UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

IMPLEMENTACIÓN DE ACELERADORES DE CÓMPUTO EN LÓGICA RECONFIGURABLE PARA APLICACIONES DE CONTROL PREDICTIVO POR MODELO UTILIZANDO SÍNTESIS DE ALTO NIVEL

ALFONSO NICOLÁS CORTÉS NEIRA

MEMORIA DE TITULACIÓN Y TESIS DE GRADO PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO Y AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA ELECTRÓNICA

PROFESOR SUPERVISOR : GONZALO CARVAJAL BARRERA
PROFESOR CO-SUPERVISOR : CÉSAR SILVA JIMÉNEZ

28 DE MARZO DE 2024

Resumen

El Control Predictivo por Modelo es una técnica de control automático que se caracteriza por predecir el comportamiento de la planta para determinar una secuencia de actuaciones y seguir una referencia a lo largo de un horizonte de tiempo. Al formularse como un problema de optimización, MPC puede utilizarse en plantas con múltiples entradas y considerar intrínsecamente las restricciones del sistema. Sin embargo, el costo computacional de resolver la optimización es significativo y aumenta con la dimensión del modelo, el número de restricciones y el horizonte de tiempo, lo que representa un desafío para su ejecución en tiempo de operación de la planta en gran número de aplicaciones.

Las *Field Programmable Gate Arrays* se han utilizado para disminuir la latencia de secciones del procesamiento de MPC. No obstante, aún usando una FPGA puede ser necesario un procesador de propósito general (CPU) para implementar el lazo de control completo, lo que implica la utilización de arquitecturas heterogéneas. Las plataformas que combinan CPU con FPGA ofrecen la posibilidad de ejecutar las partes seriales de un algoritmo en software y las partes paralelizables en lógica programable. Sin embargo, al distribuir la ejecución de un algoritmo entre la CPU y la FPGA, también debe tenerse en cuenta la latencia de la comunicación entre ambas partes. Asimismo, debe considerarse la complejidad de implementar ciertos elementos del lazo de control, como las interfaces con sensores, actuadores, con el usuario o con otros sistemas. La integración de lógica programable junto a procesadores de propósito general resulta entonces en nuevos compromisos que no han sido explorados en la literatura, pero que son relevantes para alcanzar objetivos de latencia usando sistemas heterogéneos.

Por otro lado, se han desarrollado flujos alternativos al diseño convencional utilizando lenguajes de descripción de hardware para el diseño de aceleradores en FPGA. Herramientas modernas, como la Síntesis de Alto Nivel, permiten reducir el uso de lenguajes de descripción de hardware, facilitando la verificación del diseño y potencialmente reduciendo el tiempo de diseño. Trabajos recientes han explorado el uso de herramientas de alto nivel para la aceleración de Control Predictivo por Modelo en particular.

El trabajo propuesto pretende analizar la implementación de sistemas digitales con aceleradores en FPGA. Se explorará el uso de arquitecturas heterogéneas y de Síntesis de Alto Nivel para la implementación de sistemas que integren procesamiento en CPU y en hardware. La implementación de Control Predictivo por Modelo se considera relevante para evaluar las capacidades y limitaciones actuales de Síntesis de Alto Nivel y de los sistemas heterogéneos, por tratarse de un algoritmo computacionalmente demandante y paralelizable.

Índice general

1. Introducción 7
1.1. Contexto y motivación . . . 7
1.2. Planteamiento del problema . . . 10
1.3. Alcances y contribuciones . . . 11
1.4. Organización del Informe . . . 12

2. Antecedentes 13
2.1. Formulación del problema de control . . . 13
2.1.1. Control Predictivo por Modelo . . . 13
2.1.2. Formulación densa de MPC . . . 17
2.1.3. *Alternating Direction Method of Multipliers* (ADMM) . . . 22
2.2. Implementación de aceleradores de MPC en sistemas heterogéneos . . . 23
2.2.1. Sistemas heterogéneos . . . 23
2.2.2. Caracterización de desempeño . . . 24
2.2.3. Síntesis de Alto Nivel (HLS) . . . 26

3. Análisis de código para algoritmo MPC 28
3.1. Algoritmo de MPC usando ADMM . . . 28
3.1.1. Profiling del controlador . . . 29
3.1.2. Dimensionalidad del problema . . . 31
3.1.3. Implementación en código C++ . . . 32
3.2. Potencial de paralelización y estrategias para HLS . . . 33
3.2.1. Operaciones elemento-a-elemento . . . 34
3.2.2. Multiplicación matriz-vector . . . 36
3.2.3. Potencial de paralelización de ADMM . . . 38

4. Generación de aceleradores en hardware usando HLS 39
4.1. Flujo de herramientas para HLS . . . 39
4.1.1. Diseño funcional . . . 39
4.1.2. Diseño de hardware con HLS . . . 40
4.1.3. Implementación en SoC . . . 40
4.1.4. Ejecución del acelerador . . . 41
4.2. Exploración de soluciones usando pragmas . . . 41
4.2.1. Ejemplo de referencia: motor DC . . . 43
4.2.2. Análisis de escalabilidad . . . 47

5. Caso de estudio: MPC para Distributed Energy Resource **49**
5.1. Descripción de la aplicación. 49
5.2. Implementación del controlador en sistema heterogéneo 50
5.2.1. Caracterización del tiempo de comunicación CPU-acelerador 51
5.2.2. Diseño de acelerador con HLS 54
5.3. Validación del controlador en hardware 55
5.4. Escalabilidad respecto al número de restricciones 57

6. Conclusiones y trabajo futuro **59**
6.1. Conclusiones 59
6.2. Trabajo futuro 60

A. Multiplicación Matriz-Vector **61**

B. Frecuencia del reloj **64**

C. Latencia de la comunicación AXI con Pynq **66**

D. Hardware-Oriented Code **68**

Índice de tablas

3.1. *Profiling* del procesamiento *online*. 30
3.2. Parámetros de la aplicación. 31
3.3. Operaciones aritméticas en ADMM. 32

4.1. Resultados post implementación para el motor DC. 46
4.2. Tiempo de ejecución en software y hardware respecto al horizonte. 48

5.1. Intervalos entre transacciones AXI en Pynq. 54
5.2. Intervalos entre transacciones AXI en baremetal. 54
5.3. Resultados post HLS para DER. 55

Índice de figuras

2.1. Control Predictivo por Modelo. 14
2.2. Diagrama de bloques de un sistema heterogéneo. 23
2.3. Diagramas de bloques de chips tipo SoC de AMD-Xilinx. 24

3.1. Planificación de bucle sin dependencia de datos entre iteraciones. 35

4.1. Ejemplo de reporte de HLS. 40
4.2. Ejemplo de Block Design en Vivado. 41
4.3. Resultados de HLS al aplicar unroll parcial de EOP y/o MVM. 45
4.4. Frecuencia de muestreo contra uso de DSPs. 45
4.5. Escalabilidad respecto al horizonte. 47

5.1. Verificación funcional del MPC para DER en MatLab. 50
5.2. Diagrama de bloques de MPC en un sistema heterogéneo. 51
5.3. Transferencia AXI. 52
5.4. Block Design del controlador MPC en sistema heterogéneo con *sniffer*. 53
5.5. Intervalo medido entre transacciones AXI consecutivas. 53
5.6. Estados del sistema DER emulado en CPU con control acelerado en FPGA. . . 56
5.7. Transacciones AXI entre CPU y FPGA para al control MPC de DER. 57
5.8. Escalabilidad respecto al número de restricciones. 58

A.1. Inversa de la latencia contra uso de DSPs en MVM. 62

B.1. Latencia de acelerador contra periodo de reloj. 65

C.1. Histogramas de intervalos entre transacciones AXI. 67

Índice de códigos

3.1. Procesamiento online de MPC en MatLab. 29
3.2. MVM tipo *column-wise* en C++. 36
3.3. MVM tipo *row-wise* en C++. 37

4.1. MVM tipo *row-wise* con *pragmas*. 43
4.2. Implementación de ADMM con *pragmas*. 44

1 | Introducción

Este capítulo presenta el contexto que motiva el trabajo del presente informe, para luego establecer la problemática que buscamos resolver. Por último, especificamos los correspondientes alcances y contribuciones de esta tesis.

1.1. Contexto y motivación

El Control Predictivo por Modelo (Model Predictive Control o MPC) es una técnica avanzada de control que, utilizando un modelo matemático del sistema a controlar y el valor de los estados del sistema en un instante de tiempo dado, es capaz de predecir el comportamiento del sistema y determinar una secuencia óptima de entradas para que el sistema alcance una referencia en un número finito de muestras futuras, sujeto a restricciones en los valores de las entradas y los estados [1]. En cada instante de muestreo, un algoritmo MPC realiza una predicción del comportamiento futuro del sistema a lo largo del horizonte de predicción, usando estas predicciones para encontrar la secuencia de acciones de control que optimizan un funcional de costo que penaliza las variaciones de las entradas y el error en los salidas. MPC permite implementar objetivos de control complejos, como por ejemplo "*la energía generada por la turbina debe mantenerse lo más cercana posible a su valor de referencia, evitando que la temperatura de la caldera exceda 1200 grados y consumiendo la menor cantidad de combustible posible*"[2].

En el caso de un sistema lineal (o linealizado) e invariante en el tiempo, con restricciones lineales, el problema de optimización a resolver en cada instante de muestreo se puede formular como un problema de *Quadratic Programming* (QP) con restricciones de igualdad y desigualdad. El controlador MPC puede resolver el problema QP de forma explícita o implícita [3]. La formulación explícita explota la propiedad de que el espacio de soluciones del problema QP se puede representar por un conjunto finito de funciones afines definidas por tramos [4]. Al implementar MPC explícito, las regiones y actuaciones asociadas son calculadas en tiempo de diseño y almacenadas en memoria; luego, en tiempo de ejecución, la actuación óptima para un estado dado se obtiene mediante una operación de búsqueda de la región y la evaluación de una función afín. Por otro lado, la formulación implícita de MPC requiere formular un nuevo problema QP en cada instante de muestreo y resolverlo mediante métodos numéricos iterativos, lo cual reduce los requerimientos de memoria pero incrementa la demanda computacional en tiempo de ejecución.

En términos de costo computacional, la formulación explícita de MPC resulta adecuada para la implementación de lazos de control de dimensionalidad moderada que requieran operación en tiempo real con baja latencia. Sin embargo, el número de regiones necesario para

representar en forma exacta la ley de control aumenta significativamente con la dimensionalidad del problema, la cual a su vez crece con el número de estados, número de muestras en el horizonte de predicción y el número de restricciones del problema. En la práctica, el aumento del número de regiones conlleva mayores requerimientos de memoria para almacenarlas, y mayor tiempo de búsqueda de la región óptima durante la ejecución del lazo de control. Las restricciones de escalabilidad se vuelven particularmente relevantes cuando los lazos de control deben ser implementados en plataformas embebidas con limitada capacidad de procesamiento y almacenamiento de datos. Por lo tanto, si bien MPC explícito ha permitido implementar lazos de control con latencias en el orden de microsegundos, esto solo se ha validado en sistemas simples con horizontes de predicción acotados [4]. En respuesta a esto, la literatura reciente muestra un renovado interés en el desarrollo de *QP solvers* eficientes que permitan reducir los tiempos de ejecución requeridos para utilizar MPC implícito.

Existen distintas familias de algoritmos iterativos para resolver problemas QP [3, 5], los cuales tienen como característica común el uso intensivo de operaciones de álgebra lineal. Las operaciones requeridas suelen exponer un nivel de paralelismo que puede ser explotado mediante el uso de arquitecturas de cómputo paralelo. En este contexto, la literatura reciente reporta múltiples implementaciones de *QP solvers* en *Field Programmable Gate Arrays* (FPGAs) [6, 7], dispositivos que ofrecen capacidades de especialización de cómputo en términos de representación de datos, resolución aritmética, tipo de paralelismo, y frecuencia de reloj para alcanzar el desempeño deseado considerando diferentes compromisos. El grado de control en la especialización de la arquitectura de hardware para el *solver* permite establecer garantías en términos de latencia de ejecución de las operaciones aritméticas que no son factibles de alcanzar con sistemas de arquitectura fija, como microprocesadores de múltiples núcleos o Graphic Processing Units (GPUs).

La literatura académica reporta numerosas implementaciones de aceleradores basados en FPGA para *QP solvers* orientados a MPC. El trabajo en [5] presenta un resumen de la evolución del estado del arte al año 2018. En general, cada trabajo revisado presenta una aplicación motivacional (por ejemplo, el control de un motor) y un tipo de algoritmo iterativo para resolver el problema QP, observándose como factor común el requerimiento de acelerar operaciones de álgebra lineal que representan el cuello de botella durante la ejecución. Luego, se analizan las características del caso de estudio motivacional para identificar la dimensionalidad del problema y patrones de representación y acceso de datos, para así definir una arquitectura paralela ad-hoc al problema de interés. A la fecha en que se realizó el estudio, las operaciones de punto flotante solían considerarse inadecuadas para su implementación en FPGA, por lo cual los diseños planteados suelen utilizar representaciones de punto fijo, lo que requiere un análisis numérico en tiempo de diseño para reducir la precisión aritmética sin sacrificar calidad en el control a nivel de aplicación o afectar la estabilidad numérica de los algoritmos [7, 8]. Si bien se reportan implementaciones de MPC implícito que alcanzan períodos de control inferiores al microsegundo [7, 9], esto suele lograrse mediante arquitecturas ad-hoc al caso de estudio planteado, y no hay evidencia de que las soluciones propuestas sean adaptables y extensibles a otros problemas. Además, todas las implementaciones reportadas se basan en descripciones en Hardware Description Languages (HDL) a nivel de Register Transfer Level (RTL), lo cual requiere un expertiz en diseño de hardware y añade una barrera adicional para validar la generalidad de las soluciones propuestas.

Posterior a la publicación de [5], se desarrolló una proliferación, o más bien maduración, de herramientas de generación asistida o automatizada de descripciones RTL a partir de

descripciones funcionales de alto nivel. En particular, el paradigma de Síntesis de Alto Nivel (High Level Synthesis o HLS) que se presenta en el contexto de FPGAs, apunta a producir una descripción de hardware para un algoritmo descrito mediante un lenguaje de alto nivel como C/C++, permitiendo al usuario especificar propiedades deseadas de la arquitectura para explotar el paralelismo de las operaciones. La descripción de paralelismo se realiza por medio de directivas en forma de *pragmas* directamente en el código de alto nivel, las cuales permiten explotar paralelismo espacial y temporal sin necesidad de replicar código o describir transacciones entre registros sincronizados. La aparición de herramientas comerciales de HLS de los principales fabricantes de FPGA, ha gatillado un interés por la utilización de este paradigma para la generación de aceleradores para MPC.

En el trabajo presentado en [10], los autores muestran que HLS permite implementar controladores con distinto grado de paralelización, pero el análisis se limita a la aceleración de operaciones de multiplicación matriz-vector. Si bien este trabajo muestra el potencial de HLS para el prototipado rápido de aceleradores de hardware para MPC implícito, no hace un análisis de otras posibles estrategias costo-efectivas de aplicar *pragmas* a las operaciones de álgebra lineal, quedando un amplio espacio para mejoras. Por otro lado, en [11] se reporta que el uso de *pragmas* no incide en la arquitectura ni en el tiempo de ejecución del algoritmo. En este último caso, el uso de HLS no es el foco del trabajo y se trata en forma superficial, observándose indicios de que los autores omiten conceptos fundamentales sobre los requerimientos e implicancias del uso de pragmas, lo cual evidencia que la aplicación efectiva de HLS no es trivial. Si bien la promesa del paradigma HLS es facilitar la generación de aceleradores especializados al reducir o eliminar la necesidad de describir código RTL, en la práctica se observa que el uso efectivo de estas herramientas aún requiere conocer los fundamentos de diseño de sistemas digitales sincrónicos. Por otro lado, desde la perspectiva de alguien con experiencia en diseño para FPGAs, las herramientas actuales para HLS efectivamente facilitan la exploración del espacio de diseño para un acelerador por medio de la generación automática de diferentes alternativas de implementación utilizando distintos *pragmas* sobre un mismo código base, lo cual reduce significativamente la complejidad y el tiempo de diseño en comparación a escribir un código HDL específico para cada arquitectura que se quiera probar. En base a esto, se establece la necesidad de realizar análisis profundos y sistemáticos sobre los patrones de procesamiento de datos de algoritmos típicos para problemas MPC, para así establecer directrices generales que puedan ser aplicadas a distintos problemas considerando compromisos y requerimientos específicos.

Por último, cabe precisar que la implementación en hardware de las partes demandantes de un algoritmo de control no implica necesariamente prescindir de un procesador de propósito general (CPU). En la práctica, no todas las operaciones del controlador podrán ser implementadas en hardware, y es común requerir de una CPU para la conexión con periféricos de entrada/salida, filtraje de datos, ejecución de instrucciones seriales del algoritmo de control, monitoreo e interacción con el usuario, etc. En general, se espera que sea común que un lazo de control MPC se implemente en una arquitectura heterogénea que integre uno o más procesadores de propósito general y lógica programable interconectados mediante buses de datos. Luego, el diseño del lazo de control a nivel de sistema, incluye decidir qué subrutinas o secciones del algoritmo de MPC son procesadas en una CPU y cuáles son procesadas en una FPGA. La aceleración efectiva de sistema de control al integrar aceleradores de hardware dependerá de las latencias asociadas al sistema completo, por lo que se hace necesario incorporar la latencia de comunicación entre FPGA y CPU dentro de la caracterización del tiempo de ejecución del lazo de control, lo cual hasta ahora se ha omitido en la literatura asociada.

El presente trabajo de tesis se enfoca en explorar el uso del paradigma HLS para el diseño de aceleradores basados en FPGA para algoritmos de MPC implícito orientados a su implementación en plataformas con arquitectura heterogénea que permitan integración con un procesador de propósito general (CPU). Esto responde a la necesidad de reducir el esfuerzo y tiempo de diseño de estos aceleradores, así como de habilitar su diseño por usuarios no expertos en sistemas digitales. Aprovechando el enfoque *top-down* [1] de HLS, se busca derivar directrices generales que faciliten llegar desde una aplicación a una arquitectura especializada.

1.2. Planteamiento del problema

Considerando el contexto descrito en la Sección 1.1, se formula el problema a tratar en esta tesis de la siguiente manera: *"Dada una descripción en lenguaje de alto nivel (Matlab, C/C++) para un controlador MPC lineal basado en el algoritmo ADMM para la resolución del problema QP asociado, se requiere establecer directrices para facilitar la exploración de espacio de diseño en el desarrollo de aceleradores en FPGA utilizando HLS, considerando distintos compromisos en términos de uso de recursos y latencia del controlador. Los aceleradores deben considerar su potencial integración en plataformas heterogéneas CPU+FPGA."*

Para focalizar el trabajo y facilitar la exploración y el análisis, se utilizan como referencia dos casos de estudio de sistemas MPC lineales previamente documentados y con requerimientos definidos: (i) un motor DC, que es un problema utilizado frecuentemente en la literatura para ilustrar los conceptos de MPC y por lo tanto sirve como referencia funcional para validar las metodologías propuestas, y (ii) un convertidor DC/AC con filtro LC para un Sistema de Energía Distribuido (Distributed Energy Resource o DER), que es una aplicación de interés para el grupo de investigación que demanda operación en tiempo real y que ya ha sido estudiada en [12, 13]. El contar con referencias funcionales de los lazos de control ya definidas y validadas, nos permite abstraernos de la teoría de control y algoritmos subyacentes y enfocarnos exclusivamente en los aspectos computacionales que permitan reducir los tiempos de ejecución.

Para este estudio, se apunta a implementar controladores con MPC implícito para los sistemas de referencia que permitan operar con tiempos de muestreo inferiores al milisegundo y utilizando punto flotante. En ambos casos se utiliza el algoritmo ADMM para la resolución del problema de optimización QP asociado al lazo de control, el cual se ha establecido en la literatura reciente como el más adecuado para implementaciones de baja latencia dada su compatibilidad con cómputo paralelo [5, 14, 15].

En base a los ejemplos de referencia, se requiere realizar un análisis de las operaciones aritméticas y los patrones de acceso a datos de los algoritmos, para así explorar configuraciones de *pragmas* e identificar combinaciones costo-efectivas considerando compromisos en términos de uso de los recursos disponibles en la FPGA y la latencia del lazo de control para problemas de distinta escala. Las directrices derivadas del análisis deben ser suficientemente generales para facilitar su aplicación y validación en problemas del mismo tipo u otros problemas con flujo de operaciones y patrones de acceso a datos similar.

[1] En el contexto de este trabajo, el enfoque *top-down* se refiere a que el desarrollo de la arquitectura del acelerador de hardware parte desde una descripción algorítmica de la aplicación objetivo y a partir de esta se van agregando detalles para las capas y subsistemas inferiores en forma gradual, hasta llegar a la descripción RTL y la implementación en FPGA.

1.3. Alcances y contribuciones

Para el desarrollo de esta tesis se consideraron los siguientes alcances:

A1 Desde el punto de vista de la aplicación de control, el trabajo solo considera evaluar la aceleración por hardware de lazos MPC basados en el algoritmo ADMM, para los cuales se cuenta con descripciones en Matlab que ya han sido validadas funcionalmente por otros estudiantes e investigadores del Departamento de Electrónica. El trabajo no considera el desarrollo de nuevas teorías o algoritmos de control, más allá de las modificaciones requeridas en los algoritmos para reducir el tiempo de ejecución del lazo de control en los casos de estudio indicados en la Sección 1.2. El criterio para validar la funcionalidad de las implementaciones finales será la equivalencia numérica con las descripciones de alto nivel usadas como referencia.

A2 Se utilizará solamente representación numérica en punto flotante. Esto se considera deseable para facilitar la exploración de optimizaciones al flujo de operaciones sin requerir análisis de aspectos numéricos que puedan afectar la estabilidad del algoritmo. Dado que las FPGAs modernas de distinta gama suelen integrar bloques DSP dedicados, se considera que este alcance no representa una limitante práctica para la implementación de los algoritmos evaluados y su integración en sistemas reales.

A3 En el contexto de este trabajo nos referiremos como sistema heterogéneo a una plataforma que integre CPU y lógica reconfigurable (FPGA). Para la evaluación experimental se utilizará la plataforma de desarrollo ZCU104 de AMD-Xilinx, la cual corresponde a un *MultiProcessor System on Chip* (MPSoC) que integra CPU y FPGA en el mismo circuito integrado. Las herramientas de desarrollo a utilizar incluyen Vitis HLS 2022.2, Vivado 2022.2 y Vitis 2022.2. Se espera que las evaluaciones realizadas sean extensibles a otras tarjetas de desarrollo del mismo fabricante, sujeto a las restricciones de recursos en cada caso.

Basado en los resultados obtenidos, se identifica las siguientes principales contribuciones:

C1 Se entrega evidencia de que HLS permite diseñar aceleradores para problemas de mediana escala, para su implementación en arquitecturas que integren CPU y FPGA con latencias inferiores al milisegundo.

C2 A partir de los resultados obtenidos sobre los casos de estudio específico, se derivan directrices generales que facilitan la exploración y diseño escalable de aceleradores de cómputo, reduciendo la brecha de entrada al desarrollo de hardware especializado para usuarios no experimentados en diseño digital en HDL.

Este informe es complementario a un repositorio [2] que contiene información y códigos para replicar los resultados. Además, a partir de los resultados de este trabajo se presentó el artículo *Leveraging High Level Synthesis for the Design of Hardware Accelerators for Model Predictive Control* en la Conferencia Argentina de Electrónica (CAE) [16].

[2]Al momento de entregar este reporte, los códigos desarrollados se encuentran accesibles bajo solicitud por ser parte de un proyecto en curso. Para mayor información contactar a los profesores a cargo del grupo de investigación: Gonzalo Carvajal, César Silva o Juan Agüero, del Departamento de Electrónica de la Universidad Técnica Federico Santa María.

1.4. Organización del Informe

El contenido de esta tesis está organizado de la siguiente forma:

- Capitulo 2 presenta el desarrollo matemático de la formulación densa de MPC, así como fundamentos teóricos de diseño digital.
- Capitulo 3 propone una metodología para analizar los algoritmos de MPC y ADMM, y detectar su potencial de paralelización en hardware.
- Capitulo 4 muestra los resultados de una exploración del espacio de soluciones usando distintas estrategias disponibles en la herramienta de HLS.
- Capitulo 5 presenta el diseño de un controlador MPC para un caso de estudio, considerando la comunicación entre CPU y FPGA del sistema heterogéneo y la validando el controlador en hardware.
- Capitulo 6 concluye esta tesis con un resumen del trabajo realizado y los resultados obtenidos, así como enfoques a futuro para complementar el trabajo realizado.

2 | Antecedentes

El presente trabajo considera el diseño de controladores MPC con formulación implícita en sistemas heterogéneos usando HLS. En particular, se utiliza ADMM para resolver un problema QP formulado de forma densa. Este capítulo aborda primero la definición de MPC con formulación densa para evidenciar la relación entre los parámetros de la aplicación y el problema QP resultante. Luego, se presentan conceptos necesarios para analizar la arquitectura y desempeño de la implementación en hardware obtenida con HLS, lo cual se realiza en los capítulos posteriores.

2.1. Formulación del problema de control

La mayor parte de la formulación matemática presentada en esta sección fue tomada de trabajos previos de investigadores y estudiantes del Departamento de Electrónica de la Universidad Técnica Federico Santa María. Las partes basadas en trabajos previos cuentan con la debida autorización para ser reutilizadas en este documento, incorporándose algunos cambios y actualizaciones que resultan necesarios para el contexto de este trabajo.

