backward [56]. This is a mandatory block to use when AXI4-Stream interfaces are implemented. Figure 4.6 depicts a simple design scheme to highlight the role of this block, where in the place of the FIFO would be the IP core to deploy. The benefit when using this block is the facilitated scheduling of memory transactions, handling of address generation and burst formatting it grants, providing a correct link between the PS and PL.

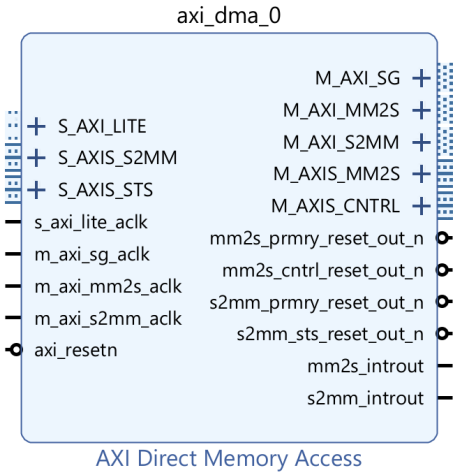


Figure 4.5: AXI DMA IP Core [56].

It presents two operational modes: Simple and Scatter-Gather. For this work, the Simple mode was used since, while fulfilling the requirements, presented an easier and lower-cost implementation in terms of resources. The Scatter-Gather mode offers capacity for offload data movement tasks from the CPU through an extra AXI bus, not presented in the scheme. More information about this mode can be found in [56].

Back to the Simple mode, keeping as reference Figure 4.6, the PS-PL connection is implemented by a set of AXI4 interfaces. The AXI-lite bus allows the processor to control the AXI DMA actions (such as initiate, setup and monitor). Then, the AXI_MM2S (Memory-Mapped to Streaming) reads data from external memory to the IP core while the AXI_S2MM (Streaming to Memory-Mapped) transports data in the opposite direction. This is a simplified overview of the entire cycle of operations and signals, with complete details available at [56]. These last two channels enable a continuous data stream between the PS and PL.

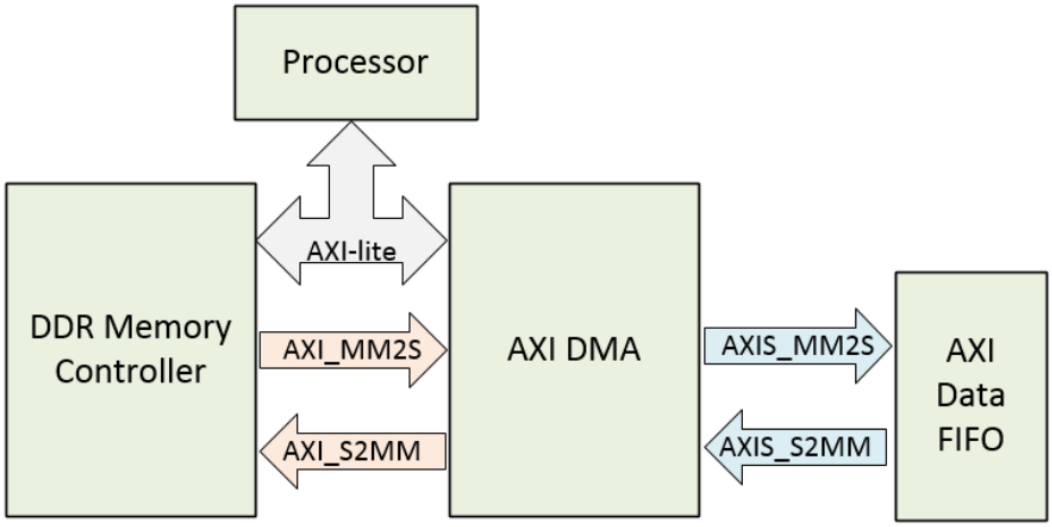


Figure 4.6: AXI DMA block scheme design example [1].

As referred, the DMA block expects any streaming IP connected to the write channel (AXI_S2MM) to set the AXI `TLAST` signal when the transaction is complete. If not properly set by the IP core, the block never completes the transfer, leading to application stalling.

[1] `https://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html/` (last accessed: November 1, 2023)

4.4. Board Memory Constraints

The scheme presented in Figure 4.2 points out the need to store the data sent to the PL in local variables. In the FPGA, once local arrays or variables are initialized, they are stored in the BRAM resource introduced in Section 3.1.3.

In this work, when synthesizing the IP core responsible for the `kkt_factor` function, the "C Synthesis" reported a resource usage of 457% of the BRAM concerning the amount available, previously presented in Table 3.3. This was expected due to the high number of nodes considered which, as a result, led to a vast workspace of variables.

Different solutions were considered:

- Replacing the hardware with one with more resources: the board of this work is positioned as a development kit that targets the acquaintance of the main processes and remarks of application development for FPGA devices. Therefore, the transition to a board of higher resources should only be made after the knowledge and methods to deploy the algorithms of interest for this study in the hardware are proven feasible. Consequently, this procedure is not considered;
- Problem partition: sending packets of data and solving the sub-problem associated with each packet, instead of sending all data and solving the entire problem. This can be easily exemplified by thinking of an IP core responsible for image processing. Rather than sending the entire image and processing it, it is split into multiple sub-images which are sent, processed and retrieved sequentially, being put together at the end. Thus, the FPGA only stores each sub-image at each instance as an alternative to storing the entire image, which would require more memory resources. However, this approach requires the original data to be broken down into independent packets which are put back together at the end. This aspect was not trivial for functions `kkt_factor` and `sparseMV`, hence being discarded;
- Problem size reduction: a reduction in the number of nodes to consider results in a cutback of the problem size, being the selected procedure. Explained in detail in the next Section 4.4.1, this approach pairs with the focus of this work which is to learn the main challenges and processes in the deployment of STO algorithms in an FPGA.

4.4.1. Problem Size Reduction

A reduction in the number of nodes to consider leads to a reduction in the workspace of variables to work with. Since less nodes are considered, the state history of the trajectory is reduced. Recalling equations (2.6), it affects the quantity of vectors of natural two-body dynamics $\mathbf{f}$, jacobian matrices $\mathbf{A}$ and changed variable z to compute and store. After some calculations, it was stipulated that the number of nodes that would generate a workspace of variables whose size is compatible with the BRAM resources available by the Zedboard was 5. Yet, simply reducing the number of nodes of the problem to solve from 100 to 5 would lead to unrealistic solutions.

To keep the logic of the problem, it was decided to select a section with 5 nodes from the trajectory solution of the original 100-nodes implementation as the new problem to solve.

Therefore, to set up this new 5-nodes implementation, illustrated in Figure 4.7, the following processes are considered:

1. Solve and retrieve the solution of the Earth to Mars minimum fuel STO problem using 100 nodes. These are the black nodes of Figure 4.7;
2. Extract the first 5 nodes of the solution, the new segment to consider. These are the red dots of Figure 4.7;
3. Set a new problem, considering the set of conditions from the 100-nodes problem, but changing the final position to the retrieved 5th node position. Besides, appropriately adjust the final time instant and set the number of nodes to 5.

Therefore, the set of conditions for the new 5-nodes problem to solve is identical to the ones of equation (4.1), changing the final position to the retrieved 5th node as equation (4.7) highlights and, once again, the number of nodes and final time instant to consider, despite not shown.

$$\mathbf{x}(t_f) = [1.0012;\ 1.7773e^{-1};\ 5.3051e^{-5},\ 1.4287e^{-2};\ 1.0217;\ 1.0343e^{-3}]. \tag{4.7}$$

These are the required stages to properly set up the new problem. The algorithm of the new problem to solve is identical to the Pseudo-algorithm 4.1. It is crucial to point out that the solution that minimizes the fuel transfer for the new 5-node problem has to be the same as the 1st and 5th nodes segment solution of the 100 nodes implementation.

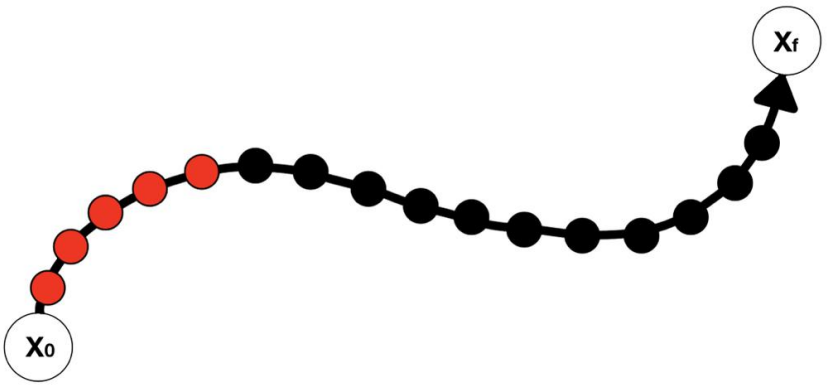


Figure 4.7: Simplified problem reduction approach.

The new solution computed with the reduced number of nodes is presented in Section 5.3 after deployment in the desktop and PS. Despite not being presented, the obtained solution fulfills a trajectory and control profile that coincides with the first 5 nodes segment and control profile of the 100 nodes solution.

From here, all the discussion and concepts introduced in Sections 4.2 and 4.3 are equally applied to the newly defined problem.

Furthermore, a profiling of the new application was performed, highlighting no significant changes with respect to the results previously presented in Figure 4.1. Therefore, the selected functions to integrate the hardware are suitable with the new problem.

As foreseen, a drop in the size of the arrays required to transfer between the PL and the PS is achieved as seen in Tables 4.1 and 4.2. Subsequently, this leads, as shown in Table 5.1, to IP core implementations whose synthesis report presents resource usages within the available capabilities.

Case	Input Stream Integer	Input Stream Double	Output Stream Integer	Output Stream Double
100 nodes	197695	97662	197696	97662
5 nodes	9886	4853	9887	4853

Table 4.1: Interface sizes of `kkt_factor` for both applications.

Case	Input Stream Integer	Input Stream Double	Output Stream Double
100 nodes	18190	19592	1406
5 nodes	805	877	76

Table 4.2: Interface sizes of `sparseMV` for both applications.

4.5. Arbitrary Precision Implementation

Vitis HLS provides fixed-point arbitrary precision data types for C++. They follow the structure shown in Figure 4.8, where W is the word length, I is the number of integer bits and, consequently, B is the number of fractional bits such that $W = I + B$.

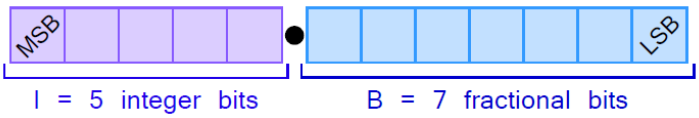


Figure 4.8: Fixed-point representation example [12].

For signed representation, the Most Significant Bit (MSB) bit is allocated while unsigned representation does not require it. For calculations where two numbers with different numbers of bits or precision are used, the binary point is automatically aligned [28].

Additionally, when defining a fixed point data type in Vitis HLS, the user must set the quantization mode Q and overflow mode O to use. The quantization mode is responsible for dictating the design behavior when greater precision to the one the store variable supports is generated. The default mode truncates the number towards minus infinity. The overflow mode, on the other hand, dictates the design behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value supported by the store variable. The default mode wraps the values around in case of overflow [28].

In Vitis HLS, an arbitrary precision fixed-point data type is defined through the syntax `ap_[u]fixed<W, I, Q, O>`, the different fields referring to the previously defined concepts.

Considering the notation defined above, the precision achieved by a fixed-point data type is given by 2^{-B} and the range is 2^{I-1}, for signed implementations. Inversely, and in

typical workflow cases, the precision and range to be defined are known. So, the number of fractional and integer bits that correctly represent the required data type are given by $B = -\log_2(\text{Precision})$ and $I = \log_2(\text{Range}) + 1$, respectively.

For the work here presented, it was decided to implement the `sparseMV` function with both arithmetics to corroborate the expected resource and performance differences. This function was selected as a prototype case for fixed-point sparse library implementation due to the simpler list of operations and integration it presents. Consequently, an easier implementation that still gathers remarks about the effects this approach has on problems such as the ones of this thesis is done.

The function works with the two data types of Table 3.4. However, to compute the required level of precision and range for the arbitrary data type to implement, one iteration of the function was traced to calculate the highest and lowest number present. For the problem under study, the two solution metrics are the minimized fuel mass and objective function. Selecting the fuel mass as the priority, a precision of 10^{-9} was targeted despite inevitable changes in this element being noted, as Section 5.3.2 exhibits.

Afterward, looking at the different sets of operations the function performs, the fixed-point data types presented in Table 4.3 were defined. Considering the function performs matrix-vector multiplication operations, as shown in equation (4.5), special care to double the amount of bits of the output $\mathbf{y}$ was carried. The selected precision also presents two extra decimal places to the one previously stated. This was done for precautionary reasons.

Table 4.4 highlights a second HW design where the precision is increased. The reasoning for this choice is covered in Section 5.3.2.

Type	W	I	B	Range	Precision
integer (input)	11	11	0	1024	1
double (input)	39	5	34	16	$\approx 5.82 \times 10^{-11}$
double (output)	78	10	68	512	$\approx 3.39 \times 10^{-21}$

Table 4.3: Range and precision of selected fixed-point representation for first design implementation.

Type	W	I	B	Range	Precision
integer (input)	11	11	0	1024	1
double (input)	55	5	50	16	$\approx 8.88 \times 10^{-16}$
double (output)	110	10	100	512	$\approx 7.89 \times 10^{-31}$

Table 4.4: Range and precision of selected fixed-point representation for second design implementation.

The type-casting operations between floating and fixed-point is done inside the IP core. The impact of this procedure and the performance difference between the two approaches is presented in Section 5.3.2. Furthermore, the effect the use of fixed-point precision has on the algorithm convergence is also highlighted.

5 | Results

This chapter presents and explores the results of the different design implementations described in Chapter 4. Section 5.1 displays the estimated synthesis reports from the developed IP cores, reasoning over the difference in resource usage. In particular, it also highlights the resources' influence of type-casting operations performed inside the IP core. Section 5.2 exhibits the hardware designs and their actual synthesis reports. In Section 5.3, the solution for the 100 and 5-nodes problems is exposed. Afterward, the results from the different hardware designs are disclosed and explained, going over the performance difference between fixed and floating-point implementations. The impact of fixed-point precision on the algorithm convergence is also uncovered, an aspect related to the range of values ECOS uses and their numerical propagation. Finally, Section 5.4 introduces an outline of the several remarks obtained from the development of this work.

5.1. HLS Synthesis

In Section 4.3 the two functions to implement as IP cores were introduced and detailed. Afterward, a choice of interfaces and workflow for data transfer between PS and PL was described. Table 5.1 features the estimated synthesis results from different configurations. A resource utilization percentage is shown to simplify the overall consumption each IP core retains from the resources available. The presented values are estimations of Vitis HLS and require confirmation once the designs are deployed on the board. This aspect is further discussed in Section 5.2.

Along this Chapter, the resource quantities reported to the `sparseMV` fixed-point implementation relate to the design settings of Table 4.3. Notice, nonetheless, that a higher number of resources is required by the configuration of Table 4.4 naturally.

IP Core	BRAM	DSP	FF	LUT
`kkt_factor` - floating-point	33 (23%)	14 (6%)	3875 (3%)	7677 (14%)
`sparseMV` - floating-point	11 (8%)	14 (6%)	2701 (2%)	3699 (6%)
`sparseMV` - fixed-point	10 (7%)	4 (1%)	3653 (3%)	5618 (11%)

Table 5.1: Estimated synthesis resource utilization for 5-nodes application.

The number of resources between both IP cores differs in some fields. A substantial difference is noted in the BRAM block of `kkt_factor`. This is expected due to the greater workspace of variables it presents.

Regarding the fixed-point implementation of the `sparseMV` IP core, it is seen that the BRAM and DSP quantities are lower concerning the floating-point implementation. The slight BRAM decrease is due to the reduced total number of data bits stored. The DSP reduction is attributed to the more efficient arithmetic operations fixed-point implementations present.

However, the number of FF and LUT increases. This is a consequence of the type-casting operations realized inside the IP core. In fact, by comparing the synthesis reports provided by Vitis HLS for both implementations, it is possible to determine the quantity of resources used for this procedure. By subtracting those from the amounts presented in Table 5.1, only 2099 (4%) LUT and 1633 (1%) FF are obtained. This is a clear reduction to the floating-point implementation.

This method of type-casting inside the IP core does not harm the performance of the fixed-point employment. However, if resource usage is a constraint of the application, type-casting operations should be performed in the PS.

Finally, Table 5.1 does not declare, for each function, the estimated number of latency clock cycles. As discussed in Section 4.3.3, both cases possess nondeterministic sequences such that the boundaries of the loops are data-dependent. Therefore, Vitis HLS can not calculate the required number of cycles for the execution of these sections.

5.2. IP Cores Design Implementation

After exporting the implementations presented in Section 5.1 as RTL, their integration in the system is done through Vivado.

As explained in Sections 5.3.1 and 5.3.2, the performance of each implementation is less

efficient than the respective software-oriented application. For this reason, separate designs for each IP core are created. Otherwise, their combination into a single design would result in an even less efficient outcome.

Their integration in the system is, in either case, quite similar.

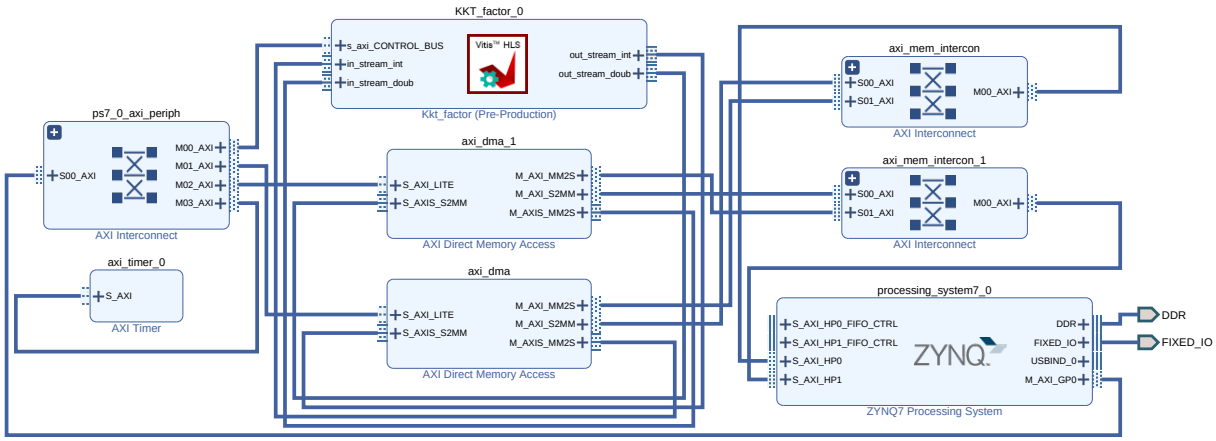
Figure 5.1 highlights the design scheme for the system including the `kkt_factor` IP core. As discussed in Section 4.3.4, the connection between the IP core and PS is through the AXI DMA blocks [56]. A dedicated block for each input and output stream pair is used, therefore presenting both Read (`M_AXIS_MM2S`) and Write (`S_AXIS_S2MM`) channels. Alternatively, an AXI MultiChannel DMA IP (MCDMA) core could be used. The Zynq PS block [57] uses AXI3 Protocol while the DMA blocks use AXI4. Consequently, the AXI Interconnect block [58] is used to automatically meet the necessary conversions.

The stream ports exhibited in Figure 5.1b are connected, through means of previously mentioned blocks, to the HP ports seen in Figure 5.1c. The AXI-lite control port `s_axi_CONTROL_BUS` is connected to the `M_AXI_GP0`.

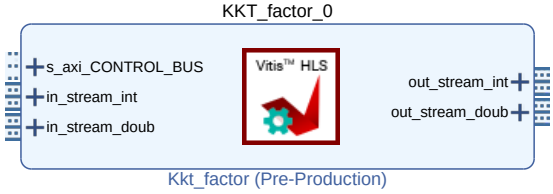
Next, the AXI Timer block [59] is used, with Section 5.2.1 discussing the reasoning behind its inclusion.

In the configuration of the AXI DMA blocks, the width of the buffer length register is maximized to 26 bits. It enables the transfer of 67,108,863 bytes ($\approx$ 63 MB) in a simple transfer, enough for the requirements of this work.

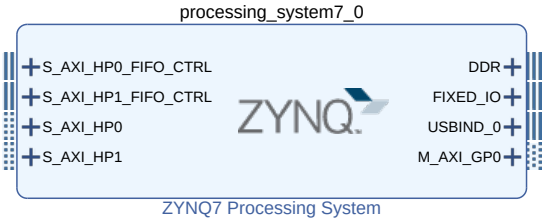
Lastly, the Processor and PL Fabric clocks are set as 666.67 MHz for the CPU, 533.33 for the DDR, and 100 MHz for the PL. The last frequency is coherent with the frequency set in the Vitis HLS development of the IP cores.



(a) Block Design Overview.



(b) `kkt_factor` IP Core.

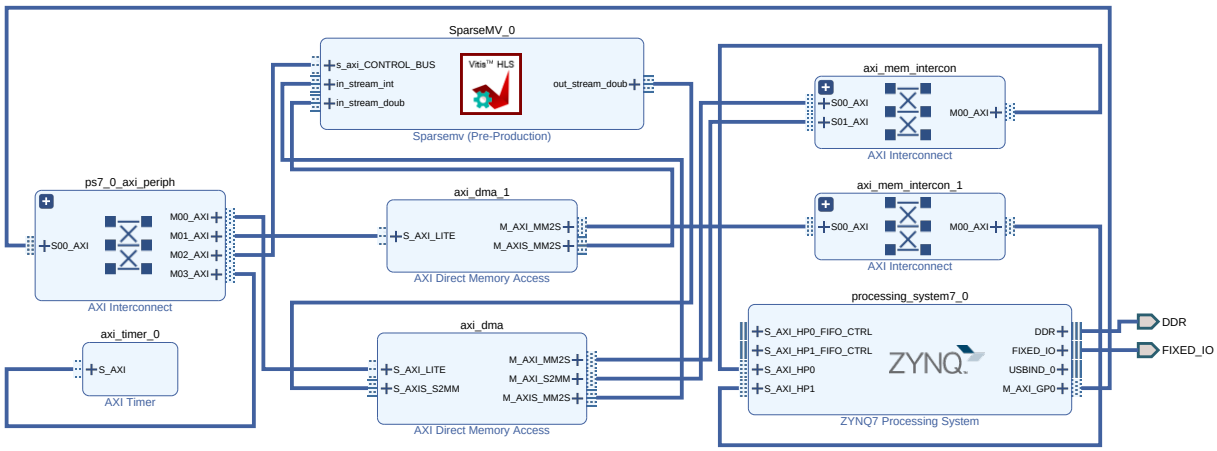


(c) Zynq PS IP Core.