2.1.1. Control Predictivo por Modelo

El Control Predictivo por Modelo (Model Predictive Control, o MPC) es una estrategia avanzada de control automático que utiliza un modelo matemático de la planta a controlar para predecir su comportamiento futuro, y con eso determinar el valor óptimo de las actuaciones en cada periodo de muestreo. MPC se plantea como un problema de optimización que considera estados y entradas del sistema en una secuencia de instantes futuros dentro de un horizonte de predicción h. Por ejemplo, la Figura 2.1 muestra el estado y la entrada para un horizonte de predicción de $h = 3$ muestras. en ella se ilustra la proyección de estados y entradas futuros, junto con una restricción para la entrada que se respeta en cada instante del horizonte.

Los objetivos de control se definen mediante el funcional de costos de la optimización, el cual pondera el error en los estados y la variación de las actuaciones (o entradas) en cada instante del horizonte de predicción. Además, el problema de optimización permite incluir restricciones lineales sobre las entradas y estados. Para sistemas linealizados e invariantes en el tiempo, la optimización asociada a MPC corresponde a un problema de Programación Cuadrática (Quadratic Programming, o QP) que se puede representar como:

$$\begin{aligned} \min_{\theta} \ & \frac{1}{2}\theta^{\mathrm{T}}\mathbf{Q}\theta + \mathbf{q}^{\mathrm{T}}\theta \\ \text{sujeto a } & \mathbf{G}\theta \leq \mathbf{g} \end{aligned} \tag{2.1}$$

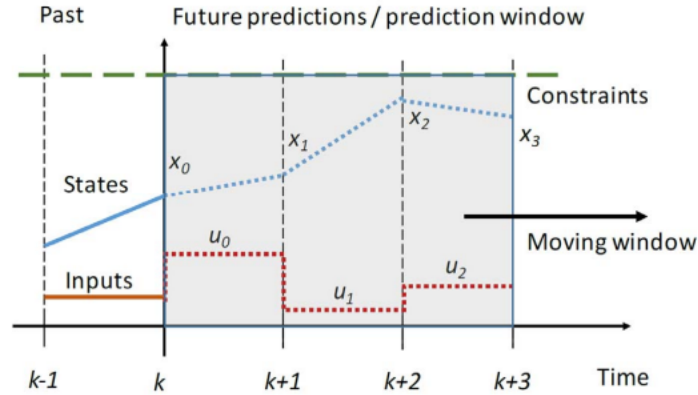


Figura 2.1: Control Predictivo por Modelo [10].

donde $\theta \in \mathbb{R}^N$ es el vector de variables de decisión, y la matriz $\mathbf{Q} \in \mathbb{R}^{N\times N}$ y el vector $\mathbf{q} \in \mathbb{R}^N$ regulan los efectos de dichas variables en el funcional de costo. Por ultimo, $\mathbf{G} \in \mathbb{R}^{M\times N}$ y $\mathbf{g} \in \mathbb{R}^M$ representan las restricciones del sistema [17].

Definición del problema de control

Considérese el siguiente sistema lineal discretizado e invariante en el tiempo:

$$\begin{aligned} x_{k+1} &= \mathbf{A}x_k + \mathbf{B}u_k \\ y_k &= \mathbf{C}x_k \end{aligned} \tag{2.2}$$

donde $x_k \in \mathbb{R}^n$ representa el valor de los n estados del sistema en el instante de muestreo k, y $u_k \in \mathbb{R}^m$ e $y_k \in \mathbb{R}^p$ son los correspondientes vectores de m entradas de control y p salidas, respectivamente. Además, $\mathbf{A}\in \mathbb{R}^{n\times n}$, $\mathbf{B}\in \mathbb{R}^{n\times m}$ y $\mathbf{C}\in \mathbb{R}^{p\times n}$ representan las dinámicas del sistema. $\mathbf{A}$ describe la influencia del estado actual en el siguiente estado, $\mathbf{B}$ describe la influencia de la entrada actual en el siguiente estado y $\mathbf{C}$ describe cómo es que el estado afecta a la salida. Inicialmente asumimos que todos los estados del sistema son medibles para el instante k.

Se requiere diseñar un controlador que sea capaz de mover la salida del sistema a un valor de referencia. Por tanto, se define el vector de error $e_k \in \mathbb{R}^p$ como la diferencia entre la referencia $r_k \in \mathbb{R}^p$ y la salida y_k:

$$e_k = r_k - y_k \tag{2.3}$$

Si se considera que la referencia es $r_k = 0$, i.e., se busca llevar la salida al origen, entonces obtenemos $e_k = -y_k$. Luego, para penalizar las desviaciones del punto de equilibrio, convenientemente se define una función de costo, o función objetivo, asociada al cuadrado del error:

$$J_k = y_k^{\mathrm{T}} y_k \tag{2.4}$$

Sustituyendo la expresión de y_k en (2.2) en (2.4), se obtiene:

$$J_k = x_k^{\mathrm{T}} \underbrace{C^{\mathrm{T}} C}_{\Omega} x_k = x_k^{\mathrm{T}} \Omega x_k \tag{2.5}$$

donde $\Omega \in \mathbb{R}^{n \times n}$ es una matriz de pesos que refleja la importancia relativa de cada estado en la determinación de la salida del sistema. $\Omega = C^{\mathrm{T}} C$ es una forma inicial de definir esta matriz de pesos y el usuario puede ajustar la relación entre los pesos de los estados. Adicionalmente, al considerar aspectos prácticos como el desgaste de los actuadores, restricciones físicas de las componentes del sistema, la seguridad de operación, el consumo energético, etc., es conveniente agregar un término cuadrático que penalice cambios drásticos en las entradas de control u_k, quedando la función de costo como:

$$J_k = x_k^{\mathrm{T}} \Omega x_k + u_k^{\mathrm{T}} \Gamma u_k \tag{2.6}$$

donde la matriz $\Gamma \in \mathbb{R}^{m \times m}$ representa factores de penalización definidos a tiempo de diseño. Para el caso de interés de múltiples entradas, podemos considerar por simplicidad una matriz diagonal $\Gamma = v \cdot I_m$, donde I_m es la matriz identidad de tamaño m y v es un escalar que pondera la entrada.

Dado que el control predictivo requiere calcular el costo obtenido para las futuras entradas dentro del horizonte de predicción, se generaliza el funcional de costo como la suma de los costos individuales dentro del horizonte de predicción como:

$$J(x_k, u_k) = \sum_{k=0}^{h-1} \left(x_k^{\mathrm{T}} \Omega x_k + u_k^{\mathrm{T}} \Gamma u_k \right) + x_h^{\mathrm{T}} \Omega_h x_h \tag{2.7}$$

donde el término $x_h^{\mathrm{T}} \Omega_h x_h$ representa un estado terminal con su correspondiente matriz de penalización $\Omega_h \in \mathbb{R}^{n \times n}$, el cual se introduce para aproximar el comportamiento de un lazo MPC con horizonte infinito (lo cual no se puede implementar en la práctica) mediante un lazo con horizonte finito. En general, la expresión terminal influencia la calidad del control logrado, la estabilidad del sistema, y la complejidad numérica del problema. La correcta selección de esta expresión es un problema no trivial, el cual se resuelve mediante la ecuación de Ricatti.

Para determinar la acción de control óptima a lo largo del horizonte, es necesario encontrar la secuencia de entradas $\vec{u}^* = [u_0^{*\mathrm{T}} \quad u_1^{*\mathrm{T}} \ldots \quad u_{h-1}^{*\mathrm{T}}]^{\mathrm{T}} \in \mathbb{R}^{(m \cdot h)}$ que minimiza el funcional de costo a lo largo del horizonte de predicción. Formalmente, y suponiendo que se dispone de una medición o estimación del vector de estados actual x_k, el problema QP se describe como:

$$\begin{aligned} \vec{u}^* &= \arg \min_{\vec{u}} J(x_k, u_k) \\ \text{sujeto a} & \\ x_k &= x_0 \\ x_{k+1} &= \mathbf{A} x_k + \mathbf{B} u_k \\ u^{\min} &\leq \mathbf{H} u_{k+i} \leq u^{\text{máx}} \\ x^{\min} &\leq \mathbf{I} x_{k+i} \leq x^{\text{máx}} \end{aligned} \tag{2.8}$$

donde x_0 corresponde al estado inicial cuyo valor al instante k puede ser medido o estimado mediante un observador, $u^{\min}$ y $u^{\max} \in \mathbb{R}^a$ especifican límites para los valores mínimos y máximos de combinaciones lineales de las entradas, y $x^{\min}$ y $x^{\max} \in \mathbb{R}^b$ especifican límites para los valores mínimos y máximos de combinaciones lineales de los estado. Las matrices

$\mathbf{H} \in \mathbb{R}^{a\times m}$ y $\mathbf{I} \in \mathbb{R}^{b\times n}$ ponderan las entradas y estados respectivamente, y corresponden a la matriz identidad en caso de restricciones de caja.

La formulación anterior plantea un problema de control en lazo abierto, ya que la secuencia de entradas óptimas calculadas al instante k no considera alteraciones o perturbaciones que puedan ocurrir al momento de ejecutar las acciones futuras. En el caso de ocurrir perturbaciones, la aplicación de las entradas precalculadas no generarán la trayectoria óptima en los estados y salidas, generando una degradación en el desempeño o incluso inestabilidad del sistema. Además, como se mencionó previamente, la solución encontrada es inherentemente subóptima debido a que el funcional de costo es solo una aproximación del problema para un horizonte infinito. Para tratar ambas limitaciones, se plante el concepto de *receding horizon*, el cual plantea que *" Un controlador subóptimo de horizonte infinito puede ser diseñado resolviendo repetidamente problemas de control óptimo de tiempo finito en una forma de horizonte en retroceso (...)"* [18].

Bajo el enfoque de receding horizon, se resuelve un problema de optimización de horizonte finito en cada tiempo de muestreo. Una vez determinada la secuencia de entradas $\vec{u}^*$, solo se aplica el primer elemento u_0^* al sistema, descartando el resto de valores de la secuencia óptima. Luego, en el siguiente intervalo de muestreo se determinan los nuevos estados obtenidos a partir de la entrada aplicada y se define un nuevo problema de optimización con un nuevo valor inicial, lo cual introduce un feedback indirecto, cerrando el lazo de control evitando que las diferencias entre el estado medido y el predicho se propaguen a instantes futuros.

Los pasos para aplicar MPC en cada instante de muestreo se resumen a continuación:

1. En el instante de muestreo k, medir (o estimar) el estado de la planta, x_0.
2. Plantear el problema QP a partir del estado medido de la planta.
3. Calcular el nuevo funcional de costo y resolver para determinar la secuencia óptima de entradas que lo minimizan, $\vec{u}^*$.
4. Aplicar en la entrada el primer elemento de la secuencia de control óptima, u_0^*.
5. Repetir para el siguiente instante de muestreo, $k+1$.

Matemáticamente, el problema QP a resolver en cada período de muestreo se puede plantear de distintas formas según la elección de las variables de decisión. Cuando se elije como única variable de decisión la entrada u_k, entonces se obtiene la forma denominada *Formulación Densa*, y si se eligen como variable de decisión tanto la entrada u_k como el estado x_k se obtiene la forma denominada *Formulación Sparse*. La forma de representar el problema QP modifica los parámetros de tamaño de las matrices asociadas de la siguiente forma:

$$\begin{array}{llcl} \text{Densa:} & N = h \cdot m & \text{y} & M = 2h(a+b) \\ \text{Sparse:} & N = h \cdot m + (h+1)n & \text{y} & M = 2h(a+b) + b + 2(h+1)n \end{array}$$

Si bien la formulación *sparse* resulta en matrices de mayor dimensión, estas matrices tienen muchos elementos con valor 0. La literatura muestra varios trabajos que explotan las características de la forma *sparse* para reducir los requerimientos de cómputo de la formulación *sparse*, usando estructuras de datos comprimidas que evitan almacenar en memoria y realizar cálculos con los elementos de valor 0 [14]. En muchos casos, se ha reportado que, a medida que aumenta la dimensión del problema, la formulación *sparse* del problema QP puede reducir

los tiempos de resolución comparado a la formulación densa equivalente, la cual tiene matrices de menor tamaño pero con valores no nulos.

En el presente trabajo solo se considera la formulación densa del problema QP, ya que los potenciales beneficios de la formulación *sparse* requieren el uso de estructuras de datos irregulares que, de acuerdo a evaluaciones preliminares, no son directamente explotables mediante HLS para su implementación eficiente en FPGAs.

A continuación, se presenta el desarrollo matemático de la formulación densa de MPC con seguimiento de referencia y restricciones de desigualdad. Si bien no es necesario entender el desarrollo matemático en detalle, a partir de la formulación del problema de control es posible derivar parámetros relevantes para dimensionar el costo computacional de los algoritmos.

2.1.2. Formulación densa de MPC

Se considera como variable de decisión la entrada -o actuación- del sistema [1]. Para un horizonte de predicción h se definen los siguientes vectores:

$$\vec{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ \vdots \\ u_{h-1} \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_h \end{bmatrix} \tag{2.9}$$

donde $\vec{u} \in \mathbb{R}^{(m \cdot h)}$ y $\vec{x} \in \mathbb{R}^{(n \cdot h)}$. Con estas definiciones podemos reescribir el sistema 2.2 en términos de $\vec{u}$ y $\vec{x}$ como:

$$\vec{x} = \underbrace{\begin{bmatrix} \mathbf{A} \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^{h-1} \\ \mathbf{A}^h \end{bmatrix}}_{\mathbf{D}} x_0 + \underbrace{\begin{bmatrix} \mathbf{B} & 0 & 0 & \dots & 0 & 0 \\ \mathbf{AB} & \mathbf{B} & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{A}^{h-2}\mathbf{B} & \mathbf{A}^{h-3}\mathbf{B} & \dots & \dots & \ddots & 0 \\ \mathbf{A}^{h-1}\mathbf{B} & \mathbf{A}^{h-2}\mathbf{B} & \dots & \dots & \mathbf{AB} & \mathbf{B} \end{bmatrix}}_{\mathbf{E}} \vec{u} \qquad = \mathbf{D}x_0 + \mathbf{E}\vec{u} \tag{2.10}$$

donde $\mathbf{D} \in \mathbb{R}^{(n \cdot h) \times n}$ y $\mathbf{E} \in \mathbb{R}^{(n \cdot h) \times (m \cdot h)}$. También se expande el funcional de costo 2.7 para dejarlo en términos de x_0 y $\vec{u}$:

$$J(x_0, \vec{u}) = x_0^{\mathrm{T}} \Omega x_0 + \vec{x}^{\mathrm{T}} \underbrace{\begin{bmatrix} \Omega & 0 & \dots & 0 \\ 0 & \Omega & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \Omega_h \end{bmatrix}}_{\mathbf{K}} \vec{x} + \vec{u}^{\mathrm{T}} \underbrace{\begin{bmatrix} \Gamma & 0 & \dots & 0 \\ 0 & \Gamma & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & \Gamma \end{bmatrix}}_{\mathbf{L}} \vec{u} = x_0^{\mathrm{T}} \Omega x_0 + \vec{x}^{\mathrm{T}} \mathbf{K} \vec{x} + \vec{u}^{\mathrm{T}} \mathbf{L} \vec{u} \tag{2.11}$$

donde $\mathbf{K} \in \mathbb{R}^{(n \cdot h) \times (n \cdot h)}$ y $\mathbf{L} \in \mathbb{R}^{(m \cdot h) \times (m \cdot N)}$. Luego, concentrando (2.10) y (2.11) se obtiene:

$$\begin{aligned} J(x_0,\vec{u}) &= x_0^{\mathrm{T}}\Omega x_0 + (\mathbf{D}x_0 + \mathbf{E}\vec{u})^{\mathrm{T}}\mathbf{K}(\mathbf{D}x_0 + \mathbf{E}\vec{u}) + \vec{u}^{\mathrm{T}}\mathbf{L}\vec{u} \\ &= x_0^{\mathrm{T}}\Omega x_0 + x_0^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{D}x_0 + x_0^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{E}\vec{u} + \vec{u}^{\mathrm{T}}\mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{D}x_0 + \vec{u}^{\mathrm{T}}\mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{E}\vec{u} + \vec{u}^{\mathrm{T}}\mathbf{L}\vec{u} \end{aligned} \tag{2.12}$$

donde los términos $x_0^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{E}\vec{u}$ y $\vec{u}^{\mathrm{T}}\mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{D}x_0$ son iguales ya que uno es el transpuesto del otro y cada uno es de dimensiones 1×1, por lo que se puede transponer el segundo termino y sumarlo al primero, resultando:

$$J(x_0,\vec{u}) = x_0(\Omega + \mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{D})x_0 + \vec{u}^{\mathrm{T}}(L + \mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{E})\vec{u} + 2x_0^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{E}\vec{u} \tag{2.13}$$

Se define $\mathbf{Q} \in \mathbb{R}^{(m\cdot h)\times(m\cdot h)}$ y $\mathbf{q}^{\mathrm{T}} \in \mathbb{R}^{1\times(m\cdot h)}$ como:

$$\frac{1}{2}\mathbf{Q} = \mathbf{L} + \mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{E} \tag{2.14}$$

$$\mathbf{q}^{\mathrm{T}} = 2x_0^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{E} \tag{2.15}$$

Reemplazando (2.14) y (2.15) en (2.13) se obtiene el funcional de costo:

$$J(x_0,\vec{u}) = \frac{1}{2}\vec{u}^{\mathrm{T}}\mathbf{Q}\vec{u} + \mathbf{q}^{\mathrm{T}}\vec{u} \tag{2.16}$$

Nótese que el termino $x_0(\Omega + \mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{D})x_0$ no aparece en el funcional de costo (2.16) ya que para cada instante de muestreo su valor es constante y por lo tanto no va a afectar la búsqueda de un $\vec{u}$ que minimice el costo.

Finalmente se re-escriben el resto de las restricciones en términos de $\vec{u}$ y $\vec{x}$ como se describe a continuación:

a) Limitaciones en la señal de entrada:

$$\begin{aligned} u^{\min} &\leq \mathbf{H}u_k \leq u^{\text{máx}} \\ \vec{u}^{\min} &\leq \mathbf{V}\vec{u} \leq \vec{u}^{\text{máx}} \end{aligned} \tag{2.17}$$

donde

$$\vec{u}^{\min} = \begin{bmatrix} u^{\min} \\ \vdots \\ u^{\min} \end{bmatrix} \qquad \mathbf{V} = \begin{bmatrix} \mathbf{H} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \mathbf{H} \end{bmatrix} \qquad \vec{u}^{\text{máx}} = \begin{bmatrix} u^{\text{máx}} \\ \vdots \\ u^{\text{máx}} \end{bmatrix}$$

tal que $\vec{u}^{\min}$ y $\vec{u}^{\max} \in \mathbb{R}^{a\cdot h}$, y $\mathbf{V} \in \mathbb{R}^{(a\cdot h)\times(m\cdot h)}$.

b) Limitaciones en el estado:

$$\begin{aligned} x^{\min} &\leq \mathbf{I}\vec{x} \leq x^{\text{máx}} \\ \vec{x}^{\min} &\leq \mathbf{W}\vec{x} \leq \vec{x}^{\text{máx}} \\ \vec{x}^{\min} &\leq \mathbf{W}\left(\mathbf{D}x_0 + \mathbf{E}\vec{u}\right) \leq \vec{x}^{\text{máx}} \\ \left(\vec{x}^{\min} - \mathbf{W}\mathbf{D}x_0\right) &\leq \mathbf{W}\mathbf{E}\vec{u} \leq \left(\vec{x}^{\text{máx}} - \mathbf{W}\mathbf{D}x_0\right) \end{aligned} \tag{2.18}$$

$$\begin{aligned} \mathbf{W}\mathbf{E}\vec{u} &\leq \left(\vec{x}^{\text{máx}} - \mathbf{W}\cdot\mathbf{D}x_0\right) \\ -\mathbf{W}\mathbf{E}\vec{u} &\leq -\left(\vec{x}^{\min} - \mathbf{W}\cdot\mathbf{D}x_0\right) \end{aligned} \tag{2.19}$$

donde

$$\vec{x}^{\min} = \begin{bmatrix} x^{\min} \\ \vdots \\ x^{\min} \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} \mathbf{I} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \mathbf{I} \end{bmatrix} \qquad \vec{x}^{\text{máx}} = \begin{bmatrix} x^{\text{máx}} \\ \vdots \\ x^{\text{máx}} \end{bmatrix}$$

tal que $\vec{x}^{\min}$ y $\vec{x}^{\max} \in \mathbb{R}^{(b \cdot h)}$, y $\mathbf{W} \in \mathbb{R}^{(b \cdot h) \times (n \cdot h)}$.

en forma matricial, todas las restricciones se pueden escribir como:

$$\underbrace{\begin{bmatrix} \mathbf{V} \\ \\ -\mathbf{V} \\ \\ \mathbf{WE} \\ \\ -\mathbf{WE} \end{bmatrix}}_{\mathbf{G}} \vec{u} \leq \underbrace{\begin{bmatrix} \vec{u}^{\text{máx}} \\ \\ -\vec{u}^{\min} \\ \\ \vec{x}^{\text{máx}} - \mathbf{WD}x_0 \\ \\ -\vec{x}^{\min} + \mathbf{WD}x_0 \end{bmatrix}}_{\mathbf{g}} \tag{2.20}$$

donde $\mathbf{G} \in \mathbb{R}^{(2(a+b)h) \times (m \cdot h)}$ y $\mathbf{g} \in \mathbb{R}^{(2(a+b)h)}$.

Nótese que las restricciones en (2.8) se transforman en restricciones lineales en $\vec{u}$, con lo cual se define el problema de optimización QP como :

$$\begin{aligned} \vec{u}^* &= \min_{\vec{u}} J(x_0, \vec{u}) \\ \text{sujeto a} & \\ \mathbf{G}\vec{u} &\leq \mathbf{g} \end{aligned} \tag{2.21}$$

Seguimiento de referencia

Para el seguimiento de una referencia $r \neq 0$ en la salida se requiere que $y_\infty \to r$, donde y_∞ representa el valor de la salida en estado estacionario. Para determinar el valor de los estados y entradas que permiten llevar la salida al valor de referencia, se reescribe la expresión en estado estacionario para el sistema en 2.2 como:

$$\begin{aligned} x_\infty &= \mathbf{A}x_\infty + \mathbf{B}u_\infty \\ y_\infty &= \mathbf{C}^* x_\infty \end{aligned} \tag{2.22}$$

donde $C^* \in \mathbb{R}^{p \times m}$ corresponde a la parte de C asociada a las salidas que tienen referencia. Entonces se despeja x_∞ y u_∞:

$$\underbrace{\begin{bmatrix} I_n - A & -B \\ C^* & 0 \end{bmatrix}}_{T} \begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = \begin{bmatrix} 0 \\ r \end{bmatrix} \tag{2.23}$$

$$\begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = T^{-1} \begin{bmatrix} 0 \\ r \end{bmatrix} \tag{2.24}$$

donde $T \in \mathbb{R}^{(n+p) \times (n+m)}$.