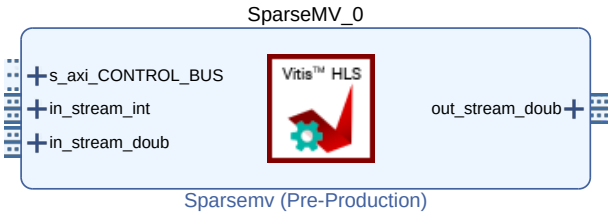
Figure 5.1: IP Core `kkt_factor` design integration.

Figure 5.2 shows the design scheme for the system including the `sparseMV` IP core. The selection, specification, and connection of the different blocks is identical to the previous discussion. Indeed, the Zynq PS block is configured in the same way as the previous case, therefore not being enlarged in Figure 5.2.

A difference is noted in the `axi_dma_1` block of Figure 5.2 which only presents a Read Channel. As the IP core presents two input and one output streams, this specific DMA block simply reads from DDR3. Coherently, the other presents both Read and Write channels for the remaining streams.



(a) Block Design Overview.



(b) `sparseMV` IP Core.

Figure 5.2: IP Core `sparseMV` design integration.

After the connections are set, the synthesis and implementation steps are executed. Therefore, the bitstream is generated. Vivado provides a full report with an overview of the design resources used. An estimation of the total on-chip power is also given. Table 5.2 presents the respective values.

IP Core	BRAM	DSP	FF	LUT	On-Chip Power [W]
`kkt_factor` - floating-point	48 (34%)	14 (6%)	18021 (17%)	12941 (24%)	1.888
`sparseMV` - floating-point	20 (14%)	14 (6%)	7299 (7%)	6202 (12%)	1.753
`sparseMV` - fixed-point	19 (13%)	4 (2%)	8446 (8%)	8311 (16%)	1.751

Table 5.2: Synthesis overview report for 5-nodes application.

As aforementioned, despite Table 5.1 revealing an estimation of the resources each IP core implementation would take, Table 5.2 presents the actual resources used in the implementation of the whole design. As other blocks are included, an increase in most resource quantities is verified for all cases, the only exception being the DSP.

The design implementation with the greatest amount of total resources produces the

highest projected on-chip power consumption.

Lastly, the placement and routing of each design circuit on the FPGA is presented in Appendix A.0.3.

5.2.1. AXI Timer

The design implementations discussed in Section 5.2 present the AXI Timer IP core [59]. Being integrated into the Vivado IP Catalog, it is a 32 or 64-bit timer module that interfaces to the AXI4-Lite interface. Along with other features, it supports two programmable interval timers with event generation and capture capabilities. Figure 5.3 highlights the block.

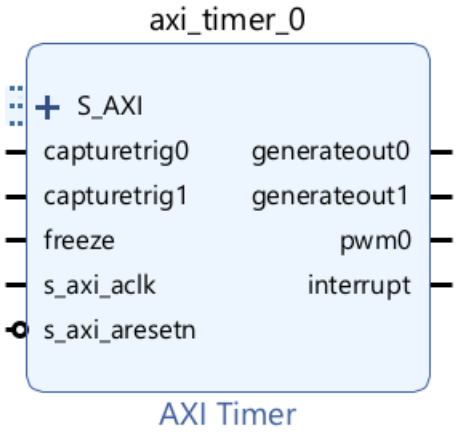


Figure 5.3: AXI Timer IP Core [59].

Its introduction is to verify the performance of the different hardware designs in terms of number of cycles. This is a common metric in the assessment of FPGAs. This workflow is similar to a reference design file provided by Xilinx [60] where the same technique is used.

Thanks to this block, the performance of the IP core is compared to the performance of its software (SW) version, a matter discussed in Section 5.3.

5.3. Results Evaluation

The solution to the 5 and 100 nodes problems is presented in Tables 5.3 and 5.4 after running the C application in the desktop and PS Zynq. The corresponding control vector solution of the problem is also retrieved, despite not being shown here since it is not relevant to this work's overall intent.

Device	J (-)	Fuel Mass (kg)	Iterations k (-)
Desktop	1.6612199448795926	128.006623	3
PS of Zynq	1.6612199448795926	128.006623	3

Table 5.3: Earth-Mars 100-nodes problem solution.

Device	J (-)	Fuel Mass (kg)	Iterations k (-)
Desktop	0.1778736542714191	15.064871	2
Zynq PS	0.1778736542714191	15.064871	2

Table 5.4: Reduced 5-nodes problem solution.

As expected, both cases show no difference in the results of the application, for the selected decimal precision, once run in the desktop or the Zynq PS. While not the target of this thesis, an element to consider is the increased computational time the application takes to run on the PS when contrasted to the desktop. Specifically more noticeable in the 100-node application, the desktop version takes 0.89 s while the PS of the Zynq takes 33.54 s to solve the problem. This is expected due to the computational power discrepancy a desktop workstation offers when compared to the device under use.

These designs, however, do not employ any FPGA capabilities yet. In fact, for reasons previously explained in Section 4.4.1, only the 5-node application explores hardware designs, therefore being the treated case in the upcoming sections.

5.3.1. Numeric Factorization IP Core

After generating the bitstream of each design, the Vitis software code is developed to program the FPGA.

The goal is to solve the 5-node problem presented in Section 4.4.1 using the `kkt_factor` IP core to solve the factorization process presented in Section 4.3.1.

The correct integration of the developed HW logic in the code is eased thanks to the C driver files generated after including the AXI4-Lite interface mentioned in Section 4.3.2. The IP core and both AXI DMA blocks have to be set up. While these details are not presented in this thesis, the methods used are similar to the ones exhibited in [60].

To understand the performance and details of the implemented design, two distinct versions are used. To ease the discussion, they are defined as:

- SW version: the application which does not use any FPGA resource. This is the original C application without any modifications to the code, whose results are shown in Table 5.4;
- HW version: the application which explores a combination of SW and HW routines by employing the `kkt_factor` IP core inside ECOS.

First, Table 5.5 displays the problem solution for the HW version, exhibiting, for convenience, the reference SW version result of Table 5.4. Up to the displayed precision, it is seen that the same results are achieved. This is expected since the IP core was designed with the IEEE standard double floating-point precision, avoiding any loss of range or precision in the FPGA.

Version	J (-)	Fuel Mass (kg)	Iterations k (-)
HW `kkt_factor`	0.1778736542714191	15.064871	2
SW (reference)	0.1778736542714191	15.064871	2

Table 5.5: Reduced 5-nodes problem solution for HW version of `kkt_factor` design.

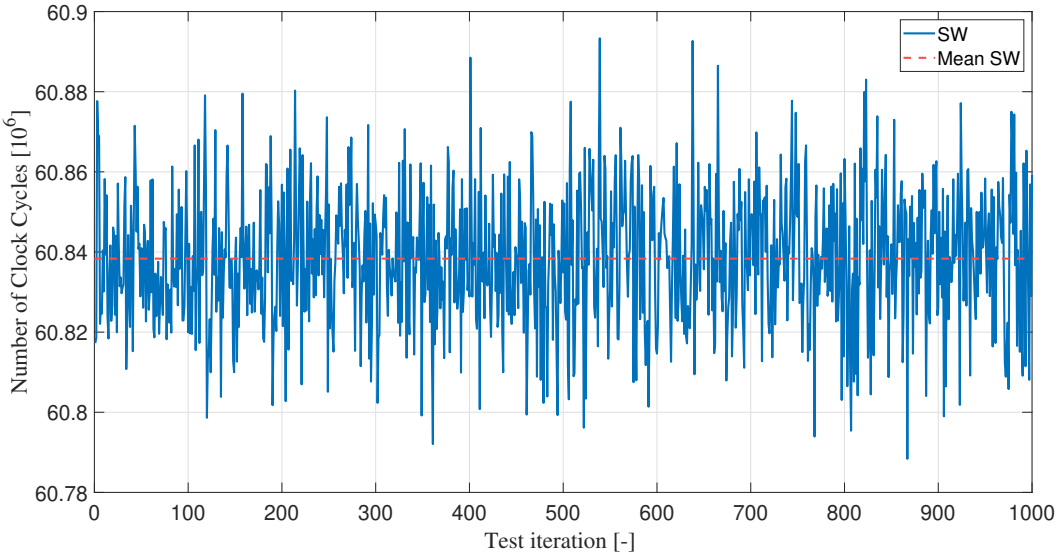
Secondly, to understand the FPGA performance of this design, a set of 1000 test runs for each version is done. This reduces variability bias and provides more reliable metrics. For each run, a reset of the problem is done and the number of cycles to solve it is acquainted. The AXI Timer is used with the default frequency of 125 MHz. Figure 5.4a highlights the SW version while Figure 5.4b the HW version. For meaningful results, both versions are run in the "Release" build.

Referring to Figure 5.4a, the mean number of clock cycles the SW version takes to solve the problem is 6.08383×10^7. Contrarily, looking at Figure 5.4b, the mean number of clock cycles the HW version takes to solve the problem is 1.70228×10^8. The HW version presents a higher number of cycles.

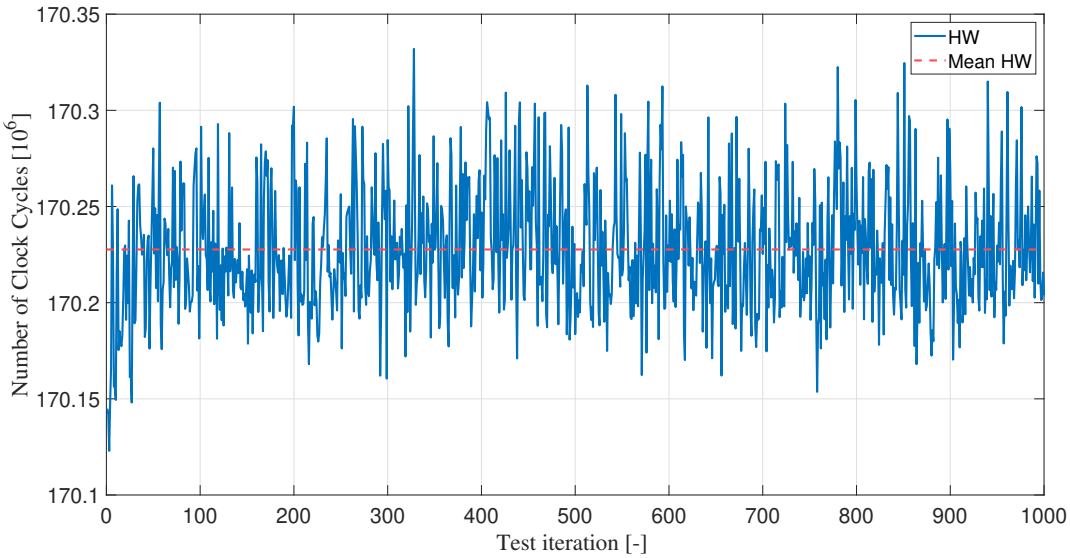
In fact, for this work, the Acceleration Factor (AF) is defined in equation (5.1) as a metric to compare the performance between the two versions.

$$AF = \frac{\text{Number of Clock Cycles for SW version}}{\text{Number of Clock Cycles for HW version}}. \tag{5.1}$$

Figure 5.5 presents the Cumulative Distribution Function (CDF) of the acceleration factor for this design. Its mean value is 0.357.



(a) Number of cycles to solve the problem using only SW functions.



(b) Number of cycles to solve the problem combining SW and HW functions.

Figure 5.4: Performance of HW and SW versions for the `kkt_factor` implementation.

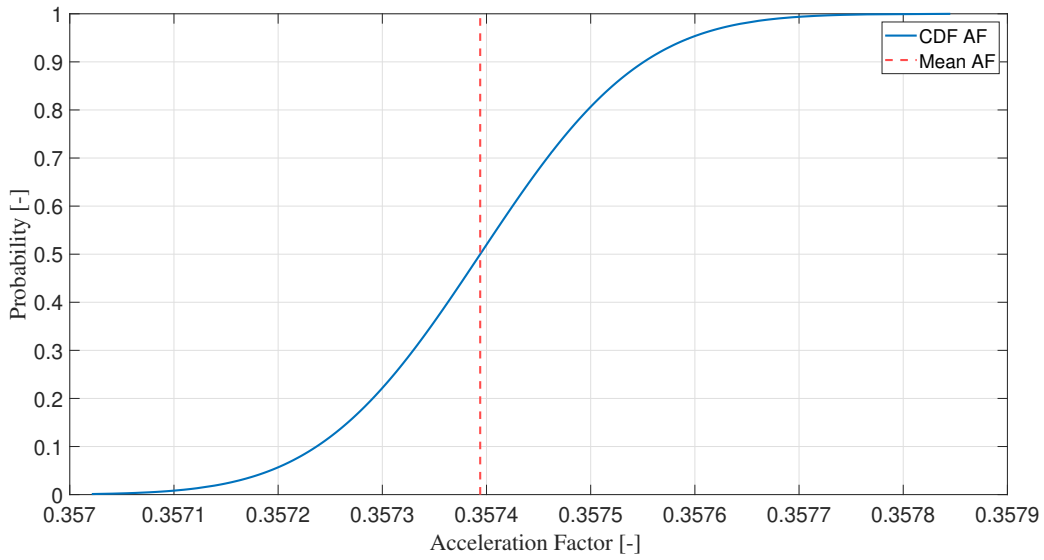


Figure 5.5: Cumulative distribution function of the acceleration factor for the `kkt_factor` design.

From the results presented, the HW version is, approximately, three times slower than the SW version. This outcome can be explained through the following reasons.

First, the number of cycles needed for the transfer of data, the address of the data to the arrays, and the read and write operations from the streams are computed. All these factors account for a total of 326494 clock cycles per use of the IP core. However, while solving the problem, it is seen that `kkt_factor` is used 202 times. Combining these two aspects, a total of 65951788 clock cycles is obtained. Subtracting it from the mean value of clock cycles the HW takes, a new acceleration factor of 0.583 is attained.

Yet, calculating the number of cycles solely the transfer takes in a single cycle, the value 902 is achieved. Compared to the 326494 clock cycles presented above, it is negligible. Therefore, it is concluded that a significant drop in the HW performance is caused by the address of data and stream management. Referring to Figure 4.2 of Section 4.3.2, these correspond to steps 2, 3, 5, and 6 of the scheme.

Even after the motives detailed above, an acceleration factor smaller than one is obtained. Consequently, this means, for the same conditions, the LDL^T factorization shown in the Pseudo-algorithm 4.2 is more efficient in SW than the developed HW. This is a result of factors such as:

- Use of floating-point precision: this results in less efficient performances, as referred in Section 3.5.

- Presence of nondeterministic and data-dependent loops: two effects which deprive the use of pipeline or unroll optimizations, as described in Section 4.3.3. Inherently, less efficient designs are obtained since concurrency capabilities are not explored.
- Frequency of the IP core: as stated in Section 4.2, the CPU is running at a much higher frequency than the PL. One additional technique that can lead to more efficient performances is the increase of the PL frequency. This increase must be within the values that guarantee the hardware design timing constraints.

Notice, however, that increasing the IP core frequency is limited to the hardware timing constraints resulting from the number of concurrency operations applied. The user can target a faster clock with non-deterministic loops or a slower clock with deterministic parallelizable loops to possibly achieve faster designs. Normally, the combination of both is impractical.

Discussion

Despite, in the molds of this work, the matrix factorization process performing worse in the FPGA, efficient throughput implementations are found in [61, 62]. These exploit block partitions of matrices **A**, **D**, and **L** to enable matrix-matrix multiplications. Afterward, concurrent operations are done such that, in the end, all matrices are assembled back. Often called "right-looking" variants, these techniques enable better results.

Having said that, in [61] is proved that as the size of the matrix increases, the speed-up factor decreases, despite being always favorable in the FPGA case. This behavior is associated with the upper bound limit the performance of Cholesky factorizations display as the problem size increases [63]. This limit could be a trade-off to keep in mind for future implementations depending on the desired level of performance increase.

Alternatively, other efficient matrix factorization processes for FPGA implementations can be explored. In [64], to achieve high throughput, a fixed-point LL^T factorization process is used to parallelize the diverse tasks. Yet, by changing the factorization method, significant changes to ECOS would follow, an important factor to take into account.

5.3.2. Matrix-Vector Multiplication IP Core

The programming of the FPGA for this case is similar to the one previously discussed in Section 5.3.1.

This time, the problem presented in Section 4.4.1 is solved using the `sparseMV` IP core to perform the required matrix-vector multiplications. While also discussed, the intent of

this implementation is not to strive for better performance efficiencies. It is, instead, to understand the impact a fixed-point arithmetic can have in problems such as the one in this work.

Once more, to ease the analysis, the terms "SW version" and "HW version" are used. They are defined similarly as in Section 5.3.1, the only difference being that the HW version refers, in this case, to the application that employs the `sparseMV` IP core.

As specified in Section 5.1, two distinct implementations of `sparseMV` are developed. The results from each are analyzed separately below.

Floating-Point Precision

The solution of the HW version achieves the same results seen in Table 5.5. Once again, as this IP core design uses double floating-point precision, no difference between the SW and HW solutions is noted.

As in the previous case, a set of 1000 test runs is done. Figure 5.6 poses a mean number of 8.43487×10^7 clock cycles for the HW version. Compared to the SW value presented in Section 5.3.1, the HW version presents a higher number of clock cycles, resulting in an acceleration factor whose mean value is 0.721.

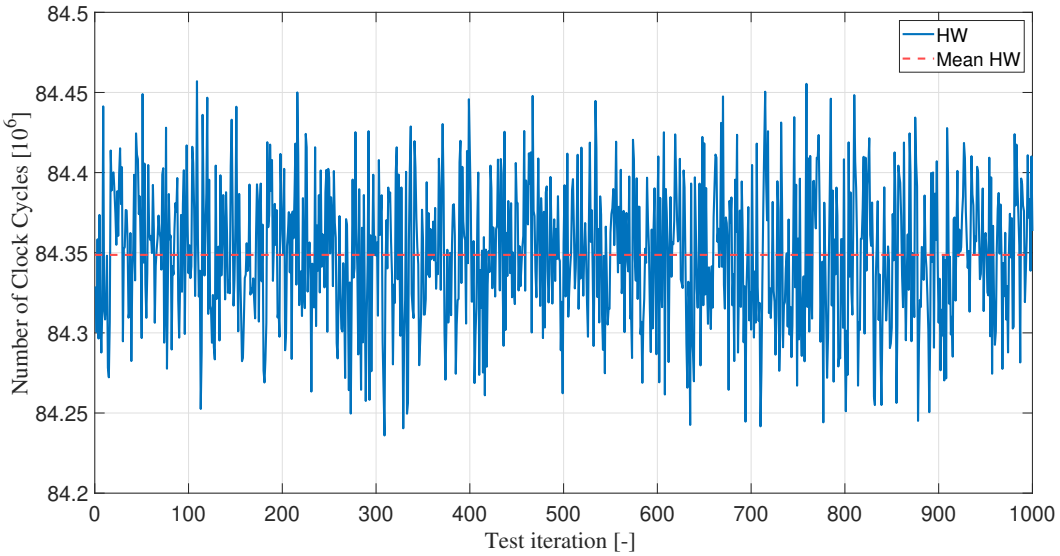


Figure 5.6: Performance of HW version for the `sparseMV` floating-point implementation.

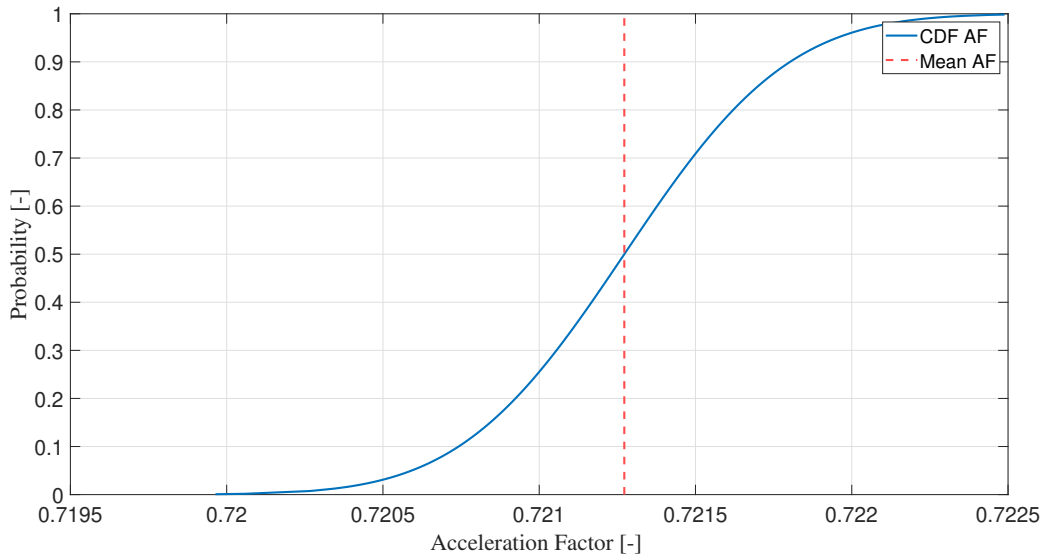


Figure 5.7: Cumulative distribution function of the acceleration factor for the `sparseMV` floating-point design.

Analyzing the data, the HW version is, approximately, 1.4 times slower than the SW version. While still a negative result, it is an improvement with respect to the previous case. In fact, the operations and number of nondeterministic loops each IP core presents are radically different, impacting each implementation diversely.

For this case, 6673 clock cycles per use are needed for data transfer, address of the data, and read and write operations for the streams. As the function is used 1806 times in a single run, a total of 12051438 clock cycles is obtained. Subtracting it from the mean value of clock cycles the HW takes, a new acceleration factor of 0.842 is calculated. Compared to the previous case, this presents a smaller relative performance increase. This is expected as the workspace of variables this IP core uses is greatly reduced as Tables 4.1 and 4.2 highlight. Consequently, the impact operations such as data address and stream management take in the implementation is reduced when compared to the previous case.

Similarly to the previous scenario, the data transfer only accounts for 340 clock cycles, making for a negligible impact on the overall implementation.

These two IP core implementations corroborate the burst transfer of data benefit the AXI4-Stream presents. In either example, the address of data and stream management are relevant factors that greatly impact the performance.

As in the previous case, the poor performance the HW version achieves can be attributed to the presence of nondeterministic loops and the use of floating-point precision. These

two factors lead to less efficient hardware designs. Additionally, once again, an increase in the IP core frequency, within values that respect the timing constraints, can also grant performance increments.

Discussion

Generally, to achieve high throughput in sparse matrix-vector operations in FPGA designs, data structure organization, and matrix partition into blocks are focal points to target concurrency of operations. These require information on the sparsity structure of the input matrix. For matrices with very irregular sparsity structures, very few improvements are accomplished.

However, in [65], a design is developed to perform such operations without any information on the sparsity structure of the matrix. For it, a Compressed Row Storage (CRS) matrix storage format is used. This enables the parallel multiplication of each row of the matrix with the vector, exploring concurrency for efficient performances. Nonetheless, this approach would require, once more, an extensive number of changes to the ECOS workflow. Instead, the focus for future implementations could be on other operations such as Section 5.4 reports.

Fixed-Point Precision

Table 5.6 presents the solution to the problem for the fixed-point HW version using the two configurations of Tables 4.3 and 4.4.