Al mismo tiempo es posible considerar perturbaciones medibles $d_k \in \mathbb{R}^d$, agregándolas al modelo del sistema (2.2)

$$\begin{aligned} x_{k+1} &= \mathbf{A}x_k + \mathbf{B}u_k + \mathbf{B_p}d_k \\ y_k &= \mathbf{C}^* x_k \end{aligned} \tag{2.25}$$

donde $B_p \in \mathbb{R}^{n \times d}$ representa el efecto de las perturbaciones medibles en el estado, de forma que reescribimos (2.24):

$$\begin{bmatrix} x_\infty \\ u_\infty \end{bmatrix} = T^{-1} \begin{bmatrix} B_p \cdot d_k \\ r \end{bmatrix} \tag{2.26}$$

Luego, se realiza el siguiente cambio de variables:

$$\begin{aligned} \tilde{x} &= x - x_\infty \\ \tilde{u} &= u - u_\infty \end{aligned} \tag{2.27}$$

donde $\tilde{x}$ y $\tilde{u}$ representan la desviación de los estados y entradas de los valores que permiten llevar la salida a su valor de referencia en estado estacionario. Considerando que las variables de desviación satisfacen el mismo modelo dinámico que las variables originales, el funcional de costo en función de las variables de desviación queda dado por:

$$J_N(\tilde{x}_0, \vec{\tilde{u}}) = \frac{1}{2} \vec{\tilde{u}}^{\mathrm{T}} \mathbf{Q} \vec{\tilde{u}} + \tilde{\mathbf{q}}^{\mathrm{T}} \vec{\tilde{u}} \tag{2.28}$$

donde:

$$\begin{aligned} \frac{1}{2}\mathbf{Q} &= \mathbf{L} + \mathbf{E}^{\mathrm{T}}\mathbf{K}\mathbf{E} \\ \tilde{\mathbf{q}}^{\mathrm{T}} &= 2\tilde{x}_0{}^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{K}\mathbf{E} \end{aligned} \tag{2.29}$$

Para encontrar las restricciones de esta nueva función de costo hay que aplicar el cambio de variables (2.27) a las restricciones originales (2.8). Se llega a:

a) Limitaciones en las entradas:

$$\begin{aligned} \vec{u}^{\min} &\leq \mathbf{V}\vec{u} \leq \vec{u}^{\max} \\ \vec{u}^{\min} &\leq \mathbf{V}(\vec{\tilde{u}} + \vec{u}_\infty) \leq \vec{u}^{\max} \\ \vec{u}^{\min} - \mathbf{V}\vec{u}_\infty &\leq \mathbf{V}\vec{\tilde{u}} \leq \vec{u}^{\text{máx}} - \mathbf{V}\vec{u}_\infty \end{aligned} \tag{2.30}$$

b) Limitaciones en los estados:

$$\begin{aligned} \vec{x}^{\min} &\leq \mathbf{W}\vec{x} \leq \vec{x}^{\text{máx}} \\ \vec{x}^{\min} &\leq \mathbf{W}(\vec{\tilde{x}} + \vec{x}_\infty) \leq \vec{x}^{\max} \\ \vec{x}^{\min} &\leq \mathbf{W}(\mathbf{D}\tilde{x}_0 + \mathbf{E}\vec{\tilde{u}} + \vec{x}_\infty) \leq \vec{x}^{\max} \\ \vec{x}^{\min} - \mathbf{W}(\mathbf{D}\tilde{x}_0 + \vec{x}_\infty) &\leq \mathbf{W}\mathbf{E}\vec{u} \leq \vec{x}^{\text{máx}} - \mathbf{W}(\mathbf{D}\tilde{x}_0 + \vec{x}_\infty) \end{aligned} \tag{2.31}$$

$$\begin{aligned} \mathbf{W}\mathbf{E}\vec{u} &\leq \vec{x}^{\text{máx}} - \mathbf{W}\vec{x}_\infty - \mathbf{W}\mathbf{D}(x_0 - x_\infty) \\ -\mathbf{W}\mathbf{E}\vec{u} &\leq -\vec{x}^{\min} + \mathbf{W}\vec{x}_\infty + \mathbf{W}\mathbf{D}(x_0 - x_\infty) \end{aligned} \tag{2.32}$$

Finalmente se pueden representar las restricciones en estado estacionario:

$$\underbrace{\begin{bmatrix} \mathbf{V} \\ -\mathbf{V} \\ \mathbf{WE} \\ -\mathbf{WE} \end{bmatrix}}_{\tilde{\mathbf{G}}} \vec{\tilde{u}} \leq \underbrace{\begin{bmatrix} \vec{u}^{\text{máx}} - \mathbf{V}\vec{u}_\infty \\ -\vec{u}^{\text{min}} + \mathbf{V}\vec{u}_\infty \\ \vec{x}^{\text{máx}} - \mathbf{W}\vec{x}_\infty - \mathbf{WD}(x_0 - x_\infty) \\ -\vec{x}^{\text{min}} + \mathbf{W}\vec{x}_\infty + \mathbf{WD}(x_0 - x_\infty) \end{bmatrix}}_{\tilde{\mathbf{g}}} \tag{2.33}$$

Planteamiento final de problema QP

Finalmente, el problema QP en forma densa, con seguimiento de referencia r distinta a 0 y con todas las restricciones en forma de desigualdad, queda planteado como encontrar $\vec{\tilde{u}}^*$ tal que minimice la función de costo $J_N(\tilde{x}_0, \vec{\tilde{u}})$:

$$J_N(\tilde{x}_0, \vec{\tilde{u}}) = \frac{1}{2}\vec{\tilde{u}}^{\mathrm{T}}\mathbf{Q}\vec{\tilde{u}} + \tilde{\mathbf{q}}^{\mathrm{T}}\vec{\tilde{u}}$$

donde:

$$\frac{1}{2}\mathbf{Q} = \mathbf{L} + \mathbf{E}^{\mathrm{T}}\mathbf{KE}$$

$$\tilde{\mathbf{q}}^{\mathrm{T}} = 2(x_0 - x_\infty)^{\mathrm{T}}\mathbf{D}^{\mathrm{T}}\mathbf{KE}$$

sujeto a

$$\tilde{\mathbf{G}}\vec{\tilde{u}} \leq \tilde{\mathbf{g}}$$

Una vez encontrado el vector $\vec{\tilde{u}}^*$ es necesario revertir el cambio de variables (2.27) que se hizo para seguir la referencia y así obtener las entradas que son apropiadas para aplicar en la planta. Esto se hace de la siguiente manera:

$$\vec{u}^* = \vec{\tilde{u}}^* + \vec{u_\infty} \tag{2.34}$$

El resultado final de MPC es el vector $\vec{u}^*$ que contiene las entradas para h instantes de tiempo que son necesarias para llevar la salida de la planta a la referencia deseada. Solo se aplica la primera entrada u_0^* a la planta y en el siguiente instante de muestreo se repite el proceso.

Para la resolución del problema QP se han desarrollado múltiples métodos, que pueden agruparse en tres familias de algoritmos: Interior-Point [19, 20]; Active-Set [21] y First-Order [17]. Estos algoritmos se caracterizan por hacer un uso extensivo de operaciones de álgebra lineal. En este trabajo nos enfocaremos exclusivamente en la implementación de ADMM, que es un método de First-Order.

Algoritmo 1 ADMM

1: **Input:** q, g ▷ Depende del estado medido
2: $R^{-1} = (Q + \rho G^{\mathrm{T}} G)^{-1}$ ▷ Constante para el sistema
3: $t_0,\ z_0,\ u_0 \leftarrow t_N,\ z_N,\ u_N$ ▷ Arranque en caliente
4: **for** $k = 0$ to $IT_{ADMM} - 1$ **do**
5: $\quad v_k = z_k - g + u_k$ ▷ Variable auxiliar
6: $\quad t_{k+1} = R^{-1}(-\rho G^{\mathrm{T}} v_k - q)$
7: $\quad z_{k+1} = max\{0, -Gt_{k+1} - u_k + g\}$
8: $\quad u_{k+1} = u_k + Gt_{k+1} + z_{k+1} - g$
9: **end for**
10: **return** t_N

2.1.3. *Alternating Direction Method of Multipliers* (ADMM)

Para resolver el problema QP empleamos el método ADMM. Se trata de un método que ha ganado notoriedad en los últimos años por tener iteraciones simples, donde la operación de álgebra lineal más compleja es la multiplicación matriz-vector [5, 15]. Las operaciones aritméticas requeridas para implementar ADMM exponen un nivel de independencia que puede ser explotado mediante cómputo paralelo. Debido a esto, ADMM es ampliamente utilizado en aplicaciones MPC que requieren baja latencia [17, 22].

El algoritmo ADMM toma la forma de un procedimiento de descomposición-coordinación, donde las soluciones a pequeños subproblemas locales se coordinan para encontrar una solución a un problema global más grande [23]. Para utilizar ADMM, se reformulan las restricciones como igualdades y se agrega una variable de *slack* z al problema de optimización [17], llegando a la siguiente representación:

$$\begin{aligned} \min_{t,z} \quad & \frac{1}{2} t^{\mathrm{T}} \mathbf{Q} t + \mathbf{q}^{\mathrm{T}} t + I_{\mathbb{Z}}(z) \\ \text{sujeto a} \quad & \mathbf{G} t + \mathbf{I} z = \mathbf{g} \end{aligned} \tag{2.35}$$

donde $\mathrm{I}_{\mathbb{Z}}(z)$ es la función indicatriz de $\mathbb{Z} = \{z : z \geq 0\}$, es decir,

$$\mathrm{I}_{\mathbb{Z}}(z) = \begin{cases} 0 & \text{si } z \in \mathbb{Z} \\ \infty & \text{si } z \notin \mathbb{Z}. \end{cases}$$

Los detalles de la formulación matemática de ADMM están fuera del alcance de este trabajo. A partir del trabajo de tesis reportado en [13] se cuenta con una versión probada de ADMM, cuyo pseudocódigo se presenta en el Algoritmo 1. Para más detalles referirse a [17].

El método ADMM se ejecuta de forma iterativa. Para ajustar la convergencia del algoritmo, el usuario cuenta con dos parámetros de diseño: la cantidad de iteraciones IT_{ADMM} y el tamaño del paso ρ. La elección de estos parámetros determina qué tanto se acercará la ejecución de ADMM al óptimo real del problema. El valor de ρ no influye en el costo computacional de ejecutar ADMM. En cambio, el tiempo de ejecución crece linealmente con el número de iteraciones. Y dado que, a mayor número de iteraciones hay mayor probabilidad de acercarse al óptimo, la elección del parámetro IT_{ADMM} conlleva un compromiso entre calidad del control y tiempo de ejecución.

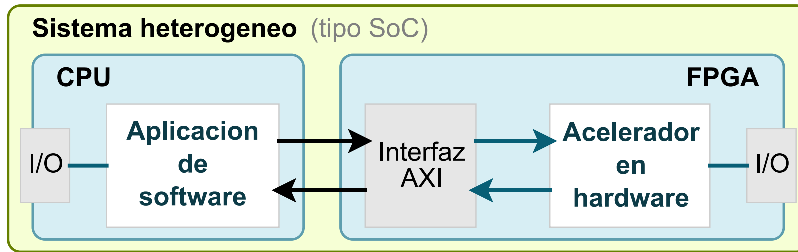


Figura 2.2: Diagrama de bloques de un sistema heterogéneo, con aceleración en hardware.

Para este trabajo, consideramos que ρ y IT_{ADMM} son constantes durante la operación de la planta y que sus valores ya están definidos por el usuario. Además, cabe destacar que los valores finales de los vectores $\mathbf{t}$, $\mathbf{z}$ y $\mathbf{u}$ se conservan y se utilizan como valores iniciales del algoritmo de ADMM en la siguiente iteración de MPC, en lo que se denomina *warm-start*.

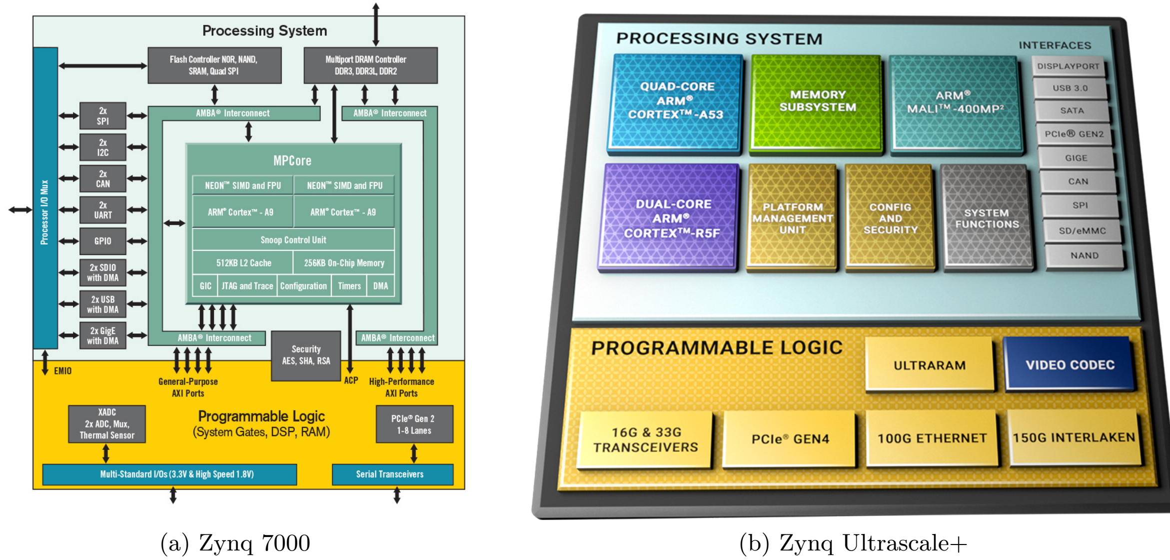
2.2. Implementación de aceleradores de MPC en sistemas heterogéneos

En esta sección se presentan tres aspectos de este trabajo que son conceptualmente independientes, pero que están todos asociados al diseño y evaluación de la implementación de aceleradores de hardware para MPC. Primero, abordamos conceptos relacionados con el uso de arquitecturas heterogéneas. Posteriormente, establecemos métricas para evaluar el desempeño de los controladores implementados. Por último, se presenta la terminología que utilizaremos para analizar el diseño de aceleradores mediante HLS.

2.2.1. Sistemas heterogéneos

Entendemos por sistema de computación heterogénea o sistema heterogéneo (en inglés, Heterogeneous System) un sistema que está compuesto por sub-sistemas de procesamiento de naturalezas distintas. Puede ser un procesador que incorpora dos o más núcleos de distinta microarquitectura, o bien un procesador de propósito general convencional junto con otro tipo de hardware para cómputo, como GPU, lógica programable o algun otro tipo de acelerador específico. En este trabajo consideraremos que un sistema heterogéneo es la asociación de un procesador convencional (CPU) con lógica programable (FPGA). El uso de estos sistema heterogéneos responde a un escenario general en que una parte del procesamiento del control se ejecuta en una CPU y otra parte se acelera en la FPGA, como se presenta en la Figura 2.2. En el presente trabajo se utilizará una interfaz con protocolo AXI [24] para la comunicación entre la CPU y el acelerador, tal como se ilustra en el diagrama.

Es importante destacar que es decisión del usuario cómo se distribuye el procesamiento entre CPU y FPGA. Además, durante el diseño será fundamental considerar la conexión entre ambas partes, ya que la latencia de esta comunicación afectará la frecuencia de muestreo del controlador. Por esta razón, en este trabajo consideraremos dispositivos modernos que incluyen tanto procesador como FPGA dentro del mismo circuito integrado. En particular, utilizaremos dispositivos de la familia Zynq del fabricante AMD-Xilinx, conocidos comercialmente como System-on-Chip (SoC) [25]. Al integrar la CPU junto a la FPGA estos dispositivos ofrecen una menor latencia de comunicación dentro del sistema heterogéneo, comparado con un sistema



(a) Zynq 7000 (b) Zynq Ultrascale+

Figura 2.3: Diagramas de bloques de chips tipo SoC de AMD-Xilinx.

con dos dispositivos conectados *off-chip*. Reducir la latencia de comunicación significa, a su vez, mayor flexibilidad para que el usuario distribuya el procesamiento de sub-rutinas entre CPU y FPGA, respetando el periodo de muestreo objetivo. La Figura 2.3 muestra diagramas de las arquitecturas de dos dispositivos SoC de gama baja (a) y media (b), donde se aprecia la integración de múltiples CPUs con lógica programable (FPGA). Además, en el recuadro 2.3a se detalla la arquitectura de la comunicación entre CPU y FPGA, mediante interfaces AXI. En este trabajo utilizaremos una tarjeta de desarrollo con un SoC Zynq Ultrascale+, como el del recuadro 2.3b.

2.2.2. Caracterización de desempeño

Antes de poder establecer las métricas para caracterizar el desempeño, es necesario introducir terminología de diseño de sistemas digitales que usaremos en el resto del trabajo:

- **Tarea**: unidad de comportamiento, correspondiente a la invocación de una función o a un subconjunto claramente definido de operaciones del algoritmo que juntas realizan una acción específica.
- **Implementación**: se refiere al software o hardware que realiza una tarea específica. También puede referirse al proceso de diseño que resulta en dicho software o hardware. En particular, en el contexto de diseño digital, se llama implementación al conjunto de procesos posteriores a la síntesis lógica que generan el diseño final de hardware (para configurar una FPGA o fabricar como ASIC).
- **Latencia**: tiempo total de procesamiento de una operación, tarea o algoritmo en su implementación en software o hardware, desde el instante en que inicia el procesamiento de un conjunto de entradas hasta el instante en que está disponible el conjunto de salidas asociado a dichas entradas.
- **Throughput**: frecuencia con la que se procesan datos en un elemento de hardware, desde el instante en que un conjunto de salidas está disponible hasta el instante que el

siguiente conjunto de salidas está disponible.

- **Instancia**: copia física de un elemento de hardware.

Para este trabajo, consideramos que la principal métrica de desempeño es la latencia requerida para el cálculo de la acción de control en cada intervalo de muestreo. En general, se debe cumplir que:

$$L_{MPC} < T_{Muestreo} \tag{2.36}$$

donde L_{MPC} es la latencia del lazo MPC para el cálculo de la actuación óptima y $T_{Muestreo}$ es el periodo de muestreo, el cual se deriva de los requerimientos de la aplicación. Por ejemplo, para controlar el filtro de salida de un DER se requiere una frecuencia de muestreo superior a 5kHz, lo cual impone una latencia máxima de 200μs para calcular la acción de control [13].

La latencia del lazo de control se puede dividir de acuerdo a los pasos detallados en la Sección 2.1:

$$L_{MPC} = L_{Sensado} + L_{Formulacion_Resolucion} + L_{Actuacion} \tag{2.37}$$

donde $L_{Sensado}$ es la latencia del sensado y acondicionamiento del estado de la planta, $L_{Actuacion}$ es la latencia de aplicar la actuación, y $L_{Formulacion_Resolucion}$ es la latencia del procesamiento de la formulación y la resolución del problema QP. Si, además, consideramos que la formulación y resolución del problema QP puede estar repartida entre CPU (software) y FPGA (hardware), con la correspondiente comunicación, podemos expresar la latencia total como:

$$L_{MPC} = L_{Sensado} + L_{SW} + L_{Com} + L_{HW} + L_{Actuacion} \tag{2.38}$$

donde L_{SW} es la latencia del procesamiento en CPU, L_{HW} es la latencia del procesamiento en FPGA y L_{Com} es la latencia de la comunicación entre CPU y FPGA.

Este trabajo se enfoca en implementar la formulación y resolución del problema QP. En particular, nos interesa diseñar el acelerador de hardware para cumplir la restricción de latencia:

$$L_{HW} < T_{Muestreo} - L_{Sensado} - L_{SW} - L_{Com} - L_{Actuacion} \tag{2.39}$$

Como se menciona en la Sección 1.3, el trabajo se enfoca en los aspectos de cómputo y no considera la implementación física en una planta real, por lo que, por simplicidad, omitiremos los retardos asociados a la captura de señales desde sensores y en envío de señales hacia los actuadores, considerando $L_{Sensado} = L_{Actuacion} = 0$. Se asume entonces que las entradas están disponibles en un *buffer* al momento de iniciar el intervalo, y la ejecución termina al escribir el resultado en otro *buffer*. Por simplicidad, en adelante usaremos el término ”controlador” para referirnos sólo al procesamiento de la formulación y resolución del problema QP que implementaremos.

Para efectos de comparar dos implementaciones del controlador, nos referiremos a la aceleración, como la diferencia de latencia entre una implementación de referencia y otra implementación. Asimismo, hablaremos de la aceleración porcentual, expresada como el porcentaje de dicha diferencia respecto a la latencia de referencia:

$$Accel = 100 \cdot \frac{L_1 - L_2}{L_1} \% \tag{2.40}$$

Por otra parte, asumiendo que existen distintas alternativas que permiten cumplir con el requerimiento de latencia, nos interesa comparar implementaciones del controlador en términos

de la relación costo-beneficio, i.e. la relación entre uso de recursos y frecuencia de muestreo del controlador. En el contexto de FPGA, los recursos de interés son los bloques *Digital Signal Processors* (DSP), los *Flip-Flops* (FF), las *Lookup Tables* (LUT) y las *Block RAM* (BRAM). En particular, para efectos de utilizar operaciones en punto flotante, es más relevante el uso de bloques DSP, en valor absoluto o como porcentaje de las unidades disponibles en un dispositivo objetivo. Utilizaremos la relación entre frecuencia de muestreo y uso de bloques DSP para comparar el costo-beneficio de los aceleradores implementados.

2.2.3. Síntesis de Alto Nivel (HLS)

Para el diseño del acelerador que permitirá reducir el tiempo de procesamiento de MPC, consideramos un flujo de diseño de sistemas digitales conocido como HLS. La HLS es un procedimiento realizado por una herramienta de software para obtener una descripción de hardware (en HDL) a partir de una función en algún lenguaje de alto nivel (de programación procedural) como MatLab o C/C++. Estas herramientas de HLS ofrecen la posibilidad de limitar o incluso eliminar la necesidad de escribir HDL para diseñar un sistema digital. Sin embargo, esto significa renunciar al control fino que se tiene sobre el hardware al diseñarlo con el método convencional en RTL. En general, HLS sirve para implementar cálculos y no tareas con restricciones estrictas de tiempo o sincronía, como protocolos. Aún si usar HLS evita diseñar en RTL, sigue siendo necesario entender conceptos de diseño digital, ya que las herramientas de HLS permiten incluir directivas que, sin modificar la funcionalidad del diseño, permiten al usuario influir en el hardware obtenido. El usuario debe entender el impacto de las directivas en la arquitectura para su aprovechamiento.

A continuación, presentamos algunos conceptos que utilizaremos en el contexto de HLS. Algunos de los términos que usaremos son específicos de la herramienta Vitis HLS de AMD-Xilinx, sin embargo, se refieren a conceptos que no son exclusivos de la herramienta:

- **Módulo**: unidad jerárquica de un diseño digital (en HDL), que corresponde a la implementación de una función o bucle (en HLS). La herramienta de HLS agrega lógica de control a cada módulo para gestionar su funcionamiento, incluyendo control de sus iteraciones y acceso a memoria.
- **Intervalo de Inicialización (II)**: tiempo de espera entre dos iteraciones consecutivas de un mismo bucle. Múltiples iteraciones pueden procesarse simultáneamente en el mismo módulo mediante *pipeline*, pero si hay dependencia de datos entre iteraciones puede que el II sea superior a un ciclo de reloj.
- **Planificación**: concepto heredado del software que se refiere a la secuencia en que se ejecutan coordinadamente los distintos módulo que implementan el acelerador diseñado con HLS. La planificación es producida por la herramienta como parte de la síntesis de alto nivel y, tratándose de hardware, puede contener ejecución simultánea de módulos.
- ***Trip count***: número efectivo de ejecuciones del módulo que implementa un bucle, tal que $trip_count = N_{it}/F_{UN}$, donde N_{it} es el número de iteraciones del bucle (en software) y F_{UN} es el factor de desenrollamiento del módulo implementado.
- **Pragma**: directiva que que permite al usuario guiar el proceso de HLS en la herramienta Vitis HLS de AMD-Xilinx.

Descripción de pragmas

A continuación, se detallan los *pragmas* más relevantes utilizados en la literatura y en el presente trabajo. Aquí también usaremos terminología de AMD-Xilinx, sin embargo, los efectos de los *pragmas* se asocian con conceptos genéricos de diseño digital y arquitecturas paralelas (más información en la guía de usuario [26]):

- `PIPELINE`: habilita la paralelización de un bucle sin aumentar los recursos requeridos. Si el procesamiento de una iteración demora más de un ciclo, entonces el módulo implementado está particionado por registros. Luego, mientras los datos de entrada están disponibles, la siguiente iteración puede empezar antes de que la anterior termine.
- `UNROLL`: requiere que el módulo de hardware que implementa un bucle ejecute múltiples iteraciones simultáneamente. En teoría, la latencia total del bucle es inversamente proporcional al factor de desenrollamiento y la utilización de recursos es proporcional al factor de desenrollamiento.
- `ARRAY_PARTITION`: fuerza que la implementación de memoria de cierto arreglo permita el acceso en lectura o escritura a múltiples elementos durante un mismo ciclo de reloj.
- `LOOP_MERGE`: implementa múltiples bucles, que tienen el mismo número de iteraciones, juntos en un mismo módulo, si no lo impide la dependencia de datos.
- `LOOP_FLATTEN`: transforma un par de bucles anidados en un sólo bucle que itera sobre todas las combinaciones de ambos bucles, si no lo impide la dependencia de datos. En general, se requiere que el bucle externo sólo contenga al bucle interno.
- `INTERFACE`: selecciona el tipo de interfaz asociado a un argumento o el método de control de un módulo.
- `LOOP_TRIPCOUNT`: especifica una cota para la cantidad de iteraciones de un bucle con número de iteraciones variable, para ser considerado en el reporte de HLS.

En este trabajo utilizaremos los *pragmas* `PIPELINE`, `UNROLL`, `ARRAY_PARTITION`, `LOOP_MERGE` y `LOOP_FLATTEN` para la optimización del acelerador de MPC diseñado con HLS. En cambio, los *pragmas* `INTERFACE` y `LOOP_TRIPCOUNT` cumplen otras funciones prácticas, que no son críticas para comprender el trabajo, y se describen por completitud.

3 Análisis de código para algoritmo MPC

Antes de poder acelerar el controlador de forma costo-efectiva, es importante analizar los patrones de acceso a datos y dependencia entre operaciones en los algoritmos asociados, para así identificar potenciales puntos que se puedan optimizar a través de la aceleración por hardware. En este capítulo analizamos la estructura del algoritmo de referencia para MPC implícito, caracterizando la latencia asociada a distintos bloques de código durante la ejecución del controlador, distinguiendo precisamente qué partes del algoritmo pueden procesarse en tiempo de diseño y cuáles deben procesarse en tiempo de ejecución. Luego, se analiza el potencial de paralelización de las operaciones de tiempo de ejecución, considerando proyecciones en términos de latencia y uso de recursos. A partir del análisis desarrollado, se realizan adaptaciones al código original y se identifican configuraciones tentativas de pragmas de HLS a utilizar para una posterior exploración cuantitativa del espacio de diseño, la cual se reporta en detalle en el Capítulo 4.

3.1. Algoritmo de MPC usando ADMM

Los pasos genéricos para ejecutar MPC desde un punto de vista algorítmico se muestran en el Algoritmo 2. Primero, hay una etapa de configuración (línea 1) que comprende parte de la formulación MPC que es constante para la aplicación. En este paso se calculan todas las matrices que son constantes, dado que la planta se considera invariante en el tiempo, lo cual permite reducir el procesamiento requerido durante la operación de la planta y con eso la latencia del controlador en tiempo de ejecución. Luego, un bucle infinito calcula y aplica la actuación en cada instante de muestreo, siguiendo la secuencia de sensado o estimación de los estados (línea 3), formulación del problema de optimización QP (línea 4), resolución de dicho problema QP para obtener las actuaciones a lo largo del horizonte (línea 5), y aplicación de la actuación determinada para el siguiente intervalo de muestreo (línea 6). Como se detalla en la Sección 2.2.2, la latencia de ejecución de una iteración del bucle debe estar acotada por el periodo de muestreo objetivo del controlador, el cual viene dado por los requerimientos de la aplicación.

En el resto de esta sección se analiza en más detalle la estructura del código base utilizado como referencia para identificar las partes dominantes en la latencia del lazo de control y caracterizar como estas escalan con la dimensionalidad del problema QP, además de plantear una reestructuración del código base en C++ para facilitar la posterior exploración de pragmas de HLS.