Regarding the first implemented design, it is possible to see a loss of precision in the obtained results for both the objective function and fuel mass. When selecting a precision of 10^{-9} from an isolated iteration of the whole algorithm, all the decimal places that are under this precision are lost. In other words, for the sample of data collected, while the values above the selected precision are covered, a loss in decimal places is, regardless, undergone. This impacts subsequent computations, resulting in an overall loss of precision.

In fact, despite the fixed-point IP core itself presenting higher efficiency, as proved below by a test case in Section 5.3.2, its implementation in the whole design leads to a lower acceleration factor concerning the reference floating-point implementation. This is strictly related to the higher number of iterations required to reach the problem solution. ECOS entails more iterations to solve the problem, resulting in a higher computational time when compared to the floating-point implementation.

This effect can be seen in the increased number of calls for the function under study.

Indeed, the 3021 calls taken are a significant rise from the floating-point case (1806). Additionally, while the major factor for the performance decrease is the higher number of iterations the IPM requires, the extra number of cycles the HW takes is also augmented throughout the resolution of the problem, intensifying the performance decrease.

To express this hypothesis, a second fixed-point implementation with higher precision is considered. A difference in the objective function, fuel mass, and number of iterations to attain the solution is still displayed. The fuel mass of this design in specific is closer to the reference value when compared to the first fixed-point implementation. Additionally, the number of iterations demanded by the ECOS solver is lower, as seen by the number of `sparseMV` function calls. Indeed, as expected, the acceleration factor improves when compared to the previous case. Still, when compared to the floating-point implementation, the less efficient performance can be attributed to the augmented number of calls.

Design	J (-)	Fuel Mass (kg)	Iterations k (-)	Number of calls for `sparseMV` (-)	Acceleration Factor (-)
HW `sparseMV` - fixed-point (10^{-11} precision)	0.1778841011968801	15.064836	3	3021	0.526
HW `sparseMV` - fixed-point (10^{-16} precision)	0.1778532160822693	15.064839	3	2585	0.589
SW (reference)	0.1778736542714191	15.064871	2	1806	-

Table 5.6: Solution, performance and details about the fixed-point implementations of `sparseMV` designs compared to the SW version.

These implementations illustrate the challenge the precision selection of fixed-point arithmetics can have in problems such as those of this work. A strict need for keeping all the significant decimal places is mandatory to achieve more efficient performances.

Indeed, from [53], the author states a need for single floating-point precision to cover the dynamic range of numbers arising from the IPM. The fixed-point precision is, accordingly, unadvised for problems that use ECOS. This is a common concern in FPGA implementations of IPMs [66]. Actually, a fixed-point configuration that entails the range and precision of a single-floating point could be done. It would, however, consume a high number of resources, therefore not being a viable option.

Performance Comparison

To demonstrate the performance impact fixed-point arithmetics can have over floating-point, a simple test case is built. This utilizes the example sparse symmetric positive-

definite matrix **A** defined in [52].

The floating-point IP core presented in Section 5.3.2 is compared to the first fixed-point IP core design whose settings are defined in Table 4.3. Table 5.7 shows the mean value of each design over 1000 runs, considering also the sole `sparseMV` function in SW.

Design	**Number of clock cycles (-)**
SW `sparseMV`	1263
HW `sparseMV` **- floating-point**	11181
HW `sparseMV` **- fixed-point** (10^{-11} **precision)**	5680

Table 5.7: Performance comparison between floating-point and fixed-point `sparseMV` IP core and its software routine.

The fixed-point implementation presents, almost, half of the number of cycles required to execute the same operations in floating-point. Consider, although not shown, that the results from the fixed-point design only comprehend up to 10^{-11} when it comes to precision.

An advice when developing an IP core using both arithmetics is the latency in number of cycles that the Co-Simulation report in Vitis HLS stipulates. This can be used as a preliminary estimation for the discrepancy both designs might present.

Additionally, the results of Table 5.7 highlight the efficient operations the function of the original code (SW version) achieves when comparing the number of cycles. In fact, considering the workspace of variables of Table 4.2, at least, 1758 cycles are required to write and read the data from the streams passed to the IP core. This value alone is larger than the number of cycles needed for the SW to perform all the computations, highlighting the gap between the SW and HW implementation of each function in this work.

An approach that could diminish this disparity is the design of an IP core which includes multiple different functions, all accessed sequentially. This would attenuate the impact that the transfer of data, data address, and stream management have in the sole implementation of functions such as `kkt_factor` and `sparseMV`.

5.4. Remarks

After presenting and discussing the results and discoveries from the different implementations, a synopsis is provided in this section. However, the following considerations are made:

- IP cores selection: in Section 4.2.1, the selection method introduced in this work is described. A key component deriving from the results is the recommendation of opting for functions that present deterministic and data-independent characteristics such that parallelism and pipeline operations can be explored. However, in [53], a theoretical model for Model Predictive Control (MPC) problems demonstrates the advantages of carrying out matrix-matrix multiplications and matrix forward substitution in an FPGA. This observation can be seen in Figure 5.8, where the right columns correspond to the microcontroller core and the left columns to the FPGA acceleration for matrix-matrix multiplications and for matrix forward substitution. Data transfers are not accounted. An interesting consideration for the development of future applications that explore IPMs could be the combination of the conclusions presented in Table 5.8 with IP core designs for the recommended operations.
- Size of the Problem: due to the characteristics of the IPM, larger speedups can be expected for larger problems [53].

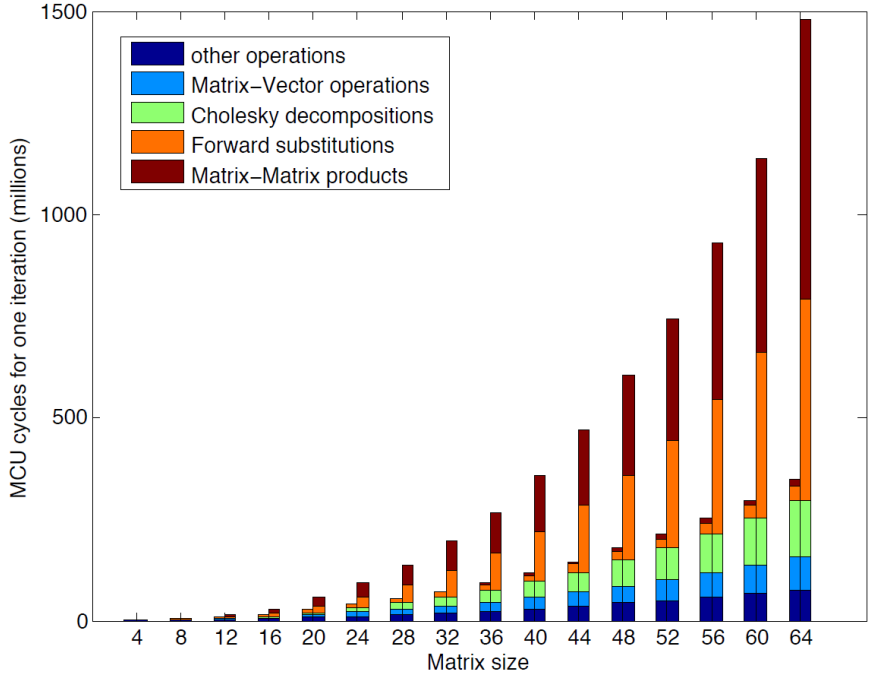


Figure 5.8: Predictions of computation times for PS for large MPC based on a theoretical model [53].

This said, Table 5.8 reports the main remarks to keep in mind for the FPGA deployment of future STO problems that explore interior point solvers (such as ECOS). While some aspects are strictly related to the techniques used in this work, others can be applied to different onboard trajectory optimization guidance algorithms as they are general considerations of FPGA scenarios.

Remark	Overview
Data Transfer	Using AXI4-Stream, the impact the transfer of data has on the IP core performance is minimal thanks to burst capabilities.
Data Address and Stream Management	Considering the implementation workflow presented in Section 4.3.2, the impact these two aspects have on the performance efficiency can be significant. This depends on the size of the workspace of variables to use. A bigger workspace of variables requires more data address and stream management operations, leading to less efficient designs.
Refined IP Core Selection (Performance Prioritization)	After profiling the application, the functions to convert into IP cores shall be the ones with the highest exclusive percentage characterized by deterministic and data-independent behaviors. Otherwise, if these aspects are not present, a change in the function structure and implementation must be done to reach such state. This enables pipeline and unroll optimizations. Additionally, to mitigate the effects of the previous two remarks, an HW design shall include multiple functions with high exclusive percentages instead of creating multiple HW designs for each. Lastly, the sections of IPMs that fit the most for FPGA implementations should be regular, but expensive, operations such as matrix-matrix multiplication or matrix forward substitution [53].
Floating-point and Fixed-point Precision	Fixed-point precision presents advantages from a performance and resource usage point of view. However, for onboard guidance algorithms that utilize interior point solvers such as ECOS, the data type of related IP cores shall be, at least, single-floating point precision. This invalidates the use of fixed-point arithmetic for such applications.

Type-casting	For applications that explore fixed-point precision, if resource usage is a constraint of the development, type-casting operations shall be performed in the PS.
Performance Optimization	Pipeline and unroll optimizations should be applied to achieve more efficient designs. Alternatively, the frequency of the design should be increased up to the maximum value which still fulfills the hardware design timing constraints.
Device Resources, Problem Size and Scale in Performance	On one side, onboard guidance algorithms that utilize a high number of nodes have, subsequently, a large workspace of variables. This is a concern and limitation for FPGA implementations that present limited resources (such as BRAM for example). On the other side, for STO problems employing IPMs, the performance gap between the FPGA and SW implementations should scale with the problem size [53].
SW Oriented Applications	Applications targeted for SW environments might exhibit characteristics that difficult the implementation into FPGAs (such as nondeterministic or data-dependent loops). Adapting a software algorithm to exploit the parallelism offered by FPGAs might require significant restructuring or redesigning of the algorithm.

Table 5.8: Overall remarks for the deployment of onboard guidance algorithms.

6 | Conclusions and future developments

6.1. Conclusion

In this work, an onboard guidance algorithm that explores convex approaches to minimum fuel space trajectory problems has been deployed in an FPGA. ECOS was selected as the interior point solver of the convex optimization problem.

An outline of the development and implementation of the algorithm into the hardware has been done in Chapter 4. A profiling sequence of the original application was carried out. The results led to the selection of two sub-routines to be implemented as IP cores. One was a numeric factorization of a sparse matrix (`kkt_factor`) while the other performed matrix-vector multiplication operations (`sparseMV`). A deep analysis of the functions to implement as IP cores and their characteristics was compelled. Afterward, a particular focus was attributed to the AXI4-Stream interface, its advantages, and design integration.

As the original problem utilized 100 nodes, a large workspace of variables resulted. Its size was incompatible with the limited resources the board used in this work presents. Therefore, a reduction of the problem size was carried out in Section 4.4.1. This scenario was viable for this thesis as the focus was to learn the main challenges and processes in the deployment of an STO algorithm in an FPGA. Other solutions could have been explored such as problem partitioning or transitioning to a board with more resources.

Chapter 5 presented and explored the implementation of the various hardware designs in comparison to the original software application. The results highlight a barrier between the original algorithm developed for software-oriented systems and its integration into hardware. Due to the ECOS environment, nondeterministic and data-dependent loops were a development bottleneck. These prevented the use of optimization techniques that exploit the parallelization and pipeline capabilities of FPGAs. A debate for alternative procedures for the selected IP cores that counter this effect was performed at the end of Sections 5.3.1 and 5.3.2. However, their integration into ECOS can be arduous due to how

convoluted the software-optimized solver is. Additionally, even after all the modifications, the design might not present the desired level of performance efficiency.

However, a possible workaround for hardware designs would be to target algorithm sections that apply multiple sequential operations as opposed to their separate implementation. This mitigates the additional cycles the transfer and data address operations that the FPGA workflow requires. Even so, these sections should be characterized by high exclusive percentages with deterministic and data-independent loops.

In Section 5.3.2 an investigation was conducted to examine the impact of fixed-point precision on interior point solvers utilized in onboard guidance algorithms. Floating-point precision data types are necessary in IP core designs due to the wide range of dynamic numbers used by the selected solver. Otherwise, even when using a meaningful fixed-point configuration, a loss in precision occurs in comparison to floating-point. As a consequence, achieving the solution requires a higher number of solver iterations, which impacts performance efficiency.

Lastly, the gained knowledge can be utilized to present a satisfactory response to the research question of this thesis, recalled hereafter.

Main Research question. *What are the key challenges and practices in deploying onboard trajectory optimization guidance algorithms on FPGA platforms?*

Answer. In essence, deploying onboard guidance algorithms on FPGAs is an ambitious task highly influenced by the board's resources. Furthermore, the optimization and structure presented by software-oriented applications pose a demanding transition into hardware. The selection of IP cores shall target functions with grand significance in execution time that convey deterministic and data-independent behaviors. These enable the parallelism and pipeline benefits FPGAs introduce. If the mentioned attributes are not present, the function's structure and implementation must be adapted. However, this can lead to complicated algorithm redesign due to how intricate interior point solvers are. A possible workaround is the selection of algorithm sections which encompass multiple sequential operations. Thus, the additional time for transfer and data address in the FPGA is counteracted by its parallelism benefit. These sections should, nonetheless, display relevance to the execution time and the before-mentioned attributes. Finally, for onboard guidance algorithms that utilize interior point solvers, floating-point precision must be employed in the IP core designs to cover the imposed range of computational accuracy.

6.2. Future Work

For future developments of onboard guidance algorithms that employ ECOS as the solver, a possible approach could be to use the `kkt_solve` in the FPGA. It solves the permuted KKT system and returns the unpermuted search directions. As reported in Figure 4.1, it is a function characterized by a high inclusive percentage but a low exclusive percentage in terms of execution time. This is due to the inclusion of several other functions which themselves are responsible for a large part of the execution time. This fits the previous scenario of an algorithm section with sequential operations of high impact.

The proposal would be to carefully analyze each function inside `kkt_solve`. As they were developed for software-oriented systems, a significant amount of alterations would be required. These should target, mainly, the data structure and organization such that deterministic and data-independent loops are achieved. One approach, opposite to changing the entire ECOS formulation, would be to transfer the data to the FPGA, store it in a structured manner that is compatible with efficient hardware implementations of the functions under use, perform the computations, and, in the end, send it back to the PS in a format consistent with the selected interior point solver. This would heavily reduce the complexity of altering the entire ECOS structure while potentially augmenting the performance efficiency.

Nonetheless, this possible algorithm section sustains several functions which, let alone, would not be compatible with the resources available. The use of optimization techniques to reinforce concurrency would exacerbate this constraint. Therefore, a different board should be selected. The Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit [1] is a device with a significant higher quantity of resources which could aid for the development of problems such as the one discussed in this thesis.

[1] `https://www.xilinx.com/products/boards-and-kits/zcu104.html` (last accessed: November 18, 2023)

Bibliography

[1] Kirk Woellert, Pascale Ehrenfreund, Antonio J. Ricco, and Henry Hertzfeld. Cubesats: Cost-effective science and technology platforms for emerging and developing nations. *Advances in Space Research*, 47:663–684, 2 2011. doi: 10.1016/j.asr.2010.10.009.

[2] Runqi Chai, Antonios Tsourdos, Al Savvaris, Senchun Chai, and Yuanqing Xia. Review of advanced guidance and control algorithms for space/aerospace vehicles. *Progress in Aerospace Sciences*, 122:100696, 04 2021. doi: 10.1016/j.paerosci.2021.100696.

[3] David Morante, Manuel Sanjurjo Rivo, and Manuel Soler. Multi-Objective Low-Thrust Interplanetary Trajectory Optimization Based on Generalized Logarithmic Spirals. *Journal of Guidance, Control, and Dynamics*, 42:1–15, 12 2018. doi: 10.2514/1.G003702.

[4] Zhenbo Wang and Michael J. Grant. Minimum-Fuel Low-Thrust Transfers for Spacecraft: A Convex Approach. *IEEE Transactions on Aerospace and Electronic Systems*, 54(5):2274–2290, 2018. doi: 10.1109/TAES.2018.2812558.

[5] Long He, Fengxiang Wang, Junxiao Wang, and Jose Rodriguez. Zynq Implemented Luenberger Disturbance Observer Based Predictive Control Scheme for PMSM Drives. *IEEE Transactions on Power Electronics*, PP:1–1, 06 2019. doi: 10.1109/TPEL.2019.2920439.

[6] Antonio Lopes Filho and Roberto d'Amore. FPGA Implementation of the JPEG XR for Onboard Earth-Observation Applications. *J. Real-Time Image Process.*, 18:2037–2048, 12 2021. doi: 10.1007/s11554-021-01078-y.

[7] Wajdi Farhat, Hassene Faiedh, Chokri Souani, and Kamel Besbes. Real-Time Embedded System for Traffic Sign Recognition Based on ZedBoard. *J. Real-Time Image Process.*, 16:1813–1823, 10 2019. doi: 10.1007/s11554-017-0689-0.

[8] Apurva S. Deulkar and Neelima R. Kolhare. FPGA implementation of audio and video processing based on Zedboard. In *2020 International Conference on Smart*

Innovations in Design, Environment, Management, Planning and Computing (ICSIDEMPC), pages 305–310, 2020. doi: 10.1109/ICSIDEMPC49020.2020.9299639.

[9] A. Fernández-León, A. Pouponnot and S. Habinc. *ESA FPGA Task Force: Lessons Learned*, September 2002.

[10] Gaisler Research. *Lessons Learned from FPGA Developments*, September 2002. Version 0.2.

[11] Gaisler Research. *Suitability of reprogrammable FPGAs in space applications*, September 2002. Version 0.4.

[12] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. ISBN 099297870X.

[13] Wang Jih-Jong, Cronquist Brian, McCollum John, Parker Wanida, Katz Rich, Kleyner Igor, Day John H. Radiation Tolerant Antifuse FPGA. Technical report, Actel Corp., NASA Goddard Space Flight Center and Orbital Sciences Corp., 1 2002.

[14] Behçet Açıkmeşe and S. Polen. Convex Programming Approach to Powered Descent Guidance for Mars Landing. *J. Guidance Control Dyn.*, 44:310–322, 01 2007. doi: 10.2514/1.27553.

[15] John T. Betts. Survey of Numerical Methods for Trajectory Optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998. doi: 10.2514/2.4231.

[16] Binfeng Pan, Ping Lu, Xun Pan, and Yangyang Ma. Double-Homotopy Method for Solving Optimal Control Problems. *Journal of Guidance, Control, and Dynamics*, 39:1–15, 06 2016. doi: 10.2514/1.G001553.

[17] Christian Hofmann and Francesco Topputo. Rapid Low-Thrust Trajectory Optimization in Deep Space Based on Convex Programming. *Journal of Guidance, Control, and Dynamics*, 44:1–10, 04 2021. doi: 10.2514/1.G005839.

[18] Jesús Gil-Fernández and Miguel Gomez-Tierno. Practical Method for Optimization of Low-Thrust Transfers. *Journal of Guidance, Control, and Dynamics*, 33:1927–1931, 11 2010. doi: 10.2514/1.50739.

[19] Xinfu Liu, Ping Lu, and Binfeng Pan. Survey of Convex Optimization for Aerospace Applications. *Astrodynamics*, 1:23–40, 02 2017. doi: 10.1007/s42064-017-0003-8.

[20] Andrea Carlo Morelli, Christian Hofmann, and Francesco Topputo. Robust Low-Thrust Trajectory Optimization Using Convex Programming and a Homotopic Ap-

proach. *IEEE Transactions on Aerospace and Electronic Systems*, 58(3):2103–2116, 11 2022. doi: 10.1109/TAES.2021.3128869.

[21] H.D. Curtis. *Orbital Mechanics: For Engineering Students*. Aerospace Engineering. Elsevier Science, 2015. ISBN 9780080470542.

[22] Francesco Topputo and Chen Zhang. Survey of Direct Transcription for Low-Thrust Space Trajectory Optimization with Applications. *Abstract and Applied Analysis*, 2014:1–15, 06 2014. doi: 10.1155/2014/851720.

[23] Christian Hofmann, Andrea C. Morelli, and Francesco Topputo. Performance Assessment of Convex Low-Thrust Trajectory Optimization Methods. *Journal of Spacecraft and Rockets*, 60(1):299–314, January 2023. doi: 10.2514/1.A35461.

[24] Nicholas Nurre and Ehsan Taheri. Comparison of Indirect and Convex-Based Methods for Low-Thrust Minimum-Fuel Trajectory Optimization. In *2022 AAS/AIAA Astrodynamics Specialist Conference*, 08 2022.

[25] Alexander Domahidi et al. ECOS: An SOCP solver for embedded systems. *2013 European Control Conference, ECC 2013*, pages 3071–3076, 07 2013. doi: 10.23919/ECC.2013.6669541.

[26] Steven Diamond and Stephen Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *J. Mach. Learn. Res.*, 17(1):2909–2913, 1 2016. doi: 10.5555/2946645.3007036.

[27] Maximilian Schaller, Goran Banjac, Steven Diamond, Akshay Agrawal, Bartolomeo Stellato, and Stephen Boyd. Embedded Code Generation with CVXPY. *IEEE Control Systems Letter*, 6:2653–2658, 3 2022. doi: 10.48550/arXiv.2203.11419.

[28] Xilinx Inc. *Vitis High-Level Synthesis User Guide, UG1399*, May 2023. Version 2023.1.

[29] Juan J. Rodríguez-Andina, María D. Valdés-Peña, and María J. Moure. Advanced Features and Industrial Applications of FPGAs — A Review. *IEEE Transactions on Industrial Informatics*, 11(4):853–864, 2015. doi: 10.1109/TII.2015.2431223.

[30] Richard Halverson and Art Lew. FPGAs for expression level parallel processing. *Microprocessors and Microsystems*, 19(9):533–540, 1995. doi: 10.1016/0141-9331(96)89281-7.

[31] Shubham Gandhare and B. Karthikeyan. Survey on FPGA Architecture and Recent Applications. In *2019 International Conference on Vision Towards Emerging Trends*

in Communication and Networking (ViTECoN), pages 1–4, 2019. doi: 10.1109/ViTECoN.2019.8899550.

[32] George Lentaris, Ioannis Stratakos, Ioannis Stamoulias, Dimitrios Soudris, Manolis Lourakis, and Xenophon Zabulis. High-Performance Vision-Based Navigation on SoC FPGA for Spacecraft Proximity Operations. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(4):1188–1202, 2020. doi: 10.1109/TCSVT.2019.2900802.

[33] MengFei Yang, Bo Liu, Jian Gong, HongJin Liu, HongKai Hu, YangYang Dong, Lei Shi, YunFu Zhao, and ZhiFu Miao. Architecture design for reliable and reconfigurable FPGA-based GNC computer for deep space exploration. *Science China Technological Sciences*, 59:pages 289–300, 11 2015. doi: 10.1007/s11431-015-5936-7.

[34] Mohammad I. AlAli, Khaldoon M. Mhaidat, and Inad A. Aljarrah. Implementing image processing algorithms in FPGA hardware. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–5, 2013. doi: 10.1109/AEECT.2013.6716446.