Algoritmo 2 MPC

1: $Q, G \leftarrow$ FORMULACION_OFFLINE() ▷ Constante para el sistema
2: **for** each sampling instant **do**
3: $x_0, d_0 \leftarrow$ SENSADO()
4: $q, g \leftarrow$ FORMULACION_ONLINE()(x_0, r_0, d_0)
5: $\theta_k \leftarrow$ RESOLUCION_QP(Q, q, G, g)
6: ACTUACION(θ_k)
7: **end for**

```
for i=1:length(k)
    [xinf, uinf] = fx_stationary(A, B, C, rk(:,i), Bp, dk(:,i));
    q = ((xk(:,i)-xinf)'*F)';
    Huinf = H*uinf;
    Ixinf = I*xinf;
    WDx = single(WD*(xk(:,i)-xinf));
    c = repmat(single(umax-Huinf), N_HOR,1);
    d = repmat(single(Huinf-umin), N_HOR,1);
    e = repmat(single(xmax-Ixinf), N_HOR,1);
    f = repmat(single(Ixinf-xmin), N_HOR,1);
    g = [c; d; e-WDx; f+WDx];
    [t_ADMM, z_ADMM, u_ADMM] = fx_qp_admm(R_inv, q, G, g,t_ADMM, z_ADMM,
   u_ADMM, rho, IT_ADMM);
    uk(:,i) = t_ADMM(1:M_SYS) + uinf;
end
```

Código 3.1: Procesamiento online de MPC en MatLab.

3.1.1. Profiling del controlador

El Código 3.1 es un extracto del código MatLab utilizado, que está basado en [13], y corresponde a la parte iterativa de MPC que se procesa durante la operación de la planta. Como se indica en la Sección 2.2.2, por simplicidad omitimos la latencia del sensado y de la actuación. Este código fue desarrollado en trabajos anteriores por otros estudiantes, y está orientado a ser descriptivo con un enfoque funcional en términos numéricos, sin considerar optimizaciones específicas para reducción del tiempo de ejecución.

La iteración de MPC, que se ejecuta en cada intervalo de muestreo, está compuesta por la formulación del problema QP y su resolución. Las líneas 2 a 11 corresponden a la formulación a realizar *online*, lo cual incluye:

1. Cálculo del estado estacionario para seguimiento de referencia a partir de x_0, r_0 y d_0, donde la función *fx_stationary()* (línea 2) ejecuta el proceso detallado en la Sección 2.1.2. La matriz de perturbaciones Bp es constante y también se puede calcular la matriz T^{-1}de la ecuación (2.23) de manera *offline*.

2. Cálculo del vector q (línea 3) según la ecuación (2.29). Para ello se define la matriz auxiliar constante $F = 2D^{\mathrm{T}}KE$.

3. Cálculo del vector g (líneas 4-11) según la ecuación (2.33). En este paso, son constantes las restricciones de entrada ($u^{\min}$, $u^{\max}$) y de estado ($x^{\min}$, $x^{\max}$), así como sus respectivas ponderaciones (H e I). Además, se define la matriz auxiliar $WD = W * D$.

Tabla 3.1: *Profiling* del procesamiento *online* de MPC, con 25 iteraciones de ADMM, para problemas de diferentes tamaños.

Horizonte de predicción	$h = 2$	$h = 4$	$h = 8$
Variables de decisión del problema QP	$N = 4$	$N = 8$	$N = 16$
Restricciones del problema QP	$M = 40$	$M = 80$	$M = 160$
Tiempo total (s)	0.289	0.318	0.463

Línea de código	Porcentaje de tiempo		
[t_ADMM,z_ADMM,u_ADMM] = fx_qp_admm(...	66.3 %	69.1 %	73.1 %
bl = [Bpd*dk;vrk];	13.6 %	12.0 %	11.6 %
c = repmat(single(umax-Huinf), N_HOR, 1);	2.3 %	2.3 %	1.9 %
g = [C; d; e-WDx; f+WDx];	2.3 %	2.2 %	1.8 %
Todas las demás lineas	15.5 %	14.4 %	11.6 %

Luego, respecto a la resolución del problema QP con ADMM (línea 12), se tiene que G es constante y que, en lugar de Q, se puede definir $R^{-1} = \left(Q + \rho(G^{\mathrm{T}}G)\right) \backslash I_N$. Además, se define la matriz auxiliar $P = -\rho G^{\mathrm{T}}$. La función que implementa ADMM se compone de un bucle que en cada iteración se acerca a la solución óptima. Las operaciones realizadas son idénticas para cada iteración, por lo que se puede expresar la latencia total de la iteración de MPC como:

$$\begin{aligned} L_{MPC}(IT_{ADMM}) &= L_{Formulacion} + L_{Resolucion_QP} \\ &= L_{Formulacion} + IT_{ADMM} \cdot L_{Iteracion_ADMM} \end{aligned} \tag{3.1}$$

donde $L_{Formulacion}$ es la latencia de la formulación *online* del problema QP, $L_{Resolucion_QP}$ es la latencia del *solver* QP, $L_{Iteracion_ADMM}$ es la latencia de una iteración individual del bucle de ADMM y IT_{ADMM} es la cantidad de iteraciones de ADMM. Luego, el impacto de reducir $L_{Iteracion_ADMM}$ está ponderado por IT_{ADMM}. Como se mencionó en la Sección 2.1.3, en este trabajo consideramos que IT_{ADMM} viene predefinido y se mantiene fijo durante el funcionamiento de la planta.

Como se menciona en el Capítulo 1, en la literatura se muestra que la resolución del problema QP suele dominar el costo computacional de MPC implícito. Como paso de verificación rápida, utilizamos la herramienta de *profiling* de Matlab para analizar el Código 3.1. La Tabla 3.1 muestra el resultado del *profiling* del controlador MPC para tres horizontes de predicción, que resultan en distintos valores para los parámetros de tamaño de las matrices del problema QP N y M. Para cada horizonte, la tabla presenta las cuatro líneas del código con mayor tiempo de ejecución, con su respectivo porcentaje de tiempo de ejecución respecto a la latencia total de la iteración de MPC. Si bien se sabe que MatLab incorpora un significativo *overhead* en los tiempos de ejecución medidos y no está orientado a desempeño, las mediciones en forma de porcentajes entregan un indicador cualitativo que verifica que la mayor parte de la latencia de cómputo durante un intervalo de control está asociada a la resolución del problema QP. Para comprender el efecto del problema QP en el tiempo de ejecución de MPC, analizaremos la relación entre los parámetros de la aplicación y el costo computacional del algoritmo.

Tabla 3.2: Parámetros de la aplicación.

Parámetro	Descripción
m	entradas del sistema
n	estados del sistema
p	salidas del sistema
a	pares de restricciones de entradas
b	pares de restricciones de estados
d	número de perturbaciones medibles
h	horizonte de predicción
IT_{ADMM}	iteraciones de ADMM

3.1.2. Dimensionalidad del problema

Por dimensión del problema MPC nos referimos a los tamaños de variables del algoritmo que afectan el costo computacional, y que corresponden al conjunto de parámetros definidos por la aplicación, presentados en la Tabla 3.2. Para acelerar MPC de una forma que sea escalable, nos interesa entender cómo evoluciona el costo computacional con dichos parámetros, en particular aquellos que dependen de decisiones de diseño del usuario. En la tabla, los parámetros m, n, p y d son determinados por el tamaño del modelo matemático discretizado de la planta. En cambio, los parámetros a y b de número de restricciones, y el parámetro del horizonte de predicción h son elecciones del usuario basadas en el desempeño de control y condiciones de funcionamiento deseados para la aplicación. Como se detalla en la Sección 2.1.3, IT_{ADMM} también es un parámetro de la aplicación definido por el usuario y que afecta la convergencia al valor óptimo del controlador, pero no afecta la dimensión del problema QP que se genera y resuelve en cada iteración.

A partir de los parámetros de la Tabla 3.2, se calculan los parámetros N y M asociados al problema QP, que corresponden a la cantidad de variables de decisión y de restricciones en todo el horizonte de predicción respectivamente. Nos referiremos al par $N \times M$ como la dimensión del problema QP, que en formulación densa está dada por:

$$\begin{aligned} N &= h \cdot m \\ M &= 2h \cdot (a + b) \end{aligned} \tag{3.2}$$

Se destaca que, M crece linealmente con el número de restricciones $a + b$, y que tanto N como M crecen linealmente con el horizonte de predicción h.

Luego, si la resolución del problema QP domina el costo computacional de MPC, podemos considerar que su dimensión $N \times M$ es representativa del costo computacional del lazo MPC, ya que condensa los efectos de todos los parámetros que afectan la latencia de una iteración de ADMM. Las operaciones que componen la iteración de ADMM se muestran en la Tabla 3.3, separadas según su tamaño y clasificadas en operaciones elemento-a-elemento (EOP) y multiplicaciones matriz-vector (MVM). Por ejemplo, hay una MVM de tamaño $M \times N$, que está compuesta por $M \cdot N$ multiplicaciones de punto flotante y $M \cdot (N - 1)$ sumas de punto flotante. El costo computacional de las MVM, en término de operaciones atómicas de punto flotante, puede llegar a crecer cuadráticamente con la dimensión del problema QP. Por otra parte, destacan las ocho EOP de tamaño M, cada una compuesta por M operaciones de punto flotante (suma o resta). Por su parte, el costo computacional de las EOP crece linealmente

Tabla 3.3: Número de operaciones aritméticas en el bucle de ADMM, por tipo y demensión.

Tipo de operación	Tamaño	Cantidad
Multiplicación Matriz-Vector (MVM)	$M \times N$	1
	$N \times M$	1
	$N \times N$	1
Elemento-a-Elemento (EOP)	M	8
	N	1

con una de las dimensiones del problema QP. Este análisis sugiere que las multiplicaciones son predominantes en la latencia total, en particular cuando el parámetro N es grande. Sin embargo, queda evaluar el potencial de aceleración de las distintas operaciones y el costo en recursos asociado. La aceleración efectiva dependerá de la paralelización que se logre al implementar las operaciones y de las dependencias de datos entre ellas, lo que será analizado en la Sección 3.2.

3.1.3. Implementación en código C++

Inicialmente se cuenta con una implementación de ADMM en C++ derivada de [13], disponible bajo solicitud en el repositorio GitLab [27]. Implementamos el lazo MPC en C++ con un código 'vanilla', como lo escribiría un desarrollador de software, sin pretender optimizar la arquitectura que resultará de la HLS. El objetivo es obtener un código base en C++ que permita aplicar y evaluar el uso de *pragmas*, para lo cual se consideran ciertas prácticas recomendadas al escribir un código para HLS:

- Se utilizan referencias de C++ para los parámetros de las funciones, evitando el uso de punteros para evitar definir funcionalidades que no sean sintetizables.
- Se utilizan variables globales para los datos constantes, en lugar de ingresarlos como parámetros de las funciones.
- Se utilizan *templates* para generalizar la definición de aquellas funciones que se implementarán para datos de distintos tamaños, como es el caso de las operaciones de álgebra lineal. Esto no presenta una ventaja en funciones que se llaman sólo una vez o siempre con arreglos del mismo tamaño.
- Se etiquetan todos los bucles para facilitar la aplicación de *pragmas* y el análisis de los reportes que entrega la herramienta de HLS.
- Se evita definir bucles con número de iteraciones que se determine en tiempo de ejecución. Es necesario que la cantidad de iteraciones sea fija para hacer optimizaciones como *unroll*. Además, se recomienda para que la herramienta pueda conocer el *trip count* y estimar la latencia. Se hace una excepción para la cantidad de iteraciones de ADMM, donde no es posible hacer unroll y se ingresa manualmente el valor del *trip count* para el cálculo de la latencia.

El uso de variables globales es el principal cambio y también el que requiere más cuidado. Los argumentos de una función generan puertos en el módulo implementado, y en la función principal se asume que son datos variables. Utilizar variables globales para los datos constantes

simplifica la interfaz del módulo y permite a la herramienta hacer optimizaciones en la implementación de los datos. La herramienta de HLS es capaz de identificar aquellos datos que no cambian, para implementarlos como ROM, y también las operaciones con resultados constantes. En principio, esto permite ahorrar hardware, por ejemplo, evitando la implementación de multiplicaciones por valores nulos. Sin embargo, existe una limitación para aprovechar estas optimizaciones. Si la multiplicación es parte de un bucle y en algunas iteraciones el resultado no es nulo, el módulo deberá implementarse de todas formas. En general, la herramienta logra aprovechar la información de las constantes cuando balancea las operaciones de una implementación desenrollada, simplificando el módulo.

Es importante considerar que las variables globales que la herramienta pueda inferir como constantes deben ser inicializadas con los valores definitivos en un archivo fuente incluido durante la síntesis de alto nivel. Las variables globales se pueden modificar en la simulación, pero esta información no se considera en la síntesis. Si no se inicializan las variables globales, la herramienta asumirá que tienen valor 0 y podría implementar optimizaciones indeseadas que no conserven la funcionalidad.

Respecto al uso de *templates*, cabe destacar que dos llamados a una misma función con parámetros diferentes producirán dos instancias distintas en la implementación. Quiere decir que la implementación de una suma de vectores de tamaño N será un módulo diferente del que implementa la suma de vectores de tamaño M. En ese sentido el uso de *templates* sirve para simplificar el código C++, generalizando la definición de funciones, y también la aplicación de *pragmas*. Aplicar *pragmas* a funciones con *templates* puede facilitar el trabajo, pero también limita la posibilidad de aplicar optimizaciones adaptadas a los tamaños de los arreglos. Consideramos entonces que el uso de *templates* es atractivo para realizar una exploración rápida de las posibles estrategias de optimización con *pragmas*.

3.2. Potencial de paralelización y estrategias para HLS

Nos interesa usar FPGAs para la implementación de MPC porque contienen una alta densidad de lógica programable, la cual sirve para paralelizar aquellas partes del algoritmo que lo permiten, y así acelerar su ejecución. En general, en base a la granularidad y densidad de recursos que ofrece una FPGA moderna, se espera que la paralelización disminuya la latencia con el costo de incrementar la utilización de recursos. Para una aplicación dada, el objetivo es reducir la latencia total del bucle completo de control debajo del periodo de muestreo, considerando restricciones de recursos disponibles. El enfoque habitual con diseño RTL provee al usuario control fino de este compromiso entre latencia y recursos, a expensas de un proceso de desarrollo extenso y propenso a errores. Usando el flujo alternativo con herramientas de HLS, se tiene un menor control sobre la arquitectura generada, y por lo tanto de la latencia y uso de recursos. Se dispone de un conjunto limitado de *pragmas* que representan estrategias precisas para la arquitectura del diseño (ver Sección 2.2.3), pero cuya implementación es aplicada de forma automatizada por la herramienta, en la medida que sea posible. Si se aplican *pragmas* incoherentes con el código o entre sí, puede que la herramienta los ignore, que no se modifique la arquitectura como se esperaba (como se reporta en [11]) e incluso que el desempeño empeore.

En esta Sección identificamos las partes del algoritmo con potencial de ser aceleradas mediante paralelización usando *pragmas*, estableciendo directrices para realizar la posterior exploración de soluciones y determinar las ventajas y limitaciones del uso de HLS. En el

Capítulo 4 se evalúa cuantitativamente qué estrategias de optimización resultan finalmente más costo-efectivas para acelerar controladores MPC en los casos de estudio.

En la práctica, el esfuerzo de diseño usando HLS se centra principalmente en los bucles del algoritmo, los cuales representan operaciones que se realizan múltiples veces con distintos valores de entrada. La mayoría de los *pragmas* de interés, presentados en la Sección 2.2.3, afectan a los bucles o a los arreglos sobre los cuales iteran bucles. Por esta razón, las operaciones de álgebra lineal en ADMM, constituidas de iteraciones sobre vectores y matrices, son candidatas para paralelización. Para analizar cómo aprovechar los *pragmas* pipeline, *unroll*, *array_partition* y *loop_merge*, ilustraremos su efecto en el hardware implementado, al estudiar el potencial de paralelización de las operaciones elemento-a-elemento (EOP), de la multiplicación matriz-vector (MVM) y del algoritmo completo de ADMM.

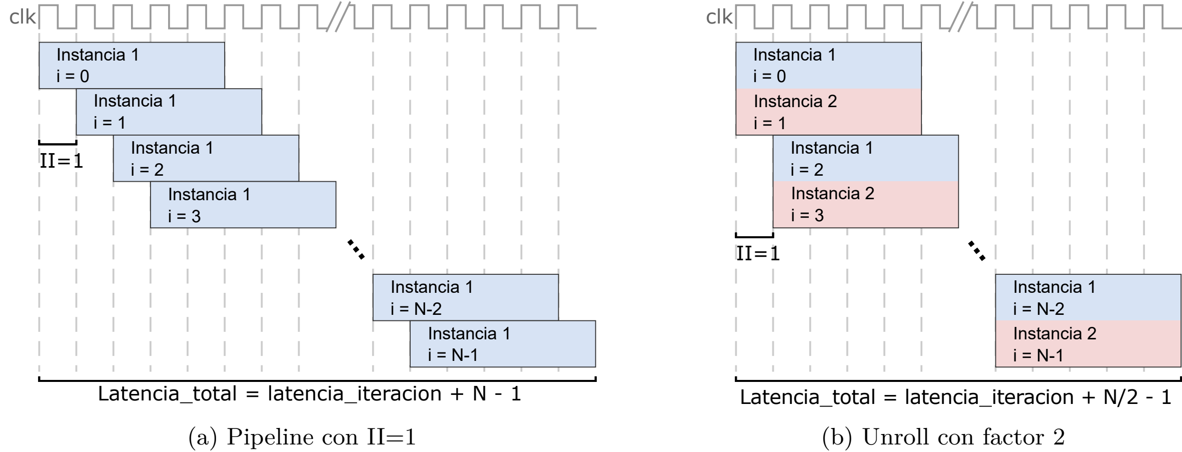
3.2.1. Operaciones elemento-a-elemento

El caso de las operaciones elemento-a-elemento (Element-wise Operations o EOP) es el más simple de comprender y predecir, ya que las iteraciones del bucle son independientes. En una ejecución secuencial, un mismo módulo de hardware procesa todos los elementos, uno detrás del otro. Se conoce como *pipeline* a la estrategia de particionar ese módulo en múltiples etapas usando registros (Flip-Flops) para realizar multiplexación temporal y procesar múltiples elementos simultáneamente, como se ilustra en la Figura 3.1a. En el contexto de diseño con HLS, el usuario elige una frecuencia de reloj objetivo para el sistema. Luego, si la lógica que implementa la iteración de un bucle demora más de un periodo del reloj, la herramienta agregará registros para separarla en etapas. En ese caso, el recurso de hardware asignado ya cuenta con soporte de *pipeline*, y la directiva *pipeline* se vuelve una solicitud de optimizar la planificación de las iteraciones reduciendo el II, por lo cual esperamos que mejore la latencia del bucle sin afectar significativamente el uso de recursos. Para las EOP, que no tienen dependencia de datos, en cada ciclo de reloj la instancia puede recibir un nuevo conjunto de entradas mientras los anteriores avanzan a etapas posteriores. Equivalentemente, se dice que el número de ciclos de espera entre ejecuciones consecutivas es mínimo, i.e., $II = 1$.

Desenrollar bucles es otra estrategia que puede utilizarse en la implementación de las EOP. Desenrollar parcialmente un bucle sin dependencia de datos con un factor $F_{UN} = X$ es equivalente a replicar X veces el hardware requerido para procesar una iteración individual. La Figura 3.1b muestra un ejemplo de desenrollamiento de un bucle con factor de desenrollamiento $F_{UN} = 2$. El módulo implementado utilizará en torno al doble de recursos que en (a), con una reducción de latencia de $N_{it}/2$ ciclos, donde N_{it} es el número de iteraciones del bucle.

Para que se pueda cumplir el desenrollamiento de la Figura 3.1b, asumimos que en cada ciclo hay dos conjuntos de entradas disponibles y que se pueden guardar dos conjuntos de salidas. En la práctica, para garantizar que el desenrollamiento sea efectivo, este debe estar acompañado por la directiva *array_partition* en aquellos arreglos sobre los cuales itera el bucle, de forma que la lectura de entradas y la escritura de salidas sea consistente con el *throughput* que alcanza el módulo. De lo contrario, la aceleración obtenida por desenrollar el bucle estará limitada por el acceso a memoria. Si no se particiona adecuadamente un arreglo, o si existe alguna condición externa que restrinja la disponibilidad de los datos de entradas, la latencia del módulo no se reducirá como es esperado.

Cuando se aplica desenrollamiento parcial a un bucle, el uso de recursos se puede estimar



(a) Pipeline con II=1

(b) Unroll con factor 2

Figura 3.1: Planificación de bucle sin dependencia de datos entre iteraciones.

como:

$$Utilizacion_Recursos(F_{un}) = F_{un} \cdot R_{it} \tag{3.3}$$

donde $F_{un} = 1, 2, ..., N_{it}$ es el factor de unroll, N_{it} es el número de iteraciones del bucle y R_{it} es la cantidad de recursos requerida para una iteración individual. Por otra parte, la latencia del bucle completo en ciclos de reloj puede estimarse como

$$L_{bucle}(F_{un}) = L_{it} + II \cdot \left(\left\lceil \frac{N_{it}}{F_{un}} \right\rceil - 1 \right) \tag{3.4}$$

donde L_{it} es la latencia de una iteración individual del bucle en ciclos de reloj y la expresión $\lceil N_{it}/F_{un} \rceil$ corresponde al *trip count*. Entonces, para operaciones sin dependencia de datos, es posible mantener $II = 1$, independiente del factor de desenrollamiento, por lo que la aceleración puede ser estimada como

$$Accel(F_{un}) = L_{bucle}(1) - L_{bucle}(F_{un}) = N_{it} - \left\lceil \frac{N_{it}}{F_{un}} \right\rceil \tag{3.5}$$

Las ecuaciones (3.3) y (3.5) muestran que, cuando se desenrolla un bucle sin dependencia de datos, la utilización de recursos aumenta linealmente con el factor de desenrollamiento, mientras que la tasa de reducción de latencia disminuye cuadráticamente con el factor de desenrollamiento, suponiendo que no haya limitaciones de acceso a memoria. En términos de la reducción del tiempo de ejecución y el costo asociado en recursos, aumentar el factor de desenrollamiento tiene retorno decreciente.

En términos de escalabilidad, la cantidad de operaciones atómicas de punto flotante en las EOP aumenta linealmente con el tamaño de los arreglos. Igualmente, en ejecución secuencial (software), se espera que el tiempo de ejecución también aumenta linealmente con el tamaño de los arreglos. Al aplicar *unroll* parcial, el tiempo de ejecución sigue aumentando linealmente con el tamaño de los arreglos, pero se consigue disminuir la tasa de crecimiento. En cambio, si el desenrollamiento es completo se tiene que el tiempo de ejecución es independiente del tamaño de los arreglos. Esto significa que la aceleración porcentual obtenida con desenrollamiento completo aumenta con el tamaño del arreglo, suponiendo que se dispone de recursos suficientes.

En resumen, las operaciones elemento-a-elemento se pueden implementar eficientemente con pipeline, reduciendo su latencia en comparación a una ejecución secuencial, suponiendo que las operaciones de punto flotante tomen más de un ciclo de reloj. Además, es posible acelerar las EOP mediante desenrollamiento. Aunque el *unroll* de las EOP tiene rendimiento decreciente, esperamos que su paralelización requiera bajo uso de recursos, presentándose como una forma económica de reducir la latencia. El potencial de acelerar ADMM optimizando las EOPs se basa en la cantidad de estas operaciones que haya en el algoritmo, y crecerá con el tamaño del problema.

3.2.2. Multiplicación matriz-vector

Comprender la implementación paralelizada de las multiplicaciones matriz-vector (Matrix-Vector Multiplication o MVM) y la potencial reducción de latencia es más complejo que para las EOP, ya que se trata de operaciones con bucles anidados y dependencias de datos. Como se ilustra en [15], la dependencia intrínseca de la multiplicación es entre las columnas. En relación a la implementación en software de la multiplicación, consideramos que la notación $\mathbf{A}[i][j]$ representa al elemento de la fila i y columna j de la matriz $\mathbf{A}$. Denominamos versión *row-wise* cuando el bucle externo del código itera sobre las filas de la matriz, y versión *column-wise* cuando el bucle externo itera sobre las columnas de la matriz.

El Código 3.2 muestra una versión *column-wise* de la MVM en C++. Se puede apreciar que la multiplicación-acumulación del bucle interno (línea 9) afecta diferentes elementos del arreglo R en cada iteración, de forma que no hay dependencia entre iteraciones y se puede tener $II = 1$. Como todo módulo, la implementación del bucle interno tendrá un *overhead* de latencia asociado a su control, lo que aumenta la latencia de la iteración del bucle externo. Para eliminar dicho *overhead* y disminuir la latencia total de la multiplicación existe la estrategia *loop flatten*, que se refiere a fusionar los bucles interno y externo en un sólo módulo. Para poder usar *loop flatten* se requiere que no haya código dentro del bucle externo salvo por el bucle interno, lo cual se cumple en la MVM *column-wise* (ver Código 3.2). Además, se puede paralelizar la MVM *column-wise* replicando el módulo obtenido al usar *loop_flatten* mediante desenrollamiento. Sin embargo, si el factor de desenrollamiento es muy alto es esperable un deterioro del intervalo de inicialización que mitigue la aceleración obtenida. Esto ocurriría si la suma de punto flotante en la multiplicación-acumulación (línea 9) de una fila tuviera que esperar a que termine la suma de la misma fila en la columna anterior para poder empezar. Esperamos que la dependencia de datos se manifieste si se cumple que $trip_count < L_{fadd} * N_{col}$, donde L_{fadd} es la latencia de la suma de punto flotante y N_{col} es el número columnas de la matriz.

```
template<int N, int M, typename>
void mvmult(const T (&A)[N][M], const T (&B)[M], T (&R)[N]){
    mvmult_column: for(int j=0; j<M; j++){
        mvmult_row: for(int i=0; i<N; i++){
            if(j==0){
                R[i] = A[i][j] * B[j];
            }
            else {
                R[i] += A[i][j] * B[j];
            }
        }
    }
    return;
}
```

Código 3.2: MVM tipo *column-wise* en C++.

```
1  template<int N, int M, typename>
2  void mvmult(const T (&A)[N][M], const T (&B)[M], T (&R)[N]){
3      mvmult_row: for(int i=0; i<N; i++){
4          R[i] = 0;
5          mvmult_column: for(int j=0; j<M; j++){
6              R[i] += A[i][j] * B[j];
7          }
8      }
9      return;
10 }
```

Código 3.3: MVM tipo *row-wise* en C++.

Por otra parte, el Código 3.3 presenta una versión *row-wise* de la MVM en C++. En este caso, la multiplicación-acumulación del bucle interno (línea 6) afecta consecutivamente al mismo elemento del arreglo R, de forma que hay dependencia entre iteraciones del bucle interno se tiene $II > 1$, salvo que el bucle interno esté completamente desenrollado. Mientras mayor sea el número de columnas, mayor será el costo en recursos de desenrollar completamente el bucle interno, pero esto habilita el desenrollamiento del bucle externo (línea 3), permitiendo mayor paralelización. Al aplicar *unroll* parcial del bucle externo no hay dependencias, de forma que se mantiene $II = 1$ y se puede estimar la aceleración con la ecuación (3.5), pero el hardware a replicar corresponde a todo el bucle interno desenrollado.

El Anexo A presenta un análisis preliminar de la implementación independiente de la MVM *column-wise* y *row-wise*. Se reporta que, en general, el enfoque *column-wise* ofrece mejor costo-beneficio en términos de uso de recursos y aceleración que el *row-wise* cuando el número de columnas es alto y el factor de desenrollamiento es bajo. La versión *row-wise* se vuelve atractiva si es razonable hacer desenrollamiento completo del loop interno. Sin embargo, analizar la implementación de ambas versiones de la MVM de forma independiente omite el impacto que puedan tener en realción a las dependencias entre operaciones de álgebra lineal, a nivel del algoritmo. De hecho, una de las posibles ventajas de utilizar la MVM *row-wise* por sobre la *colum-wise* tiene relación con la dependencia de datos entre operaciones. En la versión *row-wise* se calcula el valor final de cada elemento del vector de salida antes de pasar al siguiente, mientras que la versión *colum-wise* procesa todos los elementos de la salida al mismo tiempo. Entonces, utilizar la MVM *row-wise* puede reducir la latencia al permitir que las operaciones posteriores comiencen anticipadamente a medida que los elementos del vector de salida están disponibles [15].

En términos de escalabilidad, la cantidad de operaciones atómicas de punto flotante en la multiplicación llega a crecer cuadráticamente con el tamaño de los arreglos, tal como lo hace el tiempo de ejecución en una ejecución secuencial. Aplicando desenrollamiento parcial o completo de los bucles interno y externo se consigue reducir el orden de crecimiento del tiempo de ejecución respecto al tamaño de los arreglo. Es decir, el tiempo de ejecución de la MVM escala mejor con más desenrollamiento. Sin embargo, para los tamaños de arreglos en los problemas de interés, aún es posible que una implementación más desenrollada tenga peor tiempo de ejecución si es que se hacen aparecer dependencias de datos.

En resumen, el potencial de acelerar ADMM optimizando las multiplicaciones se basa en que estás sean dominantes en la latencia total y que su paralelización sea efectiva a pesar de la dependencia interna de datos. De acuerdo a los resultados presentados en el Anexo A la

eficacia de la paralelización de las multiplicaciones usando *pragmas* depende de la relación de aspecto de la matriz y de la forma de escribir el código. Además, el análisis de la MVM como operación individual omite el impacto en su implementación de las dependencias entre operaciones de álgebra lineal, el cuál será considerado en el Capítulo 4. Por otra parte, se destaca que al paralelizar las multiplicaciones es esperable un costo significativo en términos del uso de recursos.

3.2.3. Potencial de paralelización de ADMM

Como se comentó anteriormente, la resolución del problema QP es la etapa con mayor costo computacional del lazo de control MPC. Dado que las iteraciones del algoritmo de ADMM tienen dependencia de datos entre ellas, no es posible ejecutar múltiples iteraciones de forma paralela. En cambio, es esperable que aumentar el grado de paralelización dentro de la iteración de ADMM impacte significativamente la latencia total del controlador. Analizando el algoritmo de ADMM presentado en la Sección 3.1 observamos que los pasos principales de actualización de t_k, z_k y u_k dependen entre sí. Esta dependencia limita el grado de paralelización. Entonces, el algoritmo de ADMM admite un aumento de la paralelización principalmente en la implementación de operaciones individuales de álgebra lineal y en la implementación de los conjuntos de operaciones que conforman los pasos actualización de t_k, z_k y u_k (líneas 5 a 8 del Algoritmo 1).

Entre las operaciones elemento-a-elemento (suma, resta y máximo de vectores) y las multiplicaciones matriz-vector, es esperable que las MVM tengan mayor costo computacional individualmente. Sin embargo, la paralelización interna de las MVM mediante desenrollamiento es más costosa en recursos que el desenrollamiento de las EOP. Además, como se muestra en la Tabla 3.3, hay más EOPs que MVMs en el algoritmo de ADMM. Cabe destacar que el módulo que implementa una operación puede reutilizarse en futuros llamados a la misma función si no hay conflicto en la planificación. De esta forma, una optimización de una suma de tamaño M puede reducir la latencia en varios lineas del algoritmo de ADMM. En consecuencia, aunque en la literatura se presta más atención a la aceleración de la MVM [10, 15], se espera que resulte más costo-efectivo paralelizar internamente las EOP que las MVM. Eventualmente, puede ser prioritario aplicar optimizaciones a las MVM si la implementación por defecto se ve afectada negativamente por las dependencias de datos.

Por otra parte, al analizar posibles paralelizaciones a nivel del algoritmo, i.e. entre operaciones de álgebra lineal, se cuenta con la directiva *loop merge*. Mediante *loop merging* se puede optimizar la implementación de conjuntos de operaciones del algoritmo , por ejemplo, cada una de las líneas 5, 7 y 8 del algoritmo, que actualizan los vectores t_k, z_k y u_k. Estas líneas contienen conjuntos de operaciones entre vectores de mismo tamaño y sin dependencias entre sí. De esta forma, sus bucles pueden fusionarse e implementarse en un mismo módulo. Incluso la multiplicación puede incluirse en el *loop merge* si se expone en el bucle externo la dimensión con correspondiente al número correcto de iteraciones. Los módulos implementados con *loop merge* son menos génericos, lo que es perjudicial para la reutilización. Sin embargo, aún es posible reutilizar los DSP individualmente, que son los recursos más escasos de la FPGA y críticos para operaciones de punto flotante. En teoría, la fusión de bucles es compatible con el desenrollamiento. Más que el número de iteraciones, es necesario que los *trip count* de los bucles coincidan para que se puedan fusionar. Sin embargo, en un análisis preliminar observamos que la herramienta y versión utilizadas no soporta la combinación de directivas *unroll* y *loop merge*.

4 | Generación de aceleradores en hardware usando HLS

Este capítulo aborda el desarrollo de un controlador MPC con HLS y su optimización utilizando pragmas. La Sección 4.1 introduce el flujo de diseño de aceleradores utilizando el conjunto de herramientas de AMD-Xilinx. Luego, en la Sección 4.2 aplicamos el análisis del Capítulo 3 a la exploración del espacio de soluciones de implementación del controlador MPC.

4.1. Flujo de herramientas para HLS

Esta sección explica el flujo de trabajo utilizado para la implementación en FPGA de parte de un algoritmo usando, es decir, el diseño de un acelerador usando HLS.

4.1.1. Diseño funcional

El proceso comienza con el diseño y la verificación funcional del algoritmo completo en software. El presente trabajo considera que ya se cuenta con un diseño funcional del controlador que cumple con la calidad del control deseada para la aplicación. Desde el enfoque de la implementación, nos interesa distinguir las secciones del algoritmo que pueden procesarse *offline* de aquellas que deben procesarse *online*. Y con esto se identifican las variables constantes de la aplicación. Para esta etapa utilizamos MatLab (2023b), pero se podría usar otro lenguaje de alto nivel como Python.

Como resultado de este paso se obtienen:

- Archivos de entradas y salidas de referencia, que denominamos *golden references*, para verificaciones funcionales posteriores; y
- Archivos con los valores de las variables constantes, que se incluirán en los códigos fuente de la síntesis de alto nivel.

Por otra parte, la sección del algoritmo MPC que debe ejecutarse durante la operación de la planta se porta a C++, para poder ser implementada en hardware. La herramienta Vitis HLS permite verificar funcionalmente el código C++ mediante simulación, ejecutándolo como software y comparando los resultados con las referencias. Ya que se utilizan datos en punto flotante, es esperable que haya un porcentaje de error no nulo. Esto es porque las funciones que se escribieron en C++ para operaciones de álgebra lineal son diferentes a las de MatLab y porque las operaciones con punto flotante no son asociativas.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
mpc				-0,12	757	7,570E3	-	758	-	no	0	202	33440	25965	0
mpc_dense_constraint				-	25	250.000	-	25	-	no	0	10	3068	2145	0
mpc_dense_constraint_Pipeline_vmmult_column				-	13	130.000	-	13	-	no	0	0	406	273	0
vmmult_column				-	11	110.000	8	1	4	yes	-	-	-	-	-
mpc_dense_constraint_Pipeline_constraint1				-	10	100.000	-	10	-	no	0	0	262	91	0
constraint1				-	8	80.000	1	1	8	yes	-	-	-	-	-
mpc_dense_constraint_Pipeline_constraint2				-	10	100.000	-	10	-	no	0	0	267	121	0
constraint2				-	8	80.000	1	1	8	yes	-	-	-	-	-
mpc_dense_constraint_Pipeline_constraint3				-	6	60.000	-	6	-	no	0	0	138	96	0
constraint3				-	4	40.000	1	1	4	yes	-	-	-	-	-
mpc_dense_constraint_Pipeline_constraint4				-	6	60.000	-	6	-	no	0	0	138	96	0
constraint4				-	4	40.000	1	1	4	yes	-	-	-	-	-
qp_admm	II Violation			-0,12	728	7,280E3	-	728	-	no	0	156	25917	19587	0
loop_admm	II Violation	Memory Dependency	1	-	726	7,260E3	79	72	10	yes	-	-	-	-	-
vsub_24_float_s				-	3	30.000	-	1	-	yes	0	48	7753	5472	0
vadd_24_float_s				-	3	30.000	-	1	-	yes	0	48	7753	5472	0

Figura 4.1: Ejemplo de reporte de HLS.

4.1.2. Diseño de hardware con HLS

La herramienta Vitis HLS (2022.2) permite obtener un diseño de hardware en HDL a partir de un código C++ mediante síntesis de alto nivel. Además del código, solo es obligatorio proporcionar un dispositivo objetivo (FPGA) y una frecuencia de muestreo objetivo. Esta es la etapa más relevante para este trabajo, ya es el momento en que podremos ingresar directivas de síntesis –o *pragmas*– para influir en la arquitectura del hardware que implementa la herramienta, pudiendo gestionar el grado de paralelización del hardware resultante y con eso optimizar el acelerador en latencia y uso de recursos. Al ejecutar la síntesis de alto nivel, si no se proveen *pragmas*, la herramienta aplica algunos por defecto, realizando optimizaciones automáticas en base a heurísticas. El usuario puede agregar *pragmas* manualmente directo en el código C++ o mediante un archivo .tcl, sin que estos modifiquen la funcionalidad del código. Junto con el diseño del acelerador en HDL, la herramienta provee reportes de desempeño del resultado, donde se detalla la latencia y uso de recursos aproximado asociado a cada módulo del diseño. La Figura 4.1 muestra un ejemplo de reporte de HLS, donde se logra ver en detalle el impacto de cada módulo en la latencia total. Notar que las implementaciones de bucles (cuadrados azules) están nombradas con las etiquetas que les asignamos en el código. El uso de recursos de cada módulo no es tan simple de apreciar porque no se reporta la reutilización de recursos. Además, la herramienta permite visualizar la planificación de las operaciones y destaca la dependencias de datos que restringen la paralelización.

Por último, Vitis HLS ofrece la posibilidad de simular el comportamiento de la descripción de hardware obtenida, mediante co-simulación. Esto permite verificar que el acelerador sigue comportándose como el algoritmo original. Una vez que se cumplen los requisitos de latencia y uso de recursos y que se verifica funcionalmente, el diseño se exporta como un bloque llamado Propiedad Intelectual (Intelectual Property o IP).

4.1.3. Implementación en SoC

Empaquetamos el resultado de la etapa de HLS en una IP para incluir fácilmente el acelerador en un diseño RTL en Vivado (2022.2), y realizar posteriormente la síntesis lógica y la implementación. Lo hacemos mediante un diseño de bloques o *Block Design* como el de la Figura 4.2. Como se ve en el diagrama, además de la IP del acelerador diseñado, se incluye un bloque que represente la CPU del SoC objetivo. Usar este bloque permite conectar el acelerador a la CPU para hacer uso del sistema heterogéneo completo. El proceso es relativamente sencillo y se encuentra bien documentado, y la herramienta de diseño posee soporte para agregar y conectar automáticamente el bloque que representa la interfaz AXI que comunica la CPU con

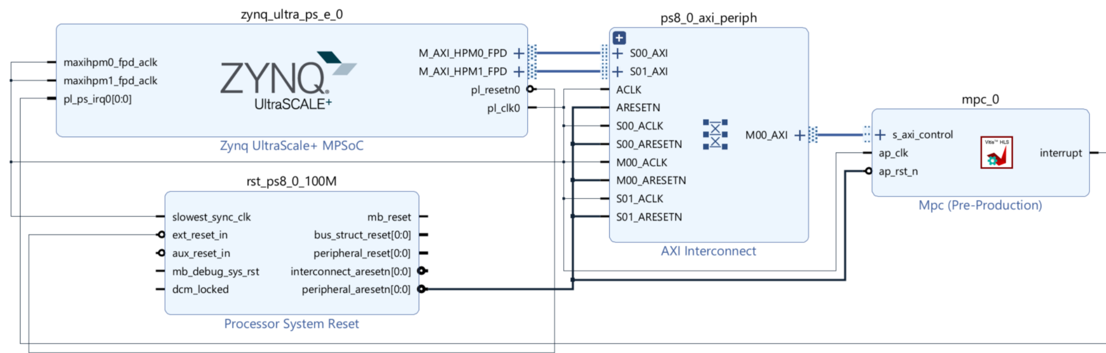


Figura 4.2: Ejemplo de Block Design en Vivado.

el acelerador en la FPGA.

En este punto es posible agregar otras IPs o bloques escritos manualmente en RTL. Por ejemplo, se pueden incluir periféricos para sensores o actuadores del controlador en la misma FPGA. Terminado el diseño de bloque, y posterior a la síntesis lógica e implementación, se obtiene el archivo binario que configurará la FPGA.

4.1.4. Ejecución del acelerador

En el diseño heterogéneo realizado, el acelerador implementado en la FPGA está subordinado al procesador. Luego, existen diferentes formas para diseñar una aplicación de software que se ejecute en la CPU y acceda al acelerador. La herramienta Vitis permite diseñar aplicaciones *baremetal*, ejecutables directamente sobre el procesador, o aplicaciones para sistemas operativos Petalinux. Vitis facilita el diseño de una aplicación que pueda utilizar el acelerador a través de AXI, así como el proceso de cargar la configuración de la FPGA junto con la aplicación.

Alternativamente, para algunos dispositivos SoC existe un sistema operativo precompilado llamado Pynq, que contiene un kernel de Python y las librerías necesarias para utilizar los aceleradores desde Python. Con Pynq se pueden cargar configuraciones en la FPGA y diseñar drivers para los aceleradores desde Jupyter Notebooks. Pynq también facilita el acceso remoto desde otro computador para probar el funcionamiento de los controladores con facilidad.

4.2. Exploración de soluciones usando pragmas

Además del uso de *pragmas*, el usuario puede afectar el hardware implementado con HLS mediante la elección de la frecuencia objetivo. El presente trabajo se centra en la optimización de aceleradores de hardware mediante *pragmas*, por lo cual trabajamos con una frecuencia de reloj fija. El estudio preliminar presentado en el Anexo B sugiere que la frecuencia de reloj se puede ajustar después de aplicar los *pragmas* y que con la herramienta Vitis HLS conviene empezar con la frecuencia por defecto de 100MHz. Esta es también la frecuencia del reloj de la tarjeta de desarrollo con nuestro SoC objetivo, que es un Zynq Ultrascale+ XCZU7EV MPSoC. Durante el trabajo mantenemos entonces una frecuencia objetivo fija de 100 MHz, con la incertidumbre por defecto de 27 %, lo que nos permite medir el desempeño del acelerador en términos de ciclos de reloj.

La exploración de soluciones se aplica al diseño de controladores MPC para un motor DC que es un caso de estudio muy utilizado en la literatura [10]. Después de verificar funcionalmente los Algoritmos 1 y 2 implementados en MatLab, se exportan las matrices constantes específicas para diferentes horizontes de predicción. La implementación en C++, que es la entrada al proceso de HLS, está parametrizada respecto al tamaño de la planta y decisiones de diseño de la aplicación (ver Tabla 3.2). El código utilizado para el caso del motor DC simplifica la formulación descrita en el Capítulo 3, de forma que se considera un problema de regulación con restricciones de caja. Esto no afecta el costo computacional de la resolución del problema QP. Además, cabe destacar que todas las soluciones presentadas incluyen una interfaz tipo AXI, la cual habilita la comunicación entre el acelerador y la CPU, y con eso una rápida verificación funcional de las implementaciones en hardware.

Comenzamos por aplicar la directiva *pipeline*. Pruebas preliminares mostraron que para el caso de estudio, donde los bucles incluyen operaciones de punto flotante de 4 o 5 ciclos de reloj, aplicar *pipeline* reduce la latencia con un efecto insignificante en el uso de recursos, como se conjeturaba en la Sección 3.2. De hecho, la herramienta Vitis HLS está configurada por defecto para aplicar la directiva en todos los bucles, y así verificar durante la síntesis si es factible implementar pipeline. Hacemos entonces *pipeline* en todas las implementaciones de este trabajo, siguiendo la recomendación de AMD-Xilinx. Aunque se aplica *pipeline* por defecto, elegimos incluir la directiva de forma explícita en cada bucle porque así la herramienta entrega más información de las dependencias de datos o problemas de acceso a memoria que impiden alcanzar $II = 1$.

Ilustraremos la aplicación de las directivas *unroll* y *array partition* con el ejemplo de la MVM *row-wise* que se muestra en el Código 4.1. Se puede apreciar que el *unroll* del bucle externo (línea 8) tiene el mismo factor que los *array partition* de los arreglos sobre los cuales itera ese bucle, i.e. el vector R (línea 6) y la primera dimensión de la matriz A (línea 5). Como se adelantaba en la Sección 3.2, para que el desenrollamiento sea efectivo se debe adaptar la memoria para poder leer y escribir los datos necesarios de forma simultánea. Pruebas preliminares mostraron que, en la práctica, conviene usar el mismo factor para *unroll* y *array partition*. Por otra parte, el código incluye *array partition* para el vector B y la segunda dimensión de la matriz A, que son de tamaño M. Esto es porque la herramienta por defecto aplica desenrollamiento completo al bucle anidado y deja al usuario resolver el problema de acceso a la memoria con *array partition*. Si bien es posible revocar el *unroll* completo automático del bucle interno de la MVM, en general conservamos esta recomendación de la herramienta porque en la MVM *row-wise* es conveniente desenrollar el bucle interno.

El Código 4.2 ilustra la aplicación de la directiva *loop merge*. Esta permite fusionar el conjunto de operaciones de álgebra lineal de un *scope* definido con llaves y una etiqueta, e.g. *admm_merge1* en las líneas 6-10. La condición es que todas las operaciones estén constituidas por bucles con la misma cantidad de iteraciones y que, sí hay dependencias entre las operaciones, estas sean elemento-a-elemento. Es decir, si la i-ésima iteración de la operación A depende sólo de la i-ésima iteración de la operación B, se trata de una dependencia compatible con *loop merge*. Como se muestra en el código, algunas de las multiplicaciones de ADMM pueden incluirse en *loop merges*, sujeto a que se utilice la versión *row-wise* y se desenrolle completamente el bucle interno.

En el resto de esta sección se presentan resultados de implementación de aceleradores de MPC según las métricas definidas en la Sección 2.2.2. Se aceleran las etapas de formulación y

```
1  template<int N, int M, typename>
2  void mvmult(const T (&A)[N][M], const T (&B)[M], T (&R)[N]){
3  #pragma HLS ARRAY_PARTITION variable=A dim=2 type=complete
4  #pragma HLS ARRAY_PARTITION variable=B dim=1 type=complete
5  #pragma HLS ARRAY_PARTITION variable=A dim=1 type=cyclic factor=2
6  #pragma HLS ARRAY_PARTITION variable=R dim=1 type=cyclic factor=2
7      mvmult_row: for(int i=0; i<N; i++){
8  #pragma HLS UNROLL factor=2
9  #pragma HLS PIPELINE II=1
10         R[i] = 0;
11         mvmult_column: for(int j=0; j<M; j++){
12             R[i] += A[i][j] * B[j];
13         }
14     }
15     return;
16 }
```

Código 4.1: MVM tipo *row-wise* con *pragmas*.

resolución del problema QP del Algoritmo 2, para evaluar la capacidad de implementar todo el procesamiento *online* de MPC en FPGA. Aunque los *pragmas* se utilizan para paralelizar a nivel de módulos individuales, estas optimizaciones pueden afectar a otros módulo por la dependencia de datos, y con eso impactar la planificación a alto nivel de los módulos y la latencia total. Por lo tanto, los resultados de latencia y uso de recurso de módulos individuales no son adecuados para comparar y reportamos sólo los resultados del acelerador completo. Utilizamos *scripts* .tcl para explorar rápidamente conjuntos de implementaciones de los controladores MPC usando *pragmas*, de los cuales se presentan los principales resultados.

4.2.1. Ejemplo de referencia: motor DC

El motor DC es un sistema utilizado abundantemente en la literatura [10]. La señal de entrada PWM afecta la velocidad angular del rotor y se utiliza para controlar su posición angular, de forma que el modelo está compuesto por dos estados y una única actuación. Además, podemos establecer restricciones de caja para ambos estados y para la entrada, para un total de seis restricciones de desigualdad. De esta forma, se tiene un problema QP de dimensiones $N = 1 \cdot h$ y $M = 6 \cdot h$. En este ejemplo consideramos un problema de regulación, lo cual no afecta el costo computacional de resolver el problema QP ni el análisis del uso de *pragmas*.

Se realizó una primera exploración del espacio de soluciones aplicando *unroll* tanto de operaciones elemento-a-elemento (EOP) como de multiplicaciones matriz-vector (MVM). Mantenemos inicialmente el *unroll* completo por defecto del bucle interno de las MVM tipo *row-wise*, para tener $II = 1$ y seguir desenrollando el bucle externo. Los resultados de HLS obtenidos al desenrollar cada EOP y/o el bucle externo de cada MVM se presentan en la Figura 4.3, para el motor DC con horizonte de predicción $h = 8$. El recuadro 4.3a muestra la disminución de la latencia con el aumento del factor de *unroll*. Y los recuadros 4.3b y 4.3c muestran la evolución del uso de bloques DSP, FFs y LUTs de la FPGA. El efecto de desenrollar EOPs es como se espera de las ecuaciones 3.5 y 3.3, a diferencia del desenrollamiento de MVMs. En general, observamos que el desenrollamiento de EOPs disminuye más la latencia y requiere menos recursos que el desenrollamiento de MVMs. También se puede observar que los efectos de desenrollar EOPs y MVMs por separado, i.e. aceleración y aumento de recursos,

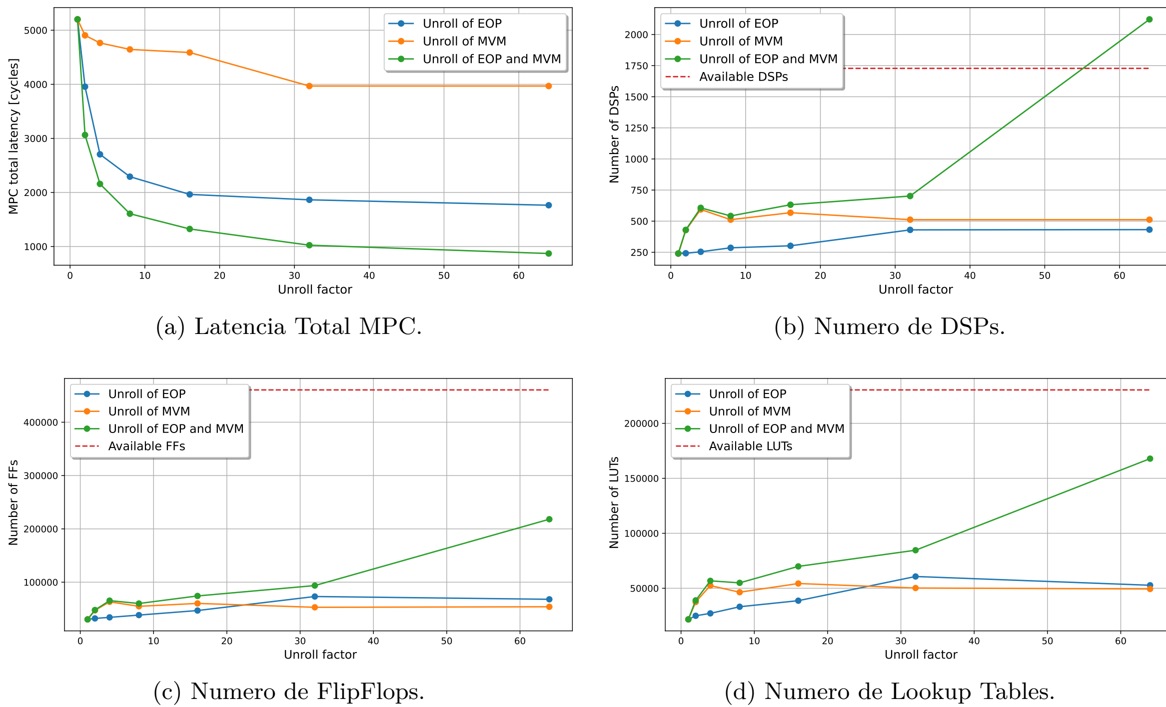
```
void qp_admm(data_t (&q)[N_QP], data_t (&g)[M_QP], int IT){
    loop_admm: for(int i = 0; i < IT; i++){
#pragma HLS LOOP_TRIPCOUNT max=10
        data_t vx[M_QP];
        data_t temp[M_QP];
        admm_merge1:{
#pragma HLS LOOP_MERGE force
            vsub<M_QP,data_t>(zk_admm, g, temp);        // temp = zk - g
            vadd<M_QP,data_t>(temp, uk_admm, vx);       // vx = temp + uk
        }
        data_t temp1[N_QP], temp2[N_QP], temp3[N_QP];
        admm_merge2:{
#pragma HLS LOOP_MERGE force
            mvmult<N_QP,M_QP,data_t>(P, vx, temp1);     // temp1 = -rho G^T *
    vx
            vsub<N_QP,data_t>(temp1, q, temp2);         // temp2 = temp1 - q
        }
        mvmult<N_QP,N_QP,data_t>(R_inv, temp2, tk_admm);// tk = R_inv * temp2
        data_t Gtk[M_QP], temp4[M_QP], temp5[M_QP], temp6[M_QP], temp7[M_QP];
        admm_merge:{
#pragma HLS LOOP_MERGE force
            mvmult<M_QP,N_QP,data_t>(G, tk_admm, Gtk);  // Gtk = G*kt
            vsub<M_QP,data_t>(g, uk_admm, temp4);       // temp4 = g - uk
            vsub<M_QP,data_t>(temp4, Gtk, temp5);       // temp5 = (g - uk) -
    Gtk
            vadd<M_QP,data_t>(uk_admm, Gtk, temp6);     // temp6 = uk + Gtk
            max0<M_QP,data_t>(temp5, zk_admm);          // zk = max{0, temp5}
            vsub<M_QP,data_t>(temp6, g, temp7);         // temp7 = (uk + Gtk)
    - g
        }
        vadd<M_QP,data_t>(temp7, zk_admm, uk_admm);     // uk = temp7 + zk
    }
    return;
}
```

Código 4.2: Implementación de ADMM con *pragmas*.

se sobreponen al desenrollar simultáneamente ambos conjuntos de operaciones, excepto para el último factor de *unroll*. Entre $F_{UN} = 32$ y $F_{UN} = 64$ hay poca diferencia en términos de uso de recursos para el desenrollamiento de EOPs y MVMs por separado, pero al desenrollar ambos conjuntos de operaciones el uso de recursos sigue aumentando. En particular, el número de DSPs aumenta dramáticamente, al punto que excede la disponibilidad de recursos en la FPGA objetivo.

Para apreciar mejor la relación costo-beneficio de las optimizaciones comparadas, se grafica en la Figura 4.4 la frecuencia de muestreo del controlador contra el uso de DSPs, que es el componente del hardware más relevante para procesamiento de punto flotante. Así se aprecia claramente que la paralelización de operaciones elemento-a-elemento es más costo-efectiva que las paralelización de multiplicaciones. Aquí también se observa el rendimiento decreciente de aumentar el factor de desenrollamiento.

Se exploran, además, otras optimizaciones usando *pragmas* para el ejemplo del motor DC, como fusionar bucles y deshabilitar el *unroll* automático. La Tabla 4.1 presenta un conjunto selecto de resultados de HLS, para un horizonte de predicción $h = 4$. La Solución 0



(a) Latencia Total MPC.

(b) Numero de DSPs.

(c) Numero de FlipFlops.

(d) Numero de Lookup Tables.

Figura 4.3: Resultados HLS de MPC, obtenidos al aplicar unroll parcial en las operaciones elemento-a-elemento (azul), multiplicaciones matriz-vector (naranjo) o ambas (verde).

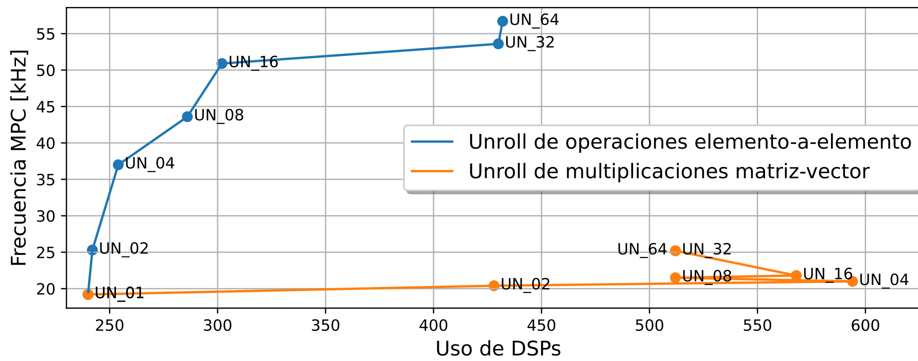


Figura 4.4: Frecuencia de muestreo contra uso de DSPs al aplicar unroll parcial de las operaciones elemento-a-elemento (azul) o de las multiplicaciones matriz-vector (naranjo).

Tabla 4.1: Resultados de frecuencia y uso de hardware post implementación de controlador MPC para motor DC, con horizonte 4 y 10 iteraciones de ADMM.

Sol.	Optimización	Frecuencia (kHz)*	DSP %	FF %	LUT %	BRAM %
0	Sin Unroll, MVM tipo column-wise	10	0.6	0.8	1.4	1.0
1	Unroll completo del loop interno de MVM tipo row-wise	36	6.9	2.4	3.6	0.0
2	Solución 1 + Loop_merge	58	6.8	2.3	3.3	0.0
3	Solución 1 + unroll parcial de EOP con factor 2	43	7.0	2.6	3.7	0.0
4	Solución 1 + unroll parcial de EOP con factor 16	75	8.8	4.6	7.1	5.1

* Frecuencia de muestreo, con 100MHz de frecuencia de operación de la FPGA.

es la implementación de referencia, con *pipeline* y sin *unroll* de ninguna operación. Se utiliza multiplicación *column-wise*, ya que evita la dependencia de datos en el bucle interno. La frecuencia de muestreo obtenida es de 10KHz, que es suficiente para controlar un motor DC. No obstante, las demás soluciones son relevantes para analizar los diferentes *pragmas* y el compromiso entre latencia y uso de recursos.

La Solución 1 corresponde al desenrollamiento por defecto del bucle interno de la MVM *row-wise*, que itera sobre las columnas. Desenrollar completamente el producto punto de las tres MVMs realizadas en la iteración de ADMM demuestra ser costoso en término de recursos, con 10 veces la cantidad de bloques DSP de la Solución 0. Sin embargo, la latencia se reduce más de tres veces y el uso de recursos se mantiene bajo 10 % para el dispositivo objetivo.

La Solución 2 se obtuvo al agregar la directiva *loop_merge* a la Solución 1, como se muestra en el Código 4.2. Combinando varios grupos de operaciones aritméticas en el algoritmo de ADMM se consigue una reducción tanto de latencia como de recursos, comparado con la Solución 1. Este resultado es particularmente bueno; sin embargo, la herramienta no parece soportar agregar desenrollamiento para seguir optimizando. Combinar *loop merge* y *unroll* en el mismo bucle requeriría editar el código fuente.

Por último, las Soluciones 3 y 4 muestran el efecto de agregar desenrollamiento parcial de las EOPs a la Solución 1, de manera similar a la Figura 4.3. Estos casos muestran una reducción de latencia contra un aumento de recursos, con rendimiento decreciente. Cabe destacar que la frecuencia de muestreo alcanzada es siete veces la de la implementación secuencial de la Solución 0, y el uso de recursos se mantiene bajo 10 % para la FPGA objetivo.

Con los recursos restantes, todavía es posible reducir la latencia paralelizando los bucles que siguen enrollados. Además, se pueden lograr optimización adicional adaptando el código C++ para ser más *hardware-oriented*, como se adelanta en el Anexo D, pero esto está fuera del alcance de este trabajo y se considera para trabajo futuro.

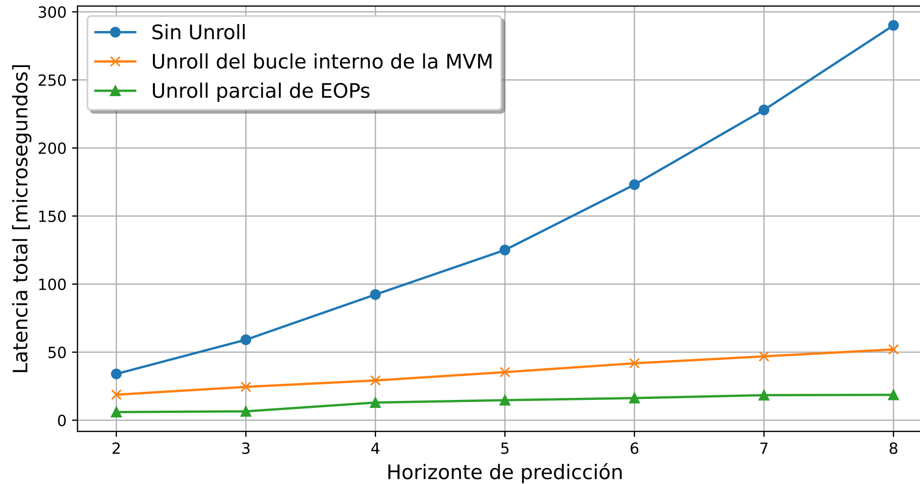


Figura 4.5: Escalabilidad respecto al horizonte de predicción.

4.2.2. Análisis de escalabilidad

Como hemos visto, el tamaño de los arreglos es un factor determinante en la latencia, ya que influye directamente en el número de iteraciones de los bucles. Hasta ahora hemos asumido que el tamaño de la aplicación es conocido y fijo, y nuestro objetivo ha sido explorar el espacio de implementaciones posibles para optimizar el diseño y alcanzar la frecuencia objetivo con bajo esfuerzo. Sin embargo, existen situaciones donde es deseable maximizar ciertos parámetros de la aplicación, lo que revela un compromiso entre diseño funcional del controlador y diseño de su implementación. En tales casos, resulta valioso analizar el desempeño de las optimizaciones al variar parámetros clave como el horizonte de predicción. Además, es también relevante considerar este análisis de escalabilidad para evaluar las limitaciones de la HLS en la implementación de MPC a alta frecuencia en plantas de mayor tamaño. Aunque el motor DC se puede controlar con un horizonte de predicción pequeño ($h = 2$), podemos usarlos para analizar la escalabilidad de la implementación de hardware aumentando la dimensión del problema QP con el horizonte.

La Figura 4.5 muestra la evolución de la latencia para tres estrategias de paralelización, respecto al aumento del horizonte de predicción del controlador. La curva azul corresponde a una implementación sin desenrollamiento, donde las operaciones de álgebra lineal se ejecutan en *pipeline*. Se aprecia que la latencia crece cuadráticamente con el horizonte, ya que la cantidad de operaciones atómicas de punto flotante en las multiplicaciones matriz-vector crece cuadráticamente. La implementación de la curva naranja tiene desenrollamiento completo del bucle interno de las multiplicaciones, de forma que el crecimiento pasa a ser lineal. Por último, para obtener la curva verde se agrega desenrollamiento parcial de las operaciones elemento-a-elemento con factor $F_{UN} = 32$. Con esto se mantiene el crecimiento lineal de la latencia respecto al horizonte de predicción, pero se consigue disminuir la taza de crecimiento.

Adicionalmente, se compara la escalabilidad de la implementación en hardware y en software. La Tabla 4.2 muestra tiempos de ejecución de MPC para dos implementaciones en hardware diseñadas con HLS y una implementación de software, con respecto al horizonte. La versión en software es compilada con el mismo código fuente C++ utilizado para HLS y ejecutada en un computador de escritorio con un procesador Intel i5 de 7ma generación.

Tabla 4.2: Tiempo de ejecución de MPC para implementaciones de software y hardware respecto al horizonte de predicción.

Horizonte de predicción	Tiempo de ejecución [μs]				
	Implementación en software			Hardware	Hardware
	Promedio	Desv. est.	Máx	Solución base	Full unroll
2	9,6	17,8	350,0	18,8	6,6
3	16,6	61,4	1893,6	23,1	7,2
4	19,1	18,9	333,1	29,2	8,2
5	26,4	22,3	245,3	33,8	8,3
6	34,4	45,5	1017,3	40,3	8,7
7	41,2	30,7	367,6	45,4	8.8
8	52,7	57,3	1422,1	52,0	9,2

Ya que la latencia de la implementación en software no es constante, a diferencia de las versiones en hardware, presentamos *Tiempo promedio*, *Desviación estándar* y *Tiempo máximo* de ejecución. La implementación por defecto en hardware, con las mismas optimizaciones de la Solución 1 de la Tabla 4.1, tiene peor desempeño que el tiempo promedio de ejecución en software, pero crece más lento con la dimensión del problema, hasta que iguala el desempeño del software en horizonte $h = 8$. La implementación en hardware fuertemente paralelizada, con desenrollamiento completo de EOPs y MVMs, es más rápida que el tiempo promedio de ejecución del software y es la versión de las tres que crece más lento con el horizonte. Sin embargo, desenrollar completamente operaciones de álgebra lineal tan grandes resulta prohibitivo en términos de uso de recursos. Los bloques DSP requeridos para implementar el controlador completamente desenrollado para el motor DC con horizonte $h = 8$ excede los recursos disponibles, y requeriría un dispositivo objetivo más grande para ser implementado.

5 | Caso de estudio: MPC para Distributed Energy Resource

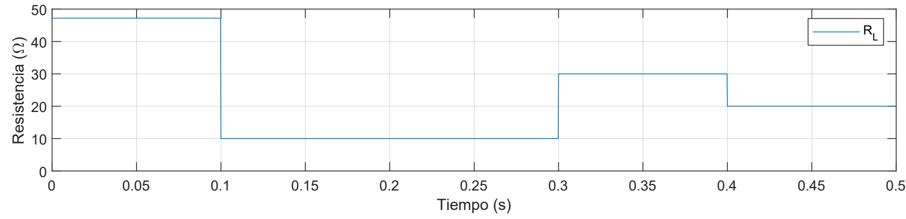
Este capítulo presenta los principales resultados obtenidos al aplicar la metodología propuesta al diseño de un controlador MPC para el filtro de un sistema DER implementado en el SoC objetivo. El DER es una aplicación de interés para el grupo de investigación [12, 13]. Después de describir la aplicación, se presenta una caracterización de la comunicación CPU-FPGA, evaluando su costo en latencia para una implementación en sistema heterogéneo. Con esta información se puede diseñar el acelerador con la latencia adecuada usando HLS. Por último, se presentan resultados de validación del controlador implementado en el SoC y se analiza la escalabilidad de la solución.

5.1. Descripción de la aplicación.

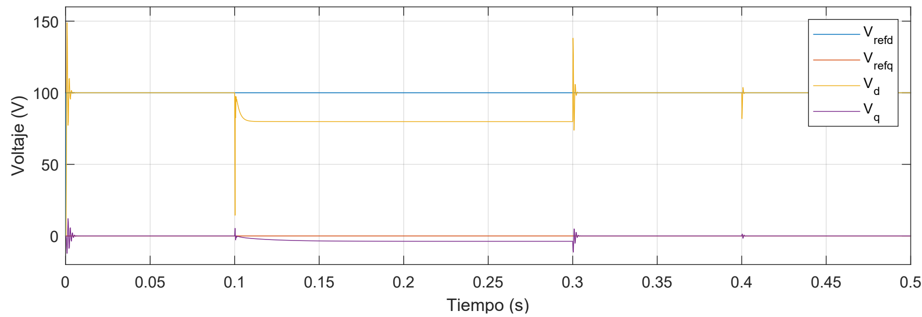
En una red con múltiples fuentes de energía –o DERs–, el convertidor a la salida de un recurso, e.g. de un conjunto de paneles solares, debe ser capaz de igualar la magnitud y frecuencia del voltaje de la red. Además, se requiere proteger el sistema ante cambios repentinos en la carga de la red, que pueden resultar en sobrecargas transitorias de corriente que dañen el sistema. Por lo tanto, MPC se presenta como un método atractivo para garantizar la regulación del voltaje y el cumplimiento de las restricciones.

El modelo del sistema DER utilizado comprende dos estados de voltaje, dos estados de corriente y dos entradas de voltaje, que corresponden a señales del filtro a la salida del convertidor. Los estados de corriente y las entradas tienen restricciones politópicas [28], es decir, conjuntos de restricciones que forman polígonos regulares. Las dimensiones del problema QP asociado son $N = 2 \cdot h$ y $M = 2(a + b) \cdot h$, donde $2(a + b)$ es la cantidad de restricciones. Idealmente se busca que los polítopos tengan la mayor cantidad de restricciones para aproximar un círculo y mejorar el desempeño del convertidor. De acuerdo al trabajo en [13], usar polítpos de 10 lados es adecuado para la aplicación, de forma que consideramos inicialmente $a = b = 5$, tal que $M = 20 \cdot h$. Además, tenemos que el periodo de muestreo objetivo para el lazo de control es de $200\mu s$.

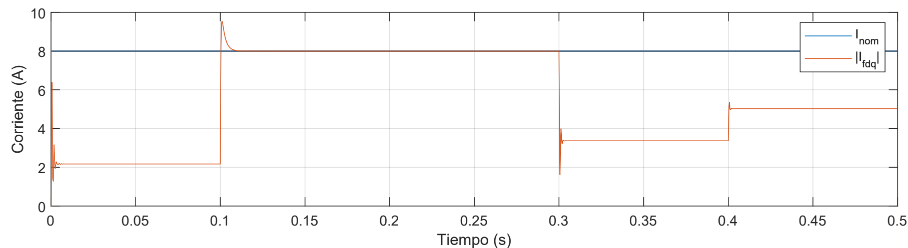
Comenzamos por verificar el comportamiento de salida del controlador simulando el sistema en MatLab. La Figura 5.1 muestra los estados de voltaje y la magnitud de la corriente durante la operación del DER en que se introducen cambios a la carga. La curva de carga presentada en el recuadro 5.1a presenta cambios bruscos en los instantes 0.1, 0.3 y 0.4s que modifican la magnitud de la corriente requerida para mantener el nivel de tensión. En el recuadro 5.1b se observa que el controlador efectivamente logra regular los estados de voltaje, salvo en el



(a) Carga R_L en el tiempo.



(b) Voltajes del filtro con sus referencias.



(c) Magnitud de la corriente y corriente nominal.

Figura 5.1: Verificación funcional del MPC para DER en MatLab.

intervalo entre 0.1 y 0.3s. Durante este periodo la carga se encuentra en su nivel más bajo y podemos observar en el recuadro 5.1c que la magnitud de la corriente se encuentra saturada en su valor nominal. Se verifica entonces el cumplimiento de las restricciones y el comportamiento general del controlador. Sin embargo, existe un transciente después del instante 0.1s donde la magnitud de la corriente sobrepasa la restricción. Como se explica en [13], el pico de corriente produce pérdidas y posibles daños al sistema que se logran mitigar disminuyendo el área bajo el pico de corriente al aumentar la cantidad de iteraciones de ADMM. En base a los resultados de [13], se considera que 10 iteraciones de ADMM es una cantidad adecuada para la aplicación.

5.2. Implementación del controlador en sistema heterogéneo

En el contexto en que parte del controlador MPC se implementa en la CPU y parte en la FPGA, debemos considerar la comunicación entre CPU y FPGA dentro del lazo de control. Nos interesa caracterizar la latencia de comunicación para determinar la factibilidad de implementar controladores con arquitectura heterogénea facilitar la decisión del usuario de cómo distribuir el procesamiento entre CPU y FPGA.

Como se menciona en la Sección 2.2.2, si omitimos las latencias de sensado y actuación,

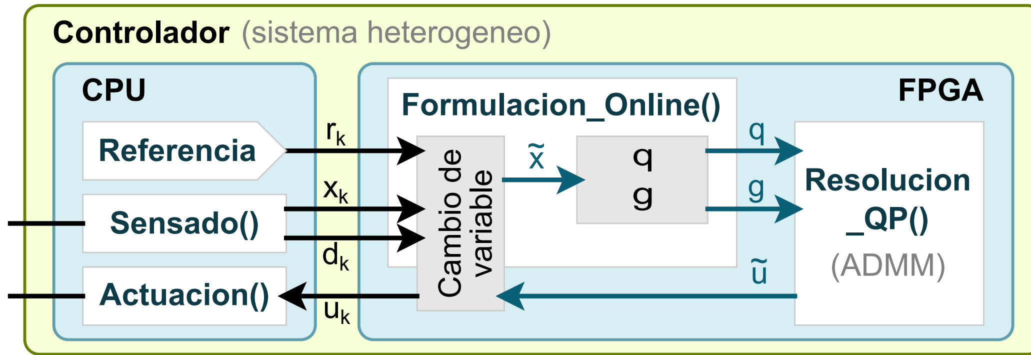


Figura 5.2: Diagrama de bloques de MPC en un sistema heterogéneo.

podemos expresar la latencia del lazo L_{MPC} como:

$$L_{MPC} = L_{SW} + L_{Com} + L_{HW} \tag{5.1}$$

donde L_{SW} y L_{HW} son las latencias del procesamiento en software y del acelerador respectivamente, que en conjunto ejecutan la formulación y resolución del problema QP, y L_{Com} es la latencia de la comunicación entre CPU y FPGA, la cual incluye tiempo efectivo de comunicación y *overhead* del software, es decir, el tiempo que se demora la aplicación en acceder a la interfaz de comunicación con la lógica programable.

Por simplicidad, comenzamos considerando el siguiente escenario:

- El sensado del estado y perturbaciones de la planta están disponibles en un *buffer* en la CPU;
- El procesamiento completo del controlador MPC, i.e. la formulación y resolución del problema QP, está acelerado en hardware;
- La entrada calculada debe llegar a la CPU que se encarga de aplicarla a la planta mediante un *buffer*.

El diagrama de bloques de la Figura 5.2 muestra las etapas de procesamiento de MPC del Algoritmo 2, distribuidas entre la CPU y la FPGA que componen el sistema heterogéneo en que se implementa el controlador. Las flechas negras representan los datos transmitidos entre ambas partes. En este escenario, la CPU proporciona la referencia, el estado medido (o estimado) y las perturbaciones medidas y el acelerador retorna la actuación calculada.

Considerando entonces que no hay procesamiento en la CPU, para definir la latencia objetivo del acelerador en hardware nos falta estimar la latencia de la comunicación.

5.2.1. Caracterización del tiempo de comunicación CPU-acelerador

El protocolo AXI define una latencia de comunicación por cada transferencia. En el supuesto de que transmisor y receptor estén listos, la transferencia AXI, que se muestra en la Figura 5.3, demora dos ciclos de reloj. Una transacción de escritura o lectura de un dato individual con AXI Lite requiere dos transferencias, una para la dirección y otra para el dato. Por lo tanto, utilizando transacciones individuales se tiene que la comunicación AXI requiere cuatro ciclos de reloj por dato.

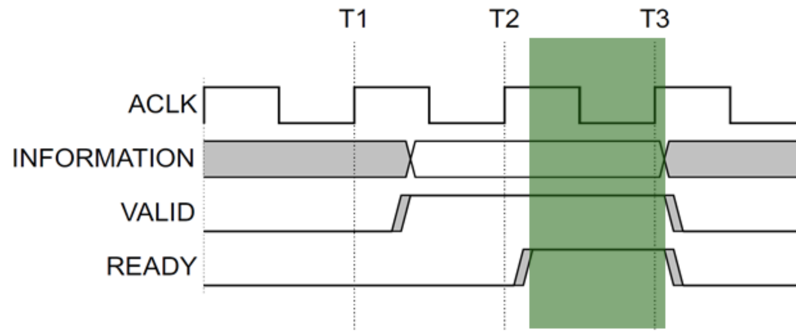


Figura 5.3: Transferencia AXI [29].

Además de la latencia de la transacción misma, la comunicación incluye una latencia asociada al tiempo que se demora la aplicación de software en solicitar transacciones a través de la interfaz AXI y leer o escribir los datos.

Setup experimental

Para medir el *overhead* del software en las transacciones AXI se agrega un bloque *sniffer* escrito en RTL en el diagrama de bloques. La Figura 5.4 muestra el diagrama de bloques en Vivado que representa el sistema heterogéneo, con la CPU y los distintos módulos a implementar en la lógica programable del SoC. El diagrama es idéntico al de la Figura 4.2, salvo que se expandió la conexión AXI para intervenir algunas de sus señales mediante un módulo adicional llamado *sniffer*. El *sniffer* está implementado en la misma FPGA y monitorea la conexión entre la CPU y el acelerador de MPC, contando los ciclos de intervalo entre transacciones consecutivas de la conexión AXI. Luego, desde la misma CPU se pueden acceder a los valores de tiempo almacenados en el *sniffer* mediante otra conexión AXI independiente. Además, las señales de interés se exponen a través de los conectores PMOD para poder observarlas con un analizador lógico externo.

La Figura 5.5 muestra un ejemplo de transacciones AXI y del intervalo medido por el *sniffer*. El contador del *sniffer* se reinicia con cada canto de las señales *VALID*, sea una transferencia de datos de lectura o transferencia de datos de escritura, y su valor se almacena en memoria. Se utiliza una aplicación de software que no hace nada más que realizar las transacciones de lectura y/o escritura, de forma que el intervalo de tiempo medido sea lo más cercano posible a la latencia efectiva de comunicación, incluido el *overhead* correspondiente que demora la aplicación de software en acceder a la interfaz AXI. Este método de medición de tiempo en hardware es más confiable y preciso que la alternativa de medir la comunicación desde la CPU en software.

Comunicación con aplicación Pynq

El sistema Pynq [30] ofrecido por AMD-Xilinx incluye un sistema operativo pre-compilado para la CPU del SoC utilizado, el cual incluye un kernel Python donde se pueden ejecutar programas que pueden acceder a los aceleradores disponibles en la FPGA mediante AXI. De esta forma, se puede fácilmente utilizar el acelerador de MPC mediante aplicaciones Python

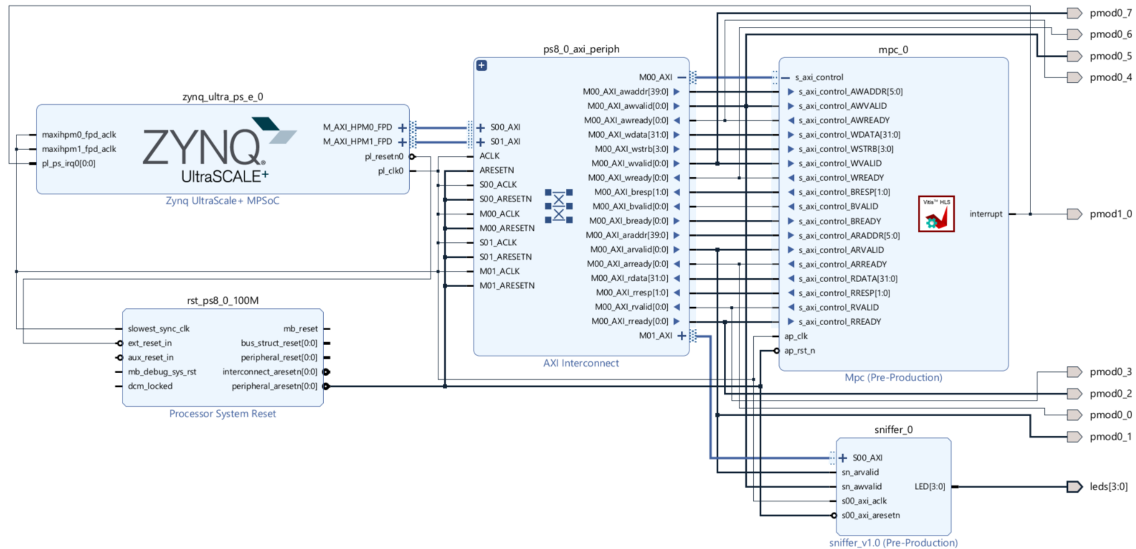


Figura 5.4: Block Design del controlador MPC en sistema heterogéneo con *sniffer*.

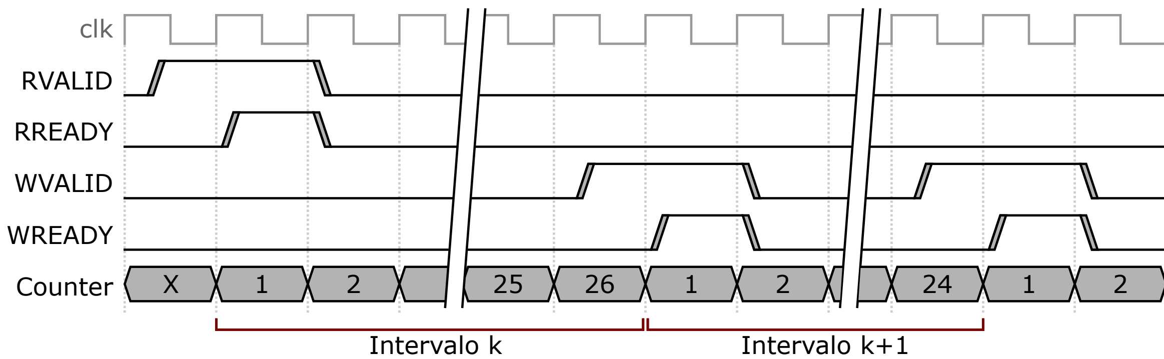


Figura 5.5: Intervalo medido entre transacciones AXI consecutivas.

usando Jupiter Notebooks[1].

Los resultados detallados de la medición de la comunicación se presentan en el Anexo C. La Tabla 5.1 muestra tiempos promedio para los cuatro tipos de intervalos posibles entre transacciones AXI. Se observa que el *overhead* del software entre dos transacciones AXI consecutivas depende del tipo de transacciones, de forma tal que la latencia entre una escritura y una lectura menor que entre una lectura y una escritura. Cabe destacar que los valores indicados en la tabla son promedios y que la latencia de comunicación con Pynq presenta gran varianza, al punto que se midieron intervalos de tiempo sobre los $60\mu s$. Esto implica que una sola transacción podría llegar a demorar 30 % del periodo de muestreo.

Para la implementación de un controlador para DER, de acuerdo con el diagrama de la Figura 5.2, en cada periodo de muestreo se escriben seis datos de 32 bits (cuatro estados, dos referencias y dos perturbaciones), se escribe la instrucción de inicio del acelerador y luego se leen dos datos de 32 bits (las actuaciones). Considerando las latencias promedio de las transacciones con Pynq, la latencia total de la comunicación se estima en $L_{Com} = 82,76\mu s$.

[1] https://jupyter.org/

Tabla 5.1: Intervalos de tiempo entre transacciones AXI usando Pynq.

Intervalo	Tiempo promedio (μs)
escritura-escritura	9,07
lectura-lectura	8,84
escritura-lectura	5,75
lectura-escritura	13,75

Tabla 5.2: Intervalos de tiempo entre transacciones AXI usando baremetal (Vitis).

Intervalo	Tiempo máximo (μs)
escritura-escritura	0,05
lectura-lectura	0,25
escritura-lectura	0,26
lectura-escritura	0,24

Luego, la latencia máxima del acelerador para DER sería de $110\mu s$.

Comunicación bare-metal

Como alternativa al uso de Pynq, es posible implementar una aplicación de software que se ejecute de forma independiente en la CPU del SoC –o aplicación *bare-metal*– utilizando la herramienta Vitis. Esta opción requiere más trabajo, pero ofrece mayor control del procesador y se evita el *overhead* asociado a utilizar un sistema operativo.

Se repitió el experimento para medir los intervalos de comunicación AXI con una aplicación *baremetal*. Como se esperaba los tiempos de comunicación son menores que con Pynq y se reduce la varianza significativamente. De hecho, la varianza es casi nula. En la Tabla 5.2 se presentan los valores máximos medidos para los cuatro tipos de intervalos en la comunicación. Cabe destacar que el intervalo entre operaciones de escritura consecutivas es de cinco ciclos de reloj, lo que corresponde casi a la latencia mínima, que está dado por los cuatro ciclos necesarios para la transacción AXI. Este resultado sugiere que la aplicación de software provee los datos a escribir y accede a la interfaz AXI más rápido que la velocidad de transmisión de los datos al acelerador. En consecuencia, la latencia total de la comunicación en el caso *baremetal* se estima en $L_{Com} = 1,15\mu s$, de forma que la latencia máxima para el acelerador para DER sería de $198\mu s$.

5.2.2. Diseño de acelerador con HLS

Una vez que determinamos la latencia máxima que puede tener el acelerador de MPC para DER, procedemos con su diseño usando HLS. A partir de los resultados de la exploración de soluciones del Capítulo 4, aplicamos las estrategias más relevantes al controlador de DER. La Tabla 5.3 muestra los resultados de las soluciones principales comparadas con un resultado de referencia reportado en [13], donde utilizaron el mismo flujo con HLS, que coincide en tamaño del horizonte, número de iteraciones de ADMM y FPGA objetivo. La Solución 1, que corresponde a nuestra implementación por defecto del controlador, tiene una latencia del acelerador de $46.5\mu s$, lo que es ligeramente superior a la latencia de $45\mu s$ alcanzada

Tabla 5.3: Resultados de latencia y uso de hardware post HLS de controlador MPC para DER, con horizonte $h = 2$ y 10 iteraciones de ADMM.

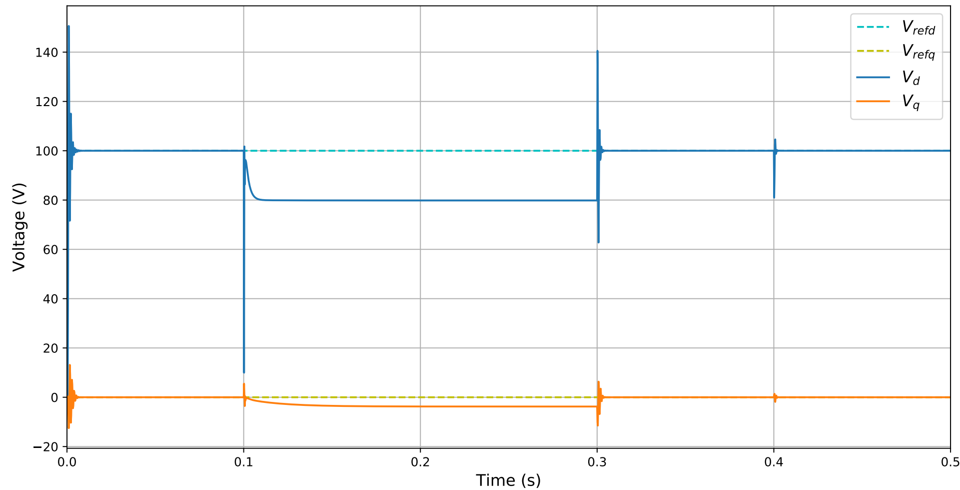
Sol.	Optimización	Latencia μs	DSP %	FF %	LUT %	BRAM %
-	Implementación previa del grupo [13]	45	18.5	15.3	35.7	28.2
1	Unroll completo del loop interno de MVM tipo row-wise	46.5	11.6	6.0	9.8	5.1
2	Solución 1 + loop merge	24.4	11.5	5.9	9.4	2.9
3	Solución 1 + unroll parcial de EOP con factor 2	29.4	11.7	6.2	10.8	1.3
4	Solución 1 + unroll parcial de EOP con factor 16	18.5	17.0	9.8	17.7	1.3

en [13]. Sin embargo, en la medida que se aplican optimizaciones utilizando *pragmas*, las demás soluciones consiguen disminuir la latencia por debajo del trabajo de referencia. El análisis de optimizaciones costo-efectivas usando *pragmas* realizado para el controlador del motor DC en el Capítulo 4 sigue siendo válido para los resultados del controlador de esta planta. La Solución 2 nuevamente evidencia la utilidad del *loop merge* y las Soluciones 3 y 4 muestran retorno decreciente de la aceleración al aumentar el factor de desenrollamiento. En particular, se destaca que la latencia alcanzada en la Solución 4 representa una mejora de 59 % respecto a [13], para una utilización similar de DSPs y aproximadamente la mitad de FFs y LUTs.

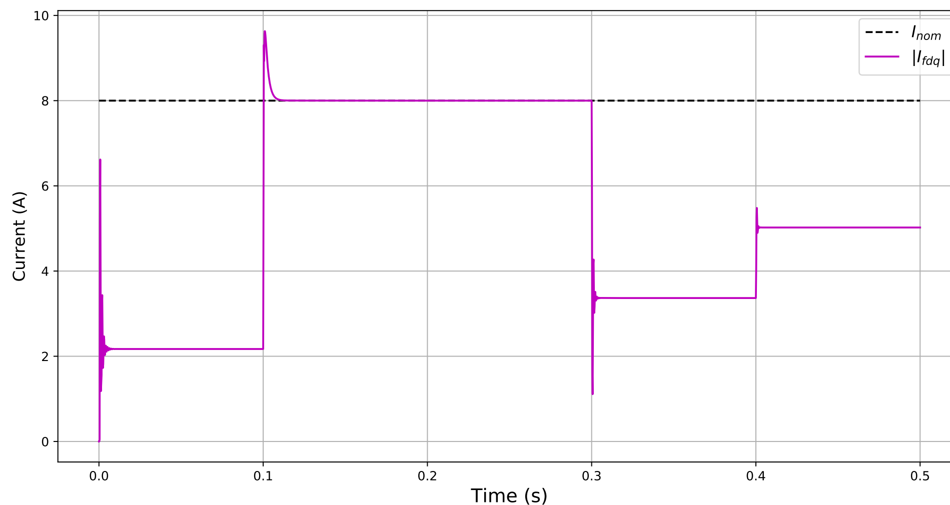
5.3. Validación del controlador en hardware

Durante el proceso de diseño, los controladores implementados con HLS se verificaron funcionalmente usando las herramientas Vitis HLS y Vivado. Se ejecutó MPC en el hardware simulado para comparar los resultados con las referencias obtenidas inicialmente en MatLab. Además, se verificó funcionalmente el acelerador implementado en el hardware real. El dispositivo SoC permite fácilmente probar el acelerador en una configuración tipo Hardware-in-the-Loop, ya que la aplicación de software que se ejecuta en la CPU puede emular el comportamiento de la planta a partir del modelo o proveer las entradas de la *golden reference*. La Figura 5.6 muestra el comportamiento del sistema DER emulado en CPU y controlado con un acelerador de MPC en hardware. Desde la CPU se provee un estado inicial y se emulan las perturbaciones en el tiempo y la actualización del estado de acuerdo a la acción de control calculada en el acelerador, para el mismo escenario de variación de carga presentado en 5.1a. El comportamiento del controlador implementado en FPGA que se observa en la figura valida la equivalencia funcional de la implementación.

De acuerdo a los resultados de la Sección 5.2.2, se consigue diseñar aceleradores con latencia inferior a $110\mu s$. Luego, sería factible implementar la aplicación en Pynq considerando la latencia promedio de la comunicación. La Figura 5.7 muestra un ejemplo de la comunicación para el control MPC, donde se transmiten ocho datos de entrada al controlador, seguidos de la instrucción de inicio y luego se leen dos datos. Se puede apreciar que la lectura de las actuaciones se realiza después de terminado el procesamiento del acelerador, que está indicado por el canto positivo de la señal *interrupt*. El periodo de muestreo, medido hasta la



(a) Estados de voltaje del DER, con sus respectivas referencias.



(b) Magnitud de la corriente del DER, con su restricción.

Figura 5.6: Estados del sistema DER emulado en CPU con control acelerado en FPGA.

primera escritura de la siguiente iteración, es de $108\mu s$, que es aproximadamente la mitad del periodo objetivo. Sin embargo, sabemos que las transacciones tiene una varianza importante, con latencias máximas que sobrepasan los $60\mu s$, lo que no es adecuado para su aplicación en tiempo-real, dado el periodo de muestreo objetivo de $200\mu s$. Esta significativa varianza en la comunicación entre CPU y FPGA usando el sistema Pynq es atribuible al sistema operativo, que distribuye la atención del procesador entre diferentes procesos, y al kernel de Python.

Por otra parte, la aplicación *baremetal* ofrece tiempos de comunicación con una latencia de peor caso mucho menor. Con la aplicación *baremetal* resulta factible implementar controladores MPC en sistemas heterogéneos considerando la latencia de comunicación entre CPU y FPGA mediante AXI Lite. Eventualmente, puede haber casos en que sea útil o necesario un sistema operativo, por ejemplo, para utilizar protocolos de red. Como tercera alternativa, se podría usar un sistema operativo orientado a tiempo real, que a diferencia de Pynq ofrecería mayores garantías de tiempo-real, pero la aplicación requeriría mayor tiempo de diseño.

Figura 5.7: Transacciones AXI entre CPU y FPGA para al control MPC de DER: ocho datos de entrada, activación del acelerador y dos datos de salida.

Utilizando entonces la aplicación *baremetal*, contábamos con un presupuesto de latencia de $198\mu s$ para el acelerador, pero en la Sección 5.2.2 hemos visto que este puede implementarse con latencias menores a $20\mu s$. Estos resultados sugieren que sería posible procesar un problema MPC de mayor costo computacional con el dispositivo SoC utilizado y respetando la restricción del periodo de muestreo objetivo. Por una parte, sería posible aumentar el número de iteraciones de ADMM. Utilizando el mismo controlador implementado en base a la Solución 4 de la Tabla 5.3, tenemos que la latencia del hardware está dada por $L_{HW} = 1.36 + IT_{ADMM} \cdot 1.71\mu s$. Luego, podemos ejecutar MPC con hasta 114 iteraciones de ADMM en el periodo de muestreo. Por otra parte, es interesante analizar la posibilidad de aprovechar el margen de tiempo en otro tipo de mejoras funcionales del controlador. En particular, para la aplicación del DER, es útil aumentar la cantidad de restricciones que forman los polítopos.

5.4. Escalabilidad respecto al número de restricciones

Si bien, no se cuenta con una referencia funcional específica para evidenciar el interés de aumentar el número de restricciones, sabemos que dentro del grupo de investigación existe interés por aumentar la cantidad de restricciones para el control del DER. En ese contexto, resulta relevante evaluar la escalabilidad del acelerador de MPC en términos del número de restricciones.

La Figura 5.8 muestra la evolución de la latencia del acelerador para tres estrategias de optimización con *pragmas* con $\{8, 12, 16, 20, 24, 32\}$ restricciones. Se observa que las tres curvas crecen con orden 1. Esto es esperable porque, a diferencia del horizonte de predicción, la cantidad de restricciones solo influye en una dimensión del problema QP (en el parámetro M). La curva sin desenrollamiento es la que tiene mayor tasa de crecimiento. Al agregar desenrollamiento completo del bucle interno de la MVM se logra disminuir la pendiente de la curva. Luego, agregando también desenrollamiento parcial de EOPs con factor $F_{UN} = 16$, la pendiente baja aún más. El caso con ambas optimizaciones combinadas parece constante por la escala, pero mantiene un crecimiento de primer orden. Este tipo de exploración permite encontrar el máximo número de restricciones con que se podría alcanzar el objetivo de frecuencia, dada una cantidad de recursos disponibles.

En el gráfico, sólo la implementación para 32 restricciones sin desenrollamiento supera la latencia máxima requerida para el acelerador. Además, ninguna de las implementaciones del presentadas supera los recursos disponibles. Esto sugiere que aún existe margen para implementar controladores con mayor número de restricciones o aumentar la cantidad de

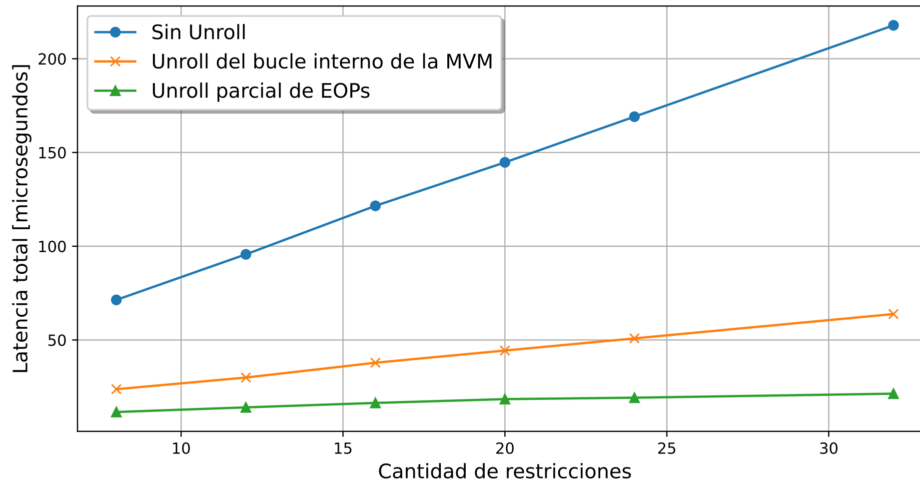


Figura 5.8: Escalabilidad de tres estrategias de optimización con HLS respecto al número de restricciones del sistema DER.

iteraciones de ADMM. También se evidencia la facilidad con que se puede implementar un acelerador con HLS para problemas de mediana escala con latencias inferiores al milisegundo.

6 | Conclusiones y trabajo futuro

En este trabajo se propone una metodología de diseño de aceleradores en lógica reconfigurable usando HLS para la implementación de un lazo de control MPC en una plataforma que integra CPU y FPGA. Después de analizar el costo computacional y el potencial de paralelización del algoritmo de MPC y de ADMM como *solver* QP, exploramos el espacio de posibles implementaciones utilizando *pragmas* de HLS, para exponer los compromisos entre desempeño en términos de latencia y costo en uso de recursos de hardware. Habiendo determinado estrategias costo-efectivas de paralelización de MPC, se aplica la metodología al diseño de un controlador para DER, como caso de estudio. En este punto también se caracteriza la latencia asociada a la comunicación entre CPU y FPGA, para evaluar si es posible implementar el controlador en la plataforma objetivo satisfaciendo el periodo de muestreo requerido para la aplicación.

En este capítulo se presentan los principales resultados y se plantean ideas para trabajo futuro.

6.1. Conclusiones

En esta tesis hemos adoptado un enfoque *top-down* de desarrollo de aceleradores de cómputo en FPGA para aplicaciones de MPC implícito, con el fin de acercar el diseño de hardware especializados a usuarios que no son expertos en sistemas digitales o descripción de hardware. Las herramientas de HLS efectivamente permiten orientar el trabajo desde la aplicación hacia el hardware. Sin embargo, aunque la HLS facilita el diseño y verificación de aceleradores, economizando tiempo y esfuerzo, se evidencia que sigue siendo necesario tener una noción de la arquitectura del hardware implementado para poder mejorar su desempeño. Proponemos entonces un análisis del algoritmo de MPC y ADMM para identificar el origen del costo computacional, deducir el potencial de paralelización del algoritmo y determinar directrices escalables para la optimización de aceleradores de MPC usando *pragmas* de HLS en base al ejemplo de un motor DC.

Los resultado de la exploración de posibles implementaciones demostró que, contrario a lo sugerido en la literatura [10, 15], el mayor costo computacional asociado a las multiplicaciones matriz-vector no implica que la paralelización de multiplicaciones sea la mejor estrategia para reducir la latencia del *solver* QP. Los resultados muestran que para la implementación de ADMM en hardware es más costo-efectivo paralelizar las operaciones elemento-a-elemento.

La estrategias de paralelización identificadas con el ejemplo del motor DC se aplicaron al diseño de un acelerador de MPC para el caso de estudio de un sistema DER. En este

proceso consideramos la implementación del controlador en un dispositivo de arquitectura heterogénea, compuesto por una CPU y una FPGA. Los resultados evidencian que en un dispositivo comercial de gama media que integra CPU y FPGA la latencia de comunicación entre la CPU y el acelerador es suficientemente baja como para contemplarla dentro del lazo de control sin dejar de cumplir el periodo de muestreo objetivo.

En general, se concluye que es posible diseñar aceleradores de cómputo en FPGA mediante HLS en latencias inferiores al milisegundo para aplicaciones de tamaño medio que contemplen comunicación con la CPU. Además, Se verifica que los dispositivos comerciales modernos cuentan con suficientes recursos de hardware como para implementar problemas computacionalmente complejos como MPC en punto flotante.

6.2. Trabajo futuro

A partir del trabajo realizado, reconocemos los siguientes temas de interés para trabajo futuro:

- Exploración del uso de punto fijo con HLS: El uso de punto fijo requiere un análisis de estabilidad del *solver* QP, para lo cual existe literatura. No obstante, falta evaluar la posibilidad de utilizar punto fijo en un flujo rápido de diseño usando HLS, que sea fácilmente escalable y portable entre aplicaciones. Para ello, además de poder usar *templates* de C++, se cuenta con librerías de AMD-Xilinx que ofrecen la posibilidad de usar tipos de datos no estandarizados.
- Evaluación de las capacidades de HLS para aprovechar la formulación *sparse* del problema QP: Análisis preliminares muestran que la herramienta optimiza, i.e. elimina, operaciones con resultado constante (como multiplicaciones por 0) en el hardware que se implementa desenrollado, pero esto no ocurre a nivel de planificación. Si la herramienta no consigue aprovechar adecuadamente la *sparsity* de las matrices, la alternativa será considerar formatos de compresión de datos en HLS.
- Aplicación de la metodología propuesta a MPC no-lineal: En aplicaciones no-lineales existe procesamiento serial o pobremente paralelizable, por lo cuál resulta más atractivo distribuir el procesamiento entre CPU y FPGA y la latencia de la comunicación entre ambas partes se vuelve más relevante en el diseño del lazo de control.
- Evaluación de estrategias de *hardware-oriented code*: En la literatura se han propuesto formas de escribir el código C++ para reducir el impacto de las dependencias de datos durante la HLS [15, 31], sin embargo, resulta necesario una comparación y un análisis más profundos de las condiciones en que estas estrategias resultan convenientes, en particular, en función del tamaño de los arreglos de datos.

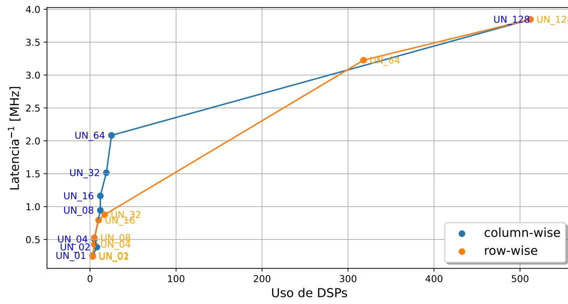
A | Multiplicación Matriz-Vector

Este anexo presenta un análisis preliminar de la implementación en hardware de la Multiplicación Matriz-Vector (Matrix-Vector Multiplication o MVM). En particular, comparamos la implementación de las formas *row-wise* y *column-wise* de escribir el código de la multiplicación, junto con las posibilidades de paralelización mediante *pragmas* de HLS. Cabe destacar que analizar el compromiso entre latencia y uso de recursos de una operación individual de álgebra lineal entrega información relevante del impacto que tiene en la latencia de un algoritmo, e.g. MCP, pero omite información relevante del impacto que tiene la dependencia de datos entre la multiplicación y otras operaciones de álgebra lineal en el algoritmo completo.

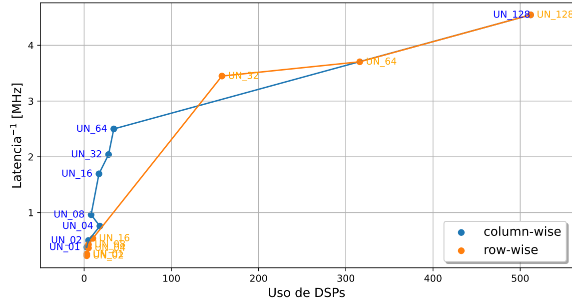
Consideramos que el costo computacional de la MVM está dado por la cantidad de operaciones atómicas de multiplicación y suma de punto flotante, que depende del tamaño de la matriz. Esencialmente, para una MVM con matriz de tamaño $X \times Y$, la cantidad de operaciones atómicas es proporcional a $X \cdot Y$, que es la cantidad de elementos de la matriz. En una implementación secuencial el tiempo de ejecución será prácticamente proporcional a la cantidad de elementos de la matriz, con poco o nulo efecto de la relación de aspecto de la matriz o de la forma de escribir el código (si ignoramos posibles efectos del almacenamiento de la matriz en memoria). Sin embargo, se presume que en la implementación en hardware, y particularmente en implementaciones paralelizadas, podría ser relevante la relación de aspecto de la matriz producto de las dependencias de datos.

Como se explica en la Sección 3.2, la MVM tiene una dependencia de datos interna entre las iteraciones del bucle que itera sobre las columnas, específicamente en la multiplicación-acumulación. Esta dependencia se encuentra en el bucle interno para la versión *row-wise* (Código 3.3) de la MVM y en el bucle externo para la versión *column-wise*(Código 3.2). En la Figura A.1 se presentan los resultados de implementación de la MVM *row-wise* y *column-wise* para distintas relaciones de aspecto y grados de desenrollamiento. En todos los casos la matriz tiene 128 elementos, de forma que la cantidad de iteraciones del bucle interno por la cantidad de iteraciones del bucle externo también es 128. Para cada relación de aspecto se evalúa el efecto de desenrollar primero el bucle interno y luego el externo, tal que el factor de desenrollamiento asociado a cada punto corresponde al producto entre los factores de desenrollamiento del bucle interno y externo, que denominaremos *factor de desenrollamiento combinado*. Los gráficos muestran la relación costo-beneficio de cada las implementaciones en términos del inverso de la latencia y el uso de bloques DSP. Además de desenrollamiento, se aplica *loop flatten* en la medida de lo posible. Para la versión *row-wise* esto es solamente en el caso sin desenrollamiento. Para la versión *column-wise* el *loop flatten* es posible mientras el bucle interno no esté completamente desenrollado.

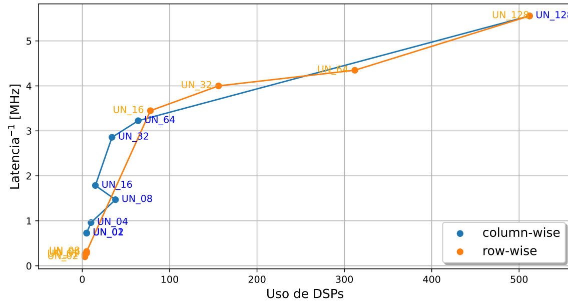
En general se observa que la versión *row-wise* tiene peor relación costo-beneficio cuando



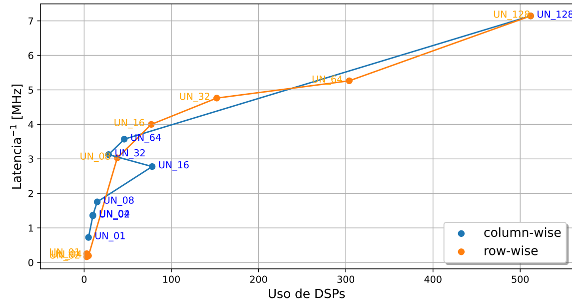
(a) MVM de tamaño 2x64.



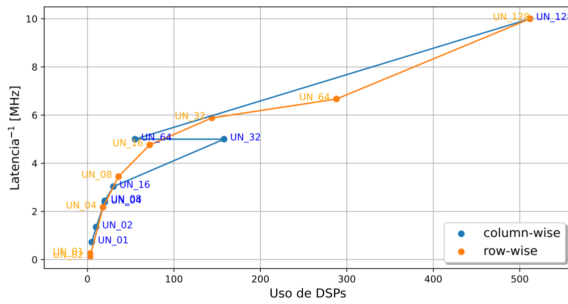
(b) MVM de tamaño 4x32.



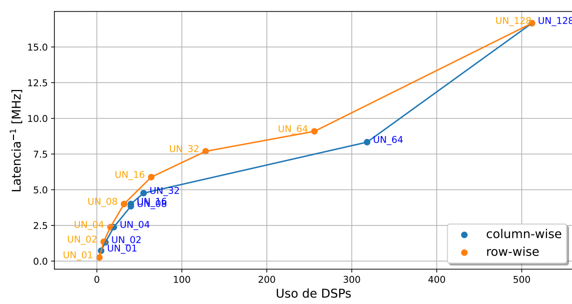
(c) MVM de tamaño 8x16.



(d) MVM de tamaño 16x8.



(e) MVM de tamaño 32x4.



(f) MVM de tamaño 64x2.

Figura A.1: Inversa de la latencia contra uso de bloques DSP al aplicar unroll parcial de la multiplicación matriz-vector con distintas relaciones de aspecto, tipo *column-wise* (azul) o *row-wise* (naranjo).

el bucle interno no está completamente desenrollado, lo que es esperable porque tiene la dependencia de datos en el bucle interno. En el caso de la versión *column-wise*, la dependencia de datos aparece cuando el factor de desenrollamiento combinado es alto. En los recuadros (c), (d) y (e) se aprecia que el costo-beneficio de la versión *column-wise* empeora cuando se desenrolla completamente el bucle interno, pero vuelve a mejorar cuando se empieza a desenrollar el bucle externo. La disminución del uso de bloques DSP implica que, aumentar el desenrollamiento permite balancear mejor las operaciones para reutilizar mejor el hardware y disminuir la latencia a pesar del deterioro del intervalo de inicialización. En particular, se destaca que para la relación de aspecto 64×2 en el recuadro (f), la versión *row-wise* es sistemáticamente mejor.

Considerando sólo el desempeño de la implementación de una MVM indiviual, podemos concluir que la versión *column-wise* de escribir el código presenta mejor relación costo-beneficio para la mayoría de grados de paralelización, salvo cuando el número de columnas es muy pequeño. Sin embargo, como se explica en [15], hay motivos para esperar que la versión *row-wise* limite el impacto de las dependencias de datos entre operaciones de álgebra lineal a nivel del algoritmo completo.

B | Frecuencia del reloj

Este anexo presenta un análisis preliminar de la elección de la frecuencia de reloj de la FPGA para optimizar el desempeño de un acelerador usando HLS.

En caso de usar datos en punto flotante, para la tecnología de los dispositivos utilizados en este trabajo, en general, se tiene que las operaciones de punto flotante implementadas en bloques DSP tienen una latencia de más de un ciclo de reloj. En la implementación de operaciones de álgebra lineal, la latencia de las operaciones atómicas de punto flotante es relevante en la latencia de una iteración individual y en el intervalo de inicialización cuando existen dependencias de datos.

La Figura B.1 muestra la evolución de la latencia de un acelerador de MPC al modificar la frecuencia de reloj objetivo de la HLS para tres tamaños del problema. Se observa que, en términos generales, disminuir el periodo del reloj disminuye la latencia, salvo que existen mínimos locales en que la latencia es menor y donde disminuir el periodo aumenta la latencia bruscamente, llegando a un máximo local desde donde vuelve a disminuir la latencia. Por ejemplo, para horizonte $h = 4$ con periodo $10ns$ se tiene una latencia de $92\mu s$ y al disminuir el periodo a $9.9ns$ la latencia aumenta a $107\mu s$. Al analizar los reportes de planificación de las operaciones identificamos una correlación entre los máximos y mínimos de la latencia y la cantidad de ciclos que demora la operación de suma de punto flotante. Se trata de la operación cuya latencia es determinante en las operaciones elemento-a-elemento y en la acumulación de la multiplicación. Por ejemplo, se cumple que con periodo $8.9ns$ la suma de punto flotante demora siete ciclos de reloj, pero con periodo $8.8ns$ pasa a demorar ocho ciclos. De esta forma, donde hay máximos locales ocurre que disminuir el periodo del reloj obliga a agregar etapas de registros, aumentando el número de ciclos y con eso la latencia, además del uso de recursos. Inversamente, los mínimos locales corresponden al mínimo periodo de reloj con que se puede ejecutar la suma de punto flotante en una cierta cantidad de ciclos, y con cierto uso de recursos.

Para efectos de optimizar el desempeño del acelerador, deseamos elegir periodos de reloj que coincidan con mínimos de latencia. Los resultados no sugieren que sea posible estimar matemáticamente a qué frecuencia se encuentran los mínimos locales, pero se observa que dichas frecuencias se mantienen independientes del costo computacional del problema ejecutado, de forma que la exploración de las frecuencias de reloj se puede realizar aparte del proceso de diseño con *pragmas*. Además, destacamos el hecho de que el mínimo local en periodo $10ns$ da menor latencia que en el mínimo local en periodo $8.9ns$, y para un menor uso de recursos. Siendo que $10ns$ es el periodo por defecto para el dispositivo, esto sugiere que la herramienta y el hardware están diseñados para entregar implementaciones más costo-efectivas a esa frecuencia.

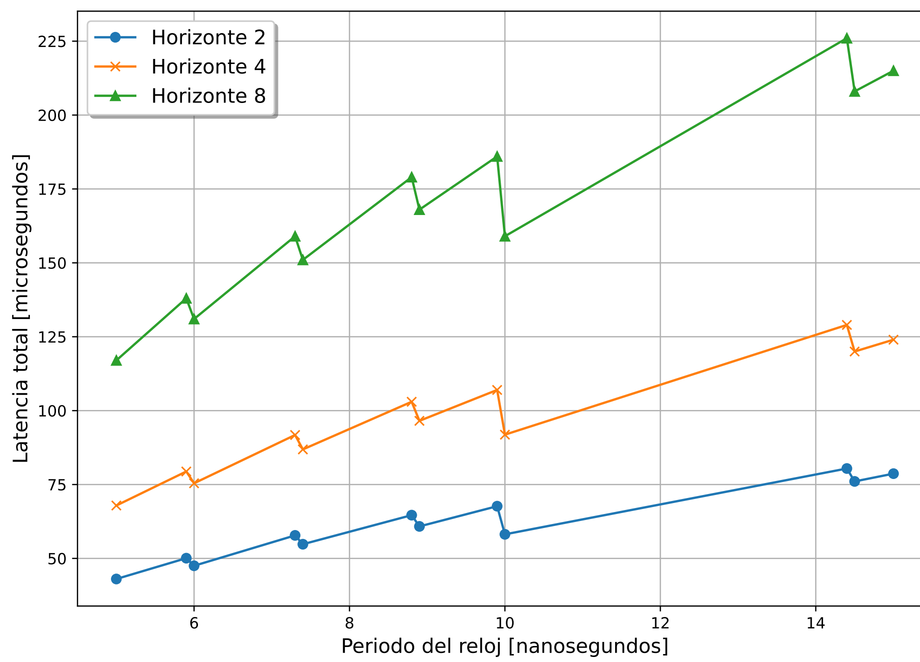
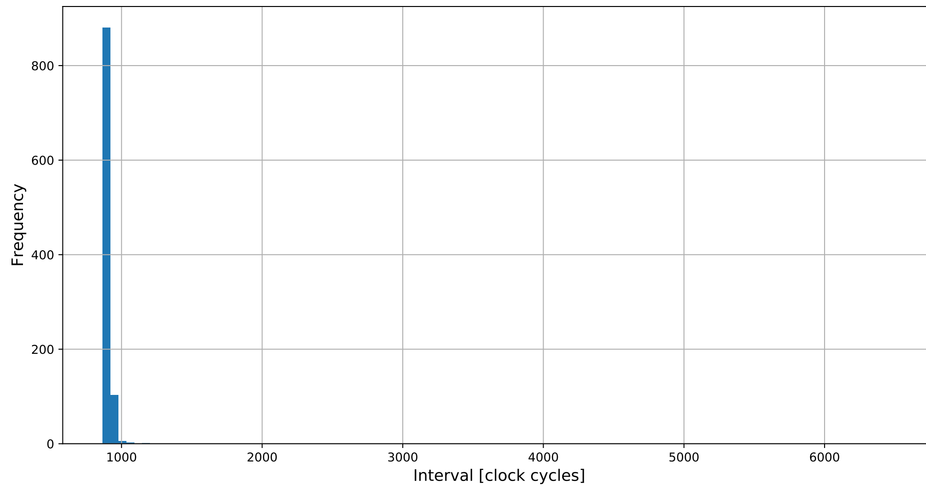


Figura B.1: Latencia de acelerador de motor DC contra periodo de reloj para problemas de distintos tamaños.

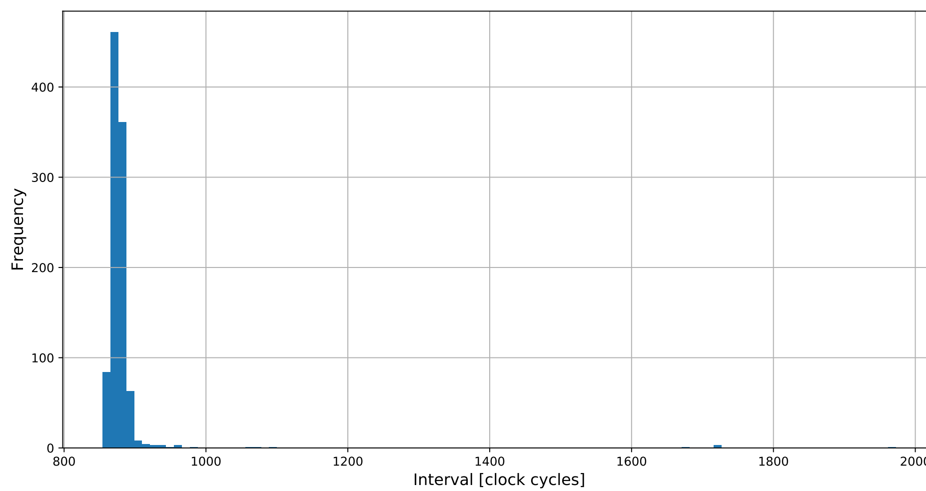
C | Latencia de la comunicación AXI con Pynq

Este anexo muestra el detalle de las mediciones de latencia de la comunicación por AXI desde una aplicación en Pynq en el dispositivo SoC utilizado.

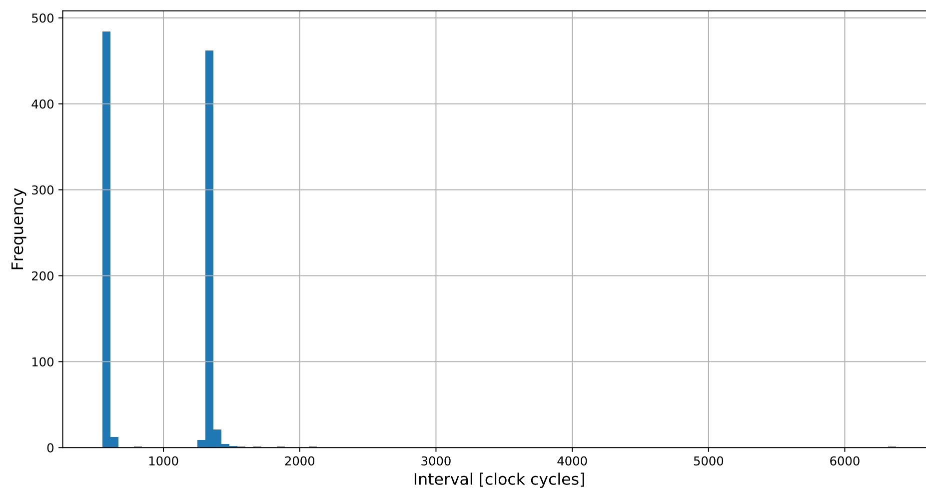
Como se explica en la Sección 5.2.1, nos interesa medir el intervalo entre transacciones AXI consecutivas para determinar el *overhead* que agrega la aplicación de software al acceder a la interfaz AXI y generar la transacción. Un análisis preliminar reveló que dicho intervalo depende del tipo de transacciones en cuestión, de forma que se distinguen cuatro tipos de intervalos: lectura-lectura, lectura-escritura, escritura-escritura y escritura-lectura. Luego, basta con medir los intervalos en tres escenarios: lecturas consecutivas, escrituras consecutivas y lecturas y escrituras alternadas. La Figura C.1 presentan histogramas de los tres escenarios. Se verifica que los cuatro tipos de intervalos tienen latencias promedio diferentes. En los cuatro casos la gran mayoría de muestras están bien agrupadas en torno a los valores promedio, sin embargo, en todos los casos hay mediciones *outliers* que tienen hasta seis veces la latencia promedio, de forma que la varianza es significativa.



(a) Escrituras consecutivas.



(b) Lecturas consecutivas.



(c) Escrituras y lecturas alternadas.

Figura C.1: Histogramas de intervalos entre transacciones AXI para tres escenarios.

D | Hardware-Oriented Code

Este anexo presenta nociones básicas de *hardware-oriented code* para trabajo futuro. Hemos visto que los *pragmas* demuestran ser útiles para explorar rápidamente el espacio de posibles implementaciones de un acelerador. Sin embargo, usar sólo *pragmas* puede resultar insuficiente para alcanzar el desempeño deseado con los recursos disponibles. El *hardware-oriented code* se refiere entonces a estrategias para escribir el código de alto nivel, que son funcionalmente equivalentes pero hacen a la herramienta de HLS inferir una arquitectura distinta para la implementación. En teoría, modificando el código podemos imitar el efecto de *pragmas*, como desenrollar un bucle o particionar un arreglo manualmente. Pero eso sería llegar a un exceso donde el esfuerzo es mayor y el código se vuelve difícil de manejar. En la práctica el *hardware-oriented code* sirve como un complemento al uso de *pragmas*.

Una forma simple de *hardware-oriented code* puede ser escribir diferentes versiones de las funciones de álgebra lineal para distintos tamaños de datos. En el presente trabajo utilizamos *templates* para generalizar la definición de las operaciones de álgebra lineal, de forma que se utiliza la misma definición con los mismos *pragmas* para implementar cada operación, independiente de su tamaño. Por ejemplo, cuando aplicamos desenrollamiento parcial a la suma de vectores lo hacemos con el mismo factor de desenrollamiento en todas las sumas independiente del tamaño de los vectores, pero se podría definir una función para la suma de tamaño M y otra con diferentes *pragmas* para la suma de tamaño N. Alternativamente, se podrían definir funciones que realicen combinaciones de operaciones. Por ejemplo, definir una función *add3()* que ejecute la suma de tres vectores equivale a fusionar dos sumas de dos vectores en un mismo bucle con *loop merge*, pero de una forma que es compatible con *unroll*. Otra posibilidad es definir simultáneamente ambas versiones de la MVM, *row-wise* y *column-wise*, e implementarlas con diferentes *pragmas* para optimizar distintas multiplicaciones del algoritmo según la relación de aspecto de la matriz correspondiente.

En la literatura se han presentado estrategias de *hardware-oriented code* para re-escribir operaciones comunes como la MVM. En [31] se propone el uso de *buffers* para enmascarar la dependencia de datos en el producto punto. Algo similar fue propuesto después en [15] para la MVM. Cabe destacar que la utilidad de las estrategias propuestas depende del tamaño de las operaciones. Al esconder las dependencias se consigue mejorar el II, pero también se incurre en un *overhead* que puede aumentar la latencia. Resulta necesario una comparación y un análisis más profundos de las condiciones en que estas estrategias resultan convenientes, en particular, en función del tamaño de los arreglos de datos.

Bibliografía

[1] J. Rawlings, D. Mayne, and M. Diehl, *Model Predictive Control: Theory, Computation, and Design.* Nob Hill Publishing, 2017. [Online]. Available: https://books.google.cl/books?id=MrJctAEACAAJ

[2] A. Diaz Dorado, "Efficient convex quadratic optimization solver for embedded mpc applications," 2018.

[3] K. M. Abughalieh and S. G. Alawneh, "A survey of parallel implementations for model predictive control," *IEEE Access*, vol. 7, pp. 34 348–34 360, 2019.

[4] A. Alessio and A. Bemporad, *A Survey on Explicit Model Predictive Control.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 345–369. [Online]. Available: https://doi.org/10.1007/978-3-642-01094-1_29

[5] I. McInerney, G. A. Constantinides, and E. C. Kerrigan, "A Survey of the Implementation of Linear Model Predictive Control on FPGAs," *IFAC-PapersOnLine*, vol. 51, no. 20, pp. 381–387, 2018, 6th IFAC Conference on Nonlinear Model Predictive Control NMPC 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896318327216

[6] M.-A. Boéchat, J. Liu, H. Peyrl, A. Zanarini, and T. Besselmann, "An architecture for solving quadratic programs with the fast gradient method on a Field Programmable Gate Array," in *21st Mediterranean Conference on Control and Automation*, 2013, pp. 1557–1562.

[7] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari, "Embedded Online Optimization for Model Predictive Control at Megahertz Rates," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, p. 3238–3251, 2014.

[8] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded Predictive Control on an FPGA using the Fast Gradient Method," in *2013 European Control Conference (ECC)*, 2013, pp. 3614–3620.

[9] H. Peyrl, A. Zanarini, T. Besselmann, J. Liu, and M.-A. Boéchat, "Parallel implementations of the fast gradient method for high-speed MPC," *Control Engineering Practice*, vol. 33, p. 22–34, 12 2014.

[10] S. Lucia, D. Navarro, O. Lucia, P. Zometa, and R. Findeisen, "Optimized FPGA Implementation of Model Predictive Control for Embedded Systems Using High Level Synthesis," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 137–145, 2018.

[11] Y. Li, S. E. Li, X. Jia, S. Zeng, and Y. Wang, “FPGA accelerated model predictive control for autonomous driving,” *Journal of Intelligent and Connected Vehicles*, vol. 5, no. 2, pp. 63–71, 2022.

[12] R. Lopez, “Control Predictivo Basado en Modelo para un inversor con un filtro LC a la salida en Recursos Energéticos Distribuidos (DERs),” Master’s thesis, Universidad Técnica Federico Santa María, 2021.

[13] J. D. Escárate, “Implementación en FPGA de control predictivo basado en modelos en un convertidor DC/AC conectado a un filtro LC en DERs,” Master’s thesis, Universidad Técnica Federico Santa María, Apr. 2023.

[14] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, “OSQP: an operator splitting solver for quadratic programs,” *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020. [Online]. Available: https://doi.org/10.1007/s12532-020-00179-2

[15] M. Jeong, M. Schoen, and J. Biela, “When fpgas meet admm with high-level synthesis (hls): A real-time implementation of long-horizon mpc for power electronic systems,” in *11th International Conference on Power Electronics and ECCE Asia*, 2023, pp. 1704–1711.

[16] A. Cortes, J. C. Aguero, C. Silva, and G. Carvajal, “Leveraging high level synthesis for the design of hardware accelerators for model predictive control,” in *2024 Argentine Conference on Electronics (CAE)*, 2024, pp. 16–21.

[17] T. V. Dang, K. Ling, and J. Maciejowski, “Embedded ADMM-based QP solver for MPC with polytopic constraints,” in *2015 European Control Conference (ECC)*, 2015, pp. 3446–3451.

[18] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[19] C. V. Rao, S. J. Wright, and J. B. Rawlings, “Application of Interior-Point Methods to Model Predictive Control,” *Journal of Optimization Theory and Applications*, vol. 99, no. 3, pp. 723–757, Dec 1998. [Online]. Available: https://doi.org/10.1023/A:1021711402723

[20] J. Liu, H. Peyrl, A. Burg, and G. 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)*, 2014, pp. 1–8.

[21] A. G. Wills, G. Knagge, and B. Ninness, “Fast Linear Model Predictive Control Via Custom Integrated Circuit Architecture,” *IEEE Transactions on Control Systems Technology*, vol. 20, no. 1, pp. 59–71, 2012.

[22] P. Zhang, J. Zambreno, and P. H. Jones, “An embedded scalable linear model predictive hardware-based controller using ADMM,” in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 176–183.

[23] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Found. Trends Mach. Learn.*, vol. 3, no. 1, p. 1–122, Jan. 2011.

[24] ARM, “AMBA AXI Protocol Specification.” [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest/

[25] AMD-Xilinx, “Zynq-7000 SoC Data Sheet: Overview (DS190),” 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview

[26] ——, “Vitis HLS User Guide (UG1399),” 2018. [Online]. Available: https://docs.xilinx.com/r/2022.2-English/ug1399-vitis-hls

[27] J. D. Escárate, “ADMM FPGA Implementation Repository in GitLab,” *https://gitlab.com/jdjotad/ipd-500_juan_escarate*, 2023. [Online]. Available: https://gitlab.com/jdjotad/ipd-500_juan_escarate

[28] J. Escárate, R. López, A. L. Cedeño, J. C. Agüero, C. Silva, and G. Carvajal, “FPGA Implementation of ADMM for Model Predictive Control in a DC/AC Converter,” in *2022 IEEE International Conference on Automation/XXV Congress of the Chilean Association of Automatic Control (ICA-ACCA)*, 2022, pp. 1–6.

[29] AMD-Xilinx, “Introduction to AXI,” 2018. [Online]. Available: https://support.xilinx.com/s/article/1053914?language=en_US

[30] AMD-Xilinx, “Documentación del sistema Pynq.” [Online]. Available: https://pynq.readthedocs.io/

[31] H. A. Shukla, B. Khusainov, E. C. Kerrigan, and C. N. Jones, “Software and Hardware Code Generation for Predictive Control Using Splitting Methods.” *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 14 386–14 391, 2017, 20th IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article