[35] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 11 1979. doi: 10.1126/science.206.4420.776.

[36] T.C. May and M.H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, 1 1979. doi: 10.1109/T-ED.1979.19370.

[37] Heather Quinn, Paul Graham, Keith Morgan, Jim Krone, Michael Caffrey, and Michael Wirthlin. An Introduction to Radiation-Induced Failure Modes and Related Mitigation Methods For Xilinx SRAM FPGAs. pages 139–145, 01 2008.

[38] S. Azimi, L. Sterpone, B. Du, and L. Boragno. On the analysis of radiation-induced Single Event Transients on SRAM-based FPGAs. *Microelectronics Reliability*, 88-90:936–940, 9 2018. doi: 10.1016/j.microrel.2018.07.135.

[39] Fredrik Bruhn, Kjell Brunberg, John Hines, Lars Asplund, and Magnus Norgren. Introducing radiation tolerant heterogeneous computers for small satellites. In *2015 IEEE Aerospace Conference*, pages 1–10, 2015. doi: 10.1109/AERO.2015.7119158.

[40] P. Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL: Second Edition*. Newnes, 2015. ISBN 9780750668453.

[41] Xilinx Inc. *7 Series DSP48E1 Slice User Guide, UG479*, March 2018. Version 1.10.

[42] Xilinx Inc. *7 Series FPGAs Memory Resources, UG473*, July 2019. Version 1.14.

[43] ARM. *ARM Cortex-A9 MPCore*, January 2016. Revision r4p1.

[44] Xilinx Inc. *Zynq-7000 Technical Reference Manual, UG585*, July 2018. Version 1.12.2.

[45] Xilinx Inc. *Vivado Design Suite: AXI Reference Guide, UG1037*, July 2017. Version 4.0.

[46] Xilinx Inc. *Vivado Design Suite User Guide, UG893*, April 2022. Version 2022.1.

[47] Xilinx Inc. *Vitis Unified Software Platform Documentation, UG1400*, April 2022. Version 2022.1.

[48] Xilinx Inc. *Vitis Model Composer User Guide, UG1483*, May 2022. Version 2022.1.

[49] Don Lahiru Nirmal Hettiarachchi, Venkata Salini Priyamvada Davuluru, and Eric J. Balster. Integer vs. Floating-Point Processing on Modern FPGA Technology. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0606–0612, 2020. doi: 10.1109/CCWC47524.2020.9031118.

[50] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[51] Caroline N. Haddad. *Cholesky Factorization*, pages 374–377. Springer US, 2009. ISBN 978-0-387-74759-0. doi: 10.1007/978-0-387-74759-0_67.

[52] Tim Davis. User Guide for LDL, a concise sparse Cholesky package, 6 2012.

[53] Alexander Domahidi. *Methods and tools for embedded optimization and control*. Doctoral thesis, ETH Zurich, 2013.

[54] L. Vandenberghe. *The CVXOPT linear and quadratic cone program solvers*, March 2010. Version 1.1.2.

[55] Iain S. Duff, Roger G. Grimes, and John Gregg Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15:1–14, 6 1982. doi: 10.1145/1057588.1057590.

[56] Xilinx Inc. *AXI DMA LogiCORE IP Product Guide, PG021*, April 2022. Version 7.1.

[57] Xilinx Inc. *Processing System 7 LogiCORE IP Product Guide, PG082*, May 2017. Version 5.5.

[58] Xilinx Inc. *AXI Interconnect LogiCORE IP Product Guide, PG059*, April 2017. Version 2.1.

[59] Xilinx Inc. *AXI Timer v2.0 Product Guide, PG079*, October 2016. Version 2.0.

[60] Daniele Bagni, A. Di Fresco, J. Noguera, and F. M. Vallina. A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS. Technical report, Xilinx Inc., 1 2016. XAPP1170 (v2.0).

[61] Depeng Yang, Junqing Sun, Junkyu Lee, Getao Liang, David Jenkins, Gregory Peterson, and Husheng Li. Performance Comparison of Cholesky Decomposition on GPUs and FPGAs. In *Conference: Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 07 2010.

[62] Depeng Yang, Gregory Peterson, Husheng Li, and Junqing Sun. An FPGA Implementation for Solving Least Square Problem. In *17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 303–306, 01 2009. doi: 10.1109/FCCM.2009.47.

[63] Jakub Kurzak and Jack Dongarra. Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead. In *Computer Science and Mathematics Division, Oak Ridge National Laboratory*, volume 178, 06 2006. ISBN 978-3-540-75754-2. doi: 10.1007/978-3-540-75755-9_18.

[64] Jun Luo, Qijun Huang, Sheng Chang, Xiaoying Song, and Yun Shang. High throughput Cholesky decomposition based on FPGA. In *2013 6th International Congress on Image and Signal Processing (CISP)*, volume 03, pages 1649–1653, 2013. doi: 10.1109/CISP.2013.6743941.

[65] Ling Zhuo and Viktor K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, page 63–74. Association for Computing Machinery, 2005. ISBN 1595930299. doi: 10.1145/1046192.1046202.

[66] Junyi Liu, Helfried Peyrl, Andreas Burg, and George A. Constantinides. FPGA implementation of an interior point method for high-speed model predictive control. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014. doi: 10.1109/FPL.2014.6927473.

[67] Avnet. *ZedBoard (Zynq Evaluation and Development) Hardware User's Guide*, January 2014. Version 2.2.

A | Appendix A

A.0.1. Zynq PS Interconnections

Figure A.1 presents a more detailed scheme of the interconnections of the Zynq PS, also showing some of the components and specifications referred in Section 3.2.2.

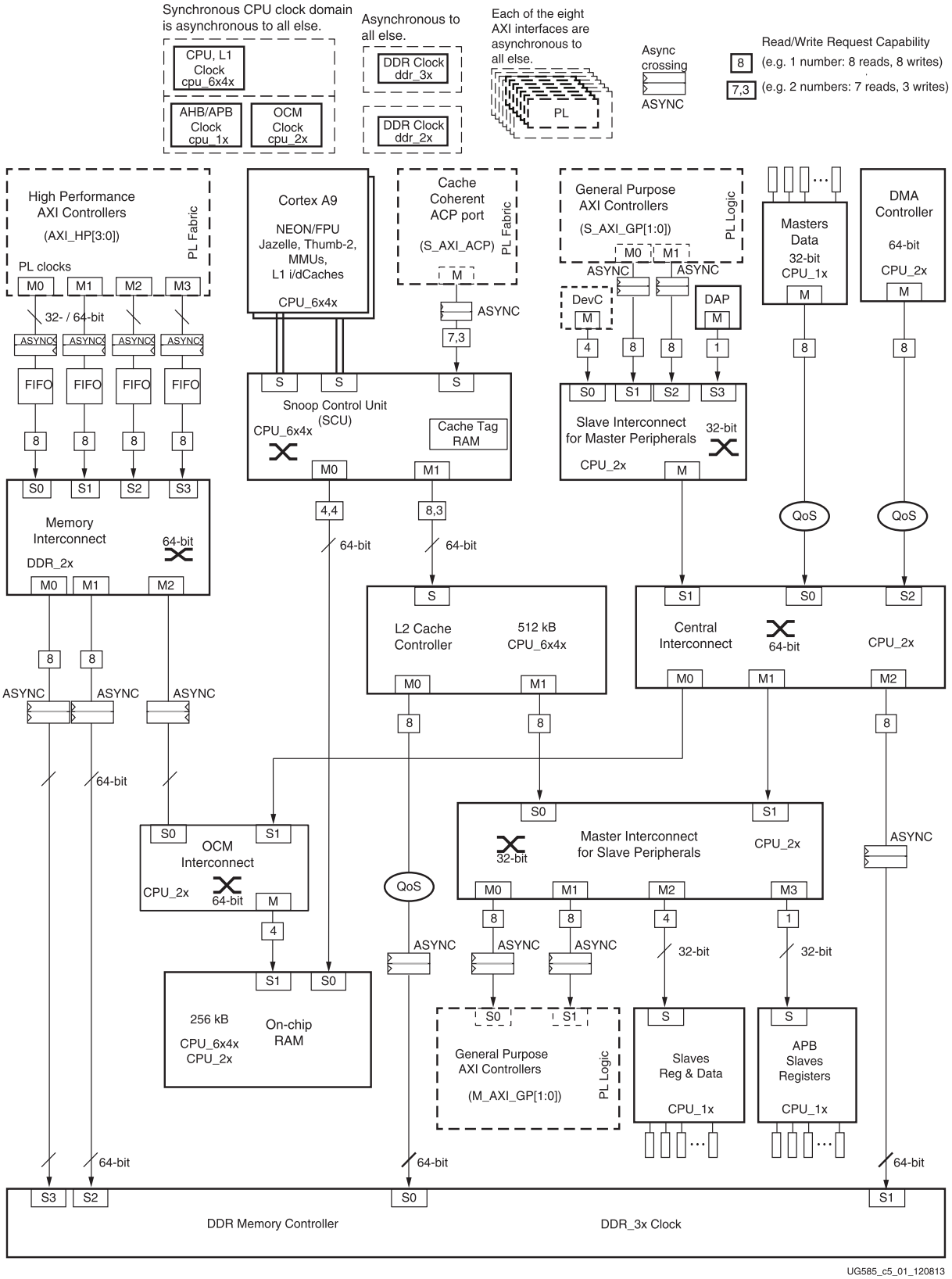


Figure A.1: Detailed diagram of interconnections [44].

A.0.2. Zedboard Booting and Programming Methods

The Zedboard presents a set of five configuration jumpers which the user can position to select the desired method of booting/programming, them being:

- USB-JTAG: the default and simpler method of programming the Zedboard.
- Traditional JTAG: similar to the previous method but with a different cable.
- Quad-SPI flash memory: the flash memory (non-volatile) can be used to store configuration data. This method removes the requirement for a wired connection to program the device.
- SD card: can be used to program the device with files stored on the SD card. Once again, no need for a wired connection for programming.

For more information about the jumpers configuration to select each programming method, check [67].

A.0.3. Placement and Routing

The placement and routing circuit on the FPGA of each implementation is provided in Figures A.2, A.3 and A.4.

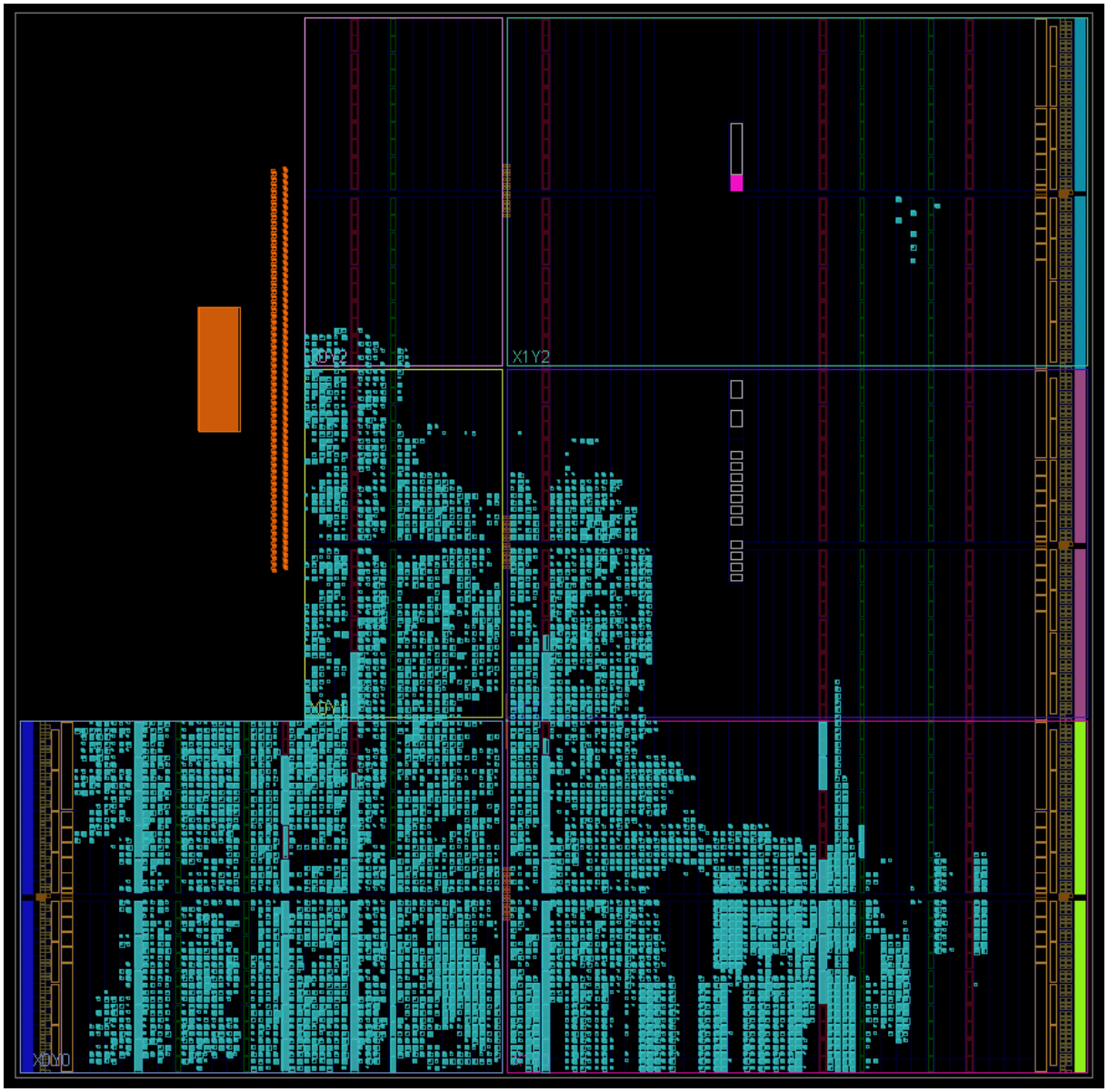


Figure A.2: Design including `kkt_factor` IP core placing and routing circuit on FPGA.

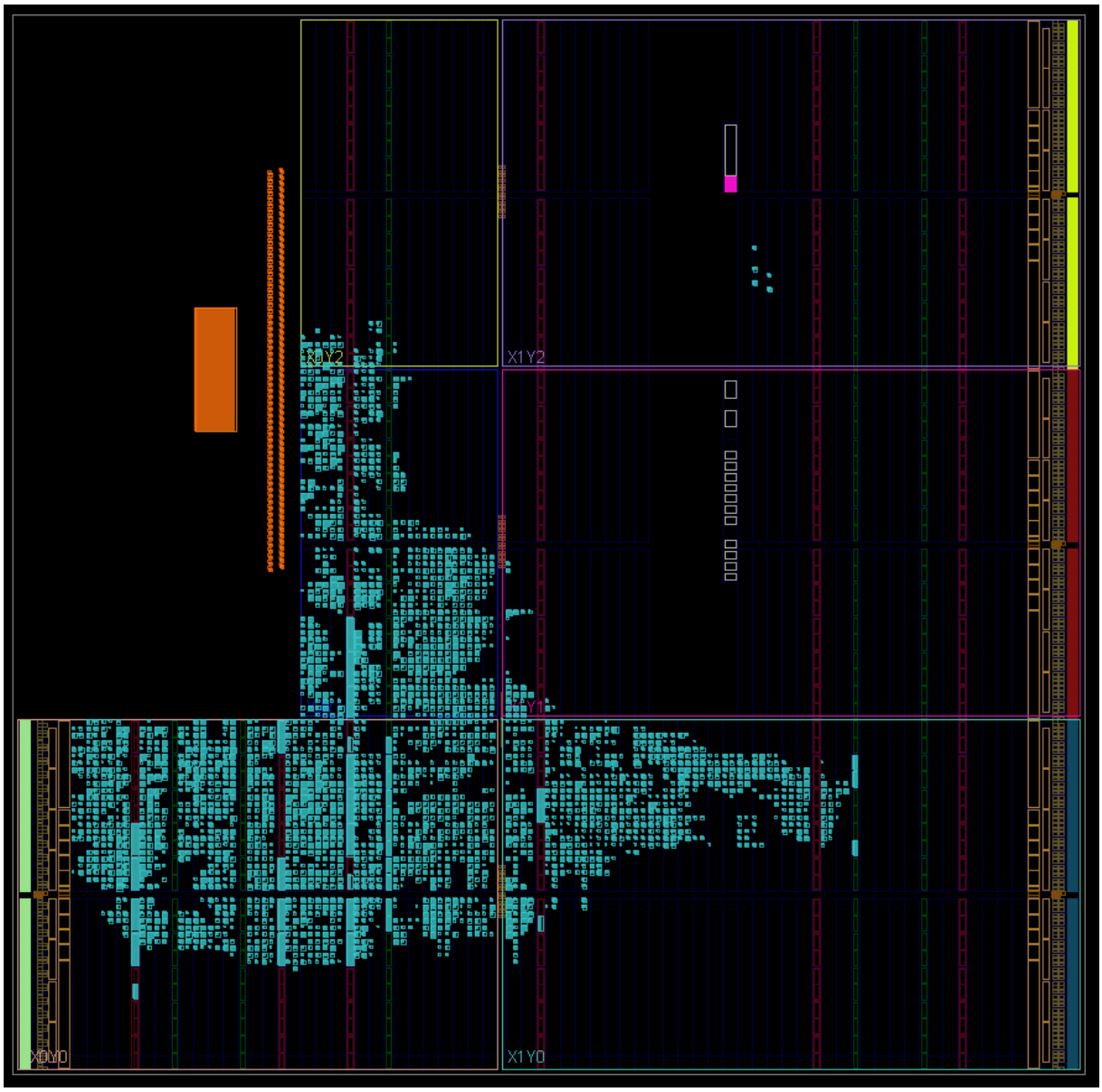


Figure A.3: Design including `sparseMV` floating-point IP core placing and routing circuit on FPGA.

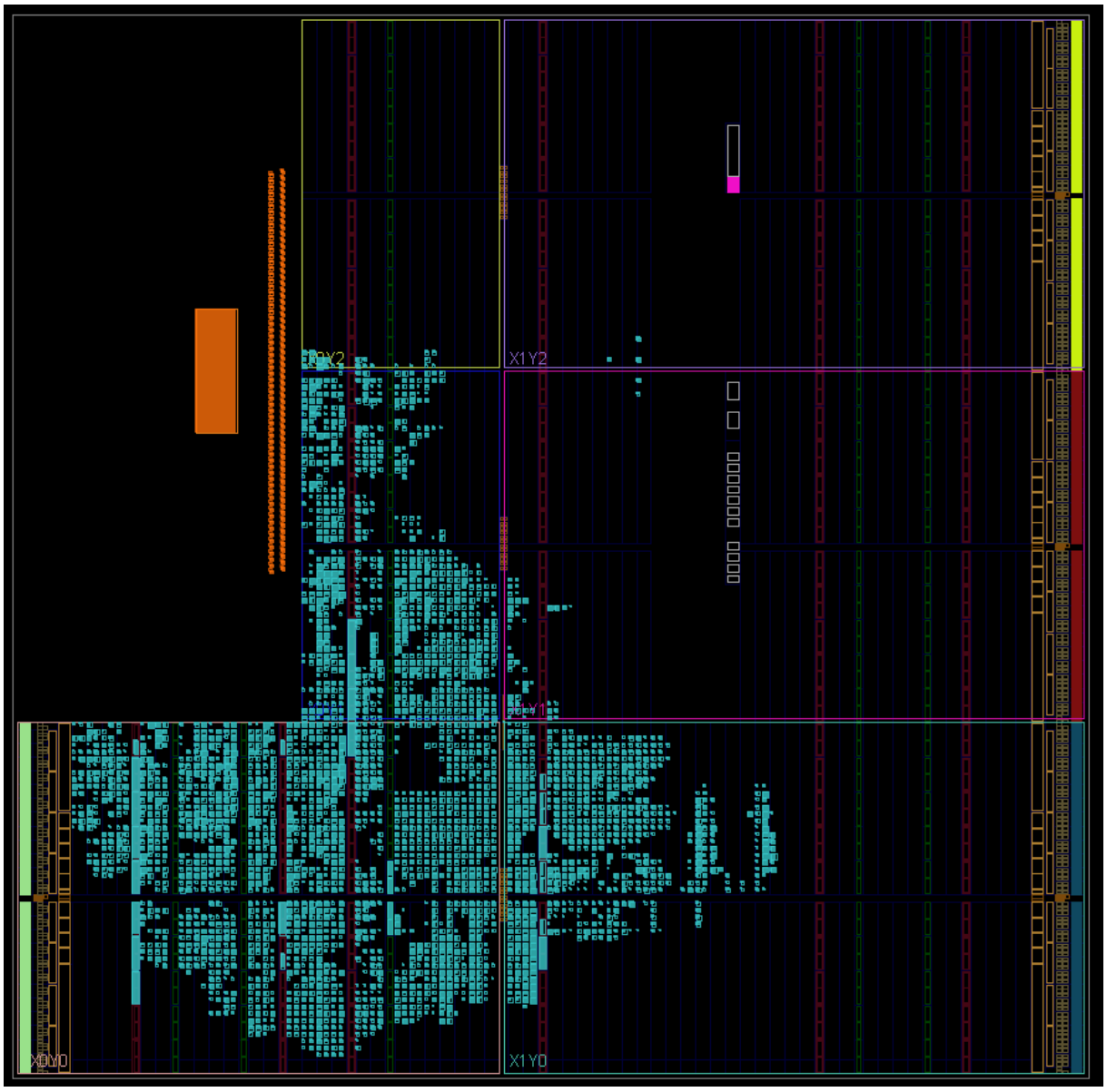


Figure A.4: Design including `sparseMV` fixed-point IP core placing and routing circuit on FPGA.

List of Figures

List of Tables

Nomenclature

ACP Accelerator Coherent Port

AMBA Advanced Microcontroller Bus Architecture

APSoC All-Programmable System-on-Chip

ASIC Application-Specific Integrated Circuit

AU Astronomical Unit

AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CCS Compressed Column Storage

CDF Cumulative Distribution Function

CLB Configurable Logic Block

COTS Commercial-off-the-Shelf

CPU Central Processing Unit

CXP Convex Problem

DMA Direct Memory Access

DSP Digital Signal Processor

FF Flip-Flops

FIFO First In First Out

FPGA Field Programmable Gate Array

GP General Purpose

GPP General Purpose Processor

HDL Hardware Description Language

HLS High-Level Synthesis

HP High Performance

IDE Integrated Design Environment

II Initiation Interval

IOB Input/Output Blocks

IP Intellectual Property

LUT Lookup Table

MCDMA MultiChannel Direct Memory Access

MM2S Memory-Mapped to Streaming

MPC Model Predictive Control

MSB Most Significant Bit

NASA National Aeronautics and Space Administration

NLP Nonlinear Programming

OBC OnBoard Computer

PL Programmable Logic

PS Processing System

ROM Read Only Memory

RTL Register Transfer Level

S2MM Streaming to Memory-Mapped

SCP Sequential Convex Programming

SEU Single Event Upset

SoC System-on-Chip

SOCP Second-Order Cone Programming

STO Space Trajectory Optimization

TCF Target Communication Framework

TCL Tool Command Language

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits