Plataforma para la programación automática del sistema hardware reconfigurable de una máquina Zynq

Eduard Gil Blasco

Facultat Informàtica de Barcelona

Universitat Politècnica de Catalunya

Memoria del Proyecto Final de Carrera

Ingeniería Informática Superior

Enero 2014

DATOS DEL PROYECTO

Título del proyecto:	Plataforma para la programación automática del sistema hardware reconfigurable de una máquina Zynq
Nombre del estudiante:	Eduard Gil Blasco
Titulación:	Ingeniería Informática superior
Créditos:	37,5
Director:	Daniel Jiménez González
Departamento:	Arquitectura de Computadores

MIEMBROS DEL TRIBUNAL *(nombre y firma)*

Presidente:	Carlos Álvarez Martínez
Vocal:	Horacio Rodríguez Hontoria
Secretario:	Daniel Jiménez González

CUALIFICACIÓN

Cualificación numérica:

Cualificación descriptiva:

Fecha:

Abstracto

Las máquinas *Zynq* son una familia de *SoC* que integran una FPGA. La intención de este proyecto es automatizar todo el proceso que se requiere para obtener el *bitstream* a partir de una aplicación escrita en C/C++ con directivas OmpSs. Además con la integración de otra aplicación realizada por el Barcelona Supercomputing Center se consigue generar el binario para los ARM del *SoC* de manera automática. De esta manera, esta plataforma permite acelerar cualquier programa escrito en C/C++ tan solo añadiendo 2 directivas OmpSs al código.

A todo aquel que me ha dedicado alguna vez parte de su tiempo, pues es algo que nunca va a recuperar.

Agradecimientos

Quisiera reconocer a Dani por su labor y todos esos momentos de frustración y alegría que hemos compartido en este proyecto. A Antonio por su tiempo en probar toda esa cantidad de *bitstreams* generados, que no son pocos. A todos los demás porque a pesar de no tener conocimientos en el campo se han interesado por el proyecto. Y por último reconocer a este proyecto todo lo que me ha aportado.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Glosario

API *Application Programming Interface*. Especifica cómo se debe interactuar con un componente software en concreto.

ARM Familia de procesadores basados en la arquitectura *RISC* (conjunto de instrucciones reducido) desarrollados por *ARM Holdings*.

Bitstream En el contexto de este proyecto se refiere al binario que configura el hardware reconfigurable.

Cast Conversión de tipo.

Core En el contexto de este proyecto hace referencia al núcleo a acelerar descrito en VHDL.

CtoVHDL Conversión de código C a VHDL.

DCVS *Distributed Concurrent Versions System*. Permite a los desarrolladores trabajar localmente de manera distribuida en proyectos software. Está basado en los sistemas de control de versiones concurrentes (CVS).

Elf En el contexto de este proyecto se refiere al binario que se ejecuta en los ARM.

FPGA *Field-Programmable Gate Array*. Es un circuito integrado diseñado para ser configurado posteriormente tantas veces como se requiera (reconfigurable). Generalmente la configuración se especifica utilizando un HDL.

Gather Empaquetado de datos de varios a uno. Generalmente varios procesos envían datos a un proceso.

Hardware reconfigurable FPGA.

HDL *Hardware Description Language*. Lenguaje de programación especializado para programar la estructura, diseño y operación de circuitos electrónicos mayormente digitales.

HLS *High-Level Synthesis*. En el contexto del proyecto se refiere a las directivas HLS que describen el comportamiento que se desea implementar. En nuestro caso por ejemplo se utiliza para indicar niveles de optimización entre otras.

IDE *Integrated Development Environment*. Aplicación que provee de las herramientas necesarias para el desarrollo de software.

JSON *JavaScript Object Notation*. Es un formato estándar abierto que utiliza texto legible para transmitir objetos de datos que consisten en parejas de atributo-valor.

Kernel En el contexto del proyecto se refiere al programa que incluye la parte a acelerar y además realiza el empaquetado y desempaquetado de datos entre el ARM y la FPGA.

MHS *Microprocessor Hardware Specification*. Este archivo es utilizado por la herramienta de generación del *bitstream* y en él se especifica la configuración del microprocesador, los elementos que contendrá la FPGA y su interconexión.

Plugin Componente software que añade una funcionalidad específica a una aplicación existente.

Pragma Directiva.

Scatter Desempaquetado de datos de uno a varios. Generalmente un proceso envía datos a varios procesos.

SDK *Software Development Kit*. Conjunto de herramientas de desarrollo que permiten la creación de aplicaciones para un cierto *framework*, paquete software, plataforma hardware o similar.

SMP *Symmetric Multiprocessor System*. Es un sistema multiprocesador con una memoria centralizada compartida.

SoC *System On a Chip*. Circuito integrado que integra todos los componentes de un ordenador o sistema electrónico en un chip.

Standalone En el contexto de este proyecto se utiliza en el ámbito de los programas. Un programa *standalone* no requiere de sistema operativo para ser ejecutado.

Stream En el contexto de este proyecto se refiere a una estructura de datos la cual contiene diferentes datos pero no están contiguos en memoria. La lectura de datos es destructiva. Los datos se escriben después del último dato escrito.

Tcl *Tool Command Language*. En el contexto de este proyecto los archivos *tcl* contienen instrucciones específicas para un programa en concreto.

Template Plantilla.

Toolchain Conjunto de herramientas programadas para la creación de un producto donde normalmente la salida de una de las herramientas es la entrada de otra de las herramientas formando así una cadena.

VHDL *VHSIC Hardware Description Language*. HDL utilizado en la automatización de diseños electrónicos para describir sistemas digitales como FPGAs y circuitos integrados.

Xilinx Xilinx Inc. es una compañía americana que se dedica principalmente a los dispositivos lógicos programables.

GLOSARIO

1

Introducción

El hardware reconfigurable es una de las opciones de supercomputación de bajo consumo. Sin embargo, programar estos dispositivos reconfigurables no es una tarea sencilla y requiere mucho esfuerzo en investigación y desarrollo para hacerlo realmente productivo.

Este proyecto se centra en explotar el paralelismo de aplicaciones científicas en arquitecturas heterogéneas que dispongan de hardware reconfigurable. En particular se centra en las *Zynq All-Programmable SoC* y en las aplicaciones desarrolladas utilizando OmpSs, un conocido modelo de programación paralela, extensión del estándar OpenMP[1].

El propósito es generar de manera automática un *bitstream* para el hardware reconfigurable de las *Zynq All-Programmable SoC* a partir de un código con directivas OmpSs y un *kernel* específico para ese código que se encarga de la comunicación de datos entre el ARM y la FPGA.

El desarrollo de este proyecto se lleva acabo dentro de otro proyecto más grande realizado por el Barcelona Supercomputing Center (BSC). El proyecto del BSC tiene como objetivo poder generar automáticamente el ejecutable de los ARM del *SoC*, el binario *elf*, como también el binario para configurar la FPGA, el bitstream. Nuestro proyecto se encarga de la generación del *bitstream*.

1.1 Motivación

El uso de hardware reconfigurable como aceleradores en sistemas *multicores* y SMP añade un nivel de complejidad adicional. Se deben tener en cuenta aspectos como la descarga de datos y la interoperabilidad con los modelos de programación paralela para memoria compartida, como OpenMP.

Actualmente, las herramientas disponibles permiten generar un ejecutable que utiliza el hardware reconfigurable de las *Zynq All-Programmable SoC*. Este *SoC* es una plataforma con dos procesadores *Core-9 ARM* con una FPGA integrada. Sin embargo, la generación del hardware necesita ser realizada a mano, con el tiempo y los errores de programación que ello puede implicar.

En este contexto, a día de hoy el usuario está provisto de todas las herramientas para aprovechar las ventajas del hardware reconfigurable. El problema está en la cantidad de tiempo que se necesita invertir en todo el proceso y los conocimientos que se requieren.

Un gran avance para usuarios que quieran utilizar dispositivos reconfigurables sería abstraerlos de estos problemas para que dediquen el tiempo y esfuerzo en crear y mejorar su aplicación. De esta manera otros usuarios también podrían interesarse ya que obtendrían las ventajas del hardware reconfigurable sin necesidad de tener conocimientos sobre ello.

1.2 Objetivos

Se quiere conseguir generar de manera automática el *bitstream* para ejecutar aplicaciones en el hardware reconfigurable de las *Zynq All-Programable SoC*. Las aplicaciones están escritas en C/C++ y utilizan directivas OmpSs que indican qué partes se deben acelerar y cómo.

Para conseguir este propósito por un lado se genera un *kernel* específico de la parte del código a acelerar que además de la parte del código a acelerar incluye la descarga de datos entre el ARM y la FPGA. Por otro lado, a partir de este *kernel* se genera un *bitstream* que configura el hardware reconfigurable teniendo en cuenta las necesidades de la aplicación.

Así pues, el proyecto tiene dos objetivos
1) Automatización de la generación del hardware
2) Generación automática del *kernel* a partir de un código con directivas OmpSs

1.2.1 Automatización de la generación del hardware

El primer objetivo es el de generar de manera automática un *bitstream* para las *Zynq All-Programmable SoC* a partir de un *kernel* con directivas HLS. Para ello tenemos el siguiente desglose en objetivos intermedios.

Generación automática del *core*

El *kernel* en C/C++ es transformado en código HDL para que pueda ser procesado por las herramientas de síntesis.

Generación automática de la configuración hardware

Se configura el comportamiento y la interconexión de los diferentes componentes del *SoC* según las necesidades de la aplicación. Entradas y salidas, interrupciones, etc. Se añaden los elementos que se sintetizarán en la FPGA.

Generación automática del *bitstream*

A partir de la configuración realizada y con el *core* acelerador en código HDL se realiza la síntesis, la implementación y finalmente la generación del *bitstream*.

Soporte para múltiples *cores* aceleradores homogéneos

Se añadirá soporte para tener más de una instancia aceleradora, *core*, del mismo tipo en una misma FPGA. Ambos cores implementan el mismo código.

Soporte para *cores* aceleradores heterogéneos

Se añadirá soporte para tener más de un *core* diferente dentro de una misma FPGA. Los cores implementan diferentes aplicaciones.

1.2.2 Automatización de la generación del *kernel*

El segundo objetivo consiste en generar un código con directivas HLS, *kernel*, a partir de un código con directivas OmpSs. El código generado contendrá la parte a acelerar y la gestión de la descarga de datos. Por último, se integra la generación del *kernel* con la generación del hardware y finalmente con el resto del proyecto del BSC. Para ello tenemos el siguiente desglose en objetivos intermedios.

Generación del *kernel*

A partir del código con directivas OmpSs, se extrae la información necesaria para crear un *kernel* con directivas HLS. El *kernel* incluye la parte a acelerar y la gestión de descarga de datos necesaria.

Integración con la generación hardware

Para automatizar todo el proceso por completo se integra la generación del hardware con la generación del *kernel*.

Integración con el proyecto existente en BSC

La plataforma de automatización se integra con el resto del proyecto del BSC para tener toda la generación con tan solo una llamada a la aplicación. Con esta integración somos capaces de generar todo lo necesario para acelerar aplicaciones en los *SoCs* especificados.

2

Metodología

En la figura 2.1 se muestra la metodología seguida en alto nivel la cuál consta de los siguientes pasos:

1. Acercamiento, sección 2.1.
2. Generación Hardware, sección 2.2.1.
3. Generación del *kernel,* sección 2.2.2.
4. Integración, sección 2.2.3.
5. Reporte.

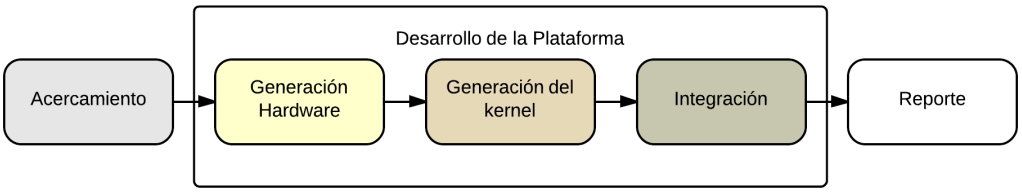

Figura 2.1: Sumario de la metodología -

En la figura 2.2 se muestran las actividades de cada una de las partes. Según el color de la actividad se puede ver a que parte de las mostradas en la figura 2.1 pertenece. A continuación se explica en detalle.

Dado que el hardware reconfigurable es un campo en el cual carecíamos de experiencia ha sido preciso realizar un acercamiento en este tema. En él hemos

2. METODOLOGÍA

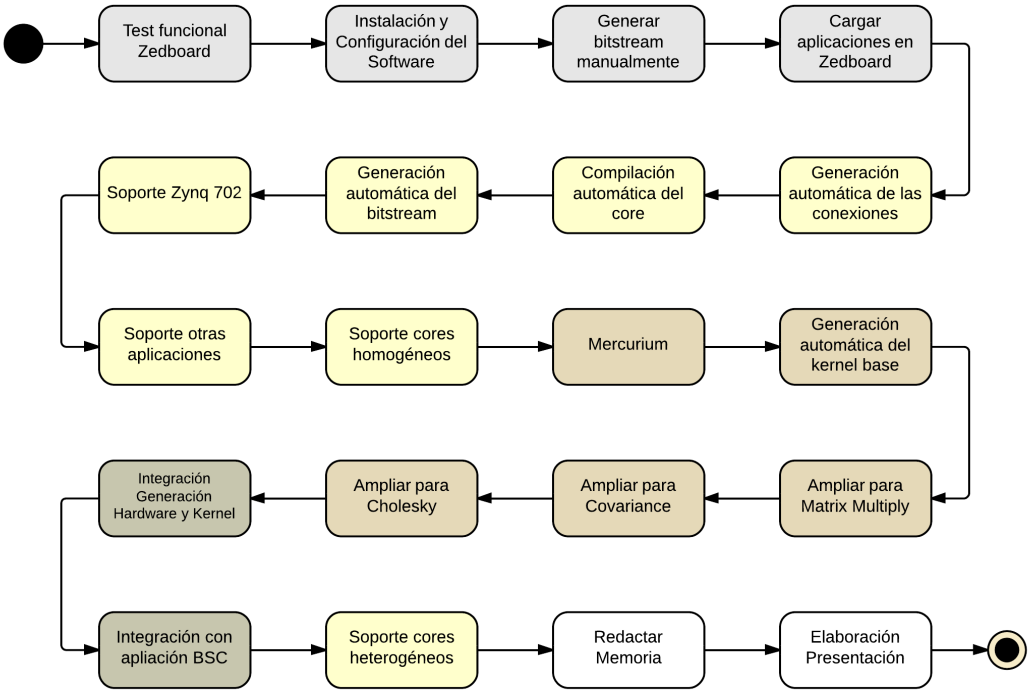

Figura 2.2: Metodología en detalle -

obtenido los conocimientos necesarios para poder desarrollar la plataforma. La primera parte que hemos desarrollado corresponde a la generación del hardware. Cuando conseguimos generar de manera automática el hardware hemos seguido con la segunda parte, la generación del *kernel*. Para finalizar la plataforma hemos integrado ambas partes. Por último hemos realizado la integración con el resto del proyecto del BSC para obtener un *toolchain* unificado y sencillo de utilizar. Al acabar hemos añadido el soporte para *cores* heterogéneos en la generación hardware y de esta manera hemos cumplido todos los objetivos. Redactar la memoria y elaborar la presentación ha sido la tarea que ha culminado el proyecto.

2.1 Acercamiento

1. Test funcional del *SoC Zedboard*
2. Instalación y Configuración del Software
3. Generar el *bitstream* manualmente
4. Cargar aplicaciones en la *Zedboard*

En el acercamiento nos hemos familiarizado con los diferentes elementos y procedimientos que utilizaremos sobretodo para la generación hardware. Hemos realizado diferentes pruebas en la *Zedboard*: de conexión, configuración de cortocircuitos, etc.[2, 3] Posteriormente, se ha instalado la Suite de Xilinx y se configuran los *drivers* para cargar los *bitstreams* en el *SoC*.[4, 5] Siguiendo diferentes manuales hemos sido capaces de generar un *bitstream* de manera manual y cargar dos aplicaciones que hacen uso de ese *bitstream* en la *Zedboard*.[6]

2.2 Desarrollo Plataforma

El desarrollo de la plataforma se ha realizado en 3 partes, la primera consiste en la Generación Hardware, la segunda en la Generación Kernel y la tercera en la Integración.

2.2.1 Generación Hardware

1. Generación automática de las conexiones
2. Compilación automática del *core*
3. Generación automática del *bitstream*
4. Añadir soporte *ZC702*
5. Añadir soporte a otras aplicaciones
6. Añadir soporte *cores* homogéneos
7. (Secundario) Añadir soporte *cores* heterogéneos

Primeramente hemos empezado con la automatización de la generación de las conexiones ya que es el proceso más complejo de la generación manual del *bitstream*. Posteriormente, hemos automatizado la sintetización de C a VHDL del *core*.[7] Para finalizar la automatización de la generación hardware se ha añadido la generación del *bitstream*.[8, 9] Cuando hemos verificado el correcto funcionamiento del *bitstream* en la *Zedboard*, se ha añadido el soporte para el *SoC ZC702*. Después de verificar el funcionamiento en la *ZC702* hemos añadido los cambios necesarios para soportar otros tipos de aplicaciones con diferentes requisitos. Para añadir un mayor impacto en la aceleración hemos añadido la posibilidad de tener tantas copias del *core* como acepte el *SoC*, *cores* homogéneos. El soporte para tener múltiples *cores* heterogéneos era secundario y no se ha realizado hasta la finalización del desarrollo de la plataforma, una vez acabada la integración.

2.2.2 Generación Kernel

1. Mercurium
2. Generación automática del *kernel* base
3. Ampliar para Matrix Multiply
4. Ampliar para Covariance
5. Ampliar para Cholesky

La generación del *kernel* consiste en la transformación de un código en C/C++ con directivas OmpSs a un código en C/C++ con directivas HLS. Para ello inicialmente se ha valorado la posibilidad de utilizar la herramienta Mercurium dado que es un compilador fuente a fuente muy versátil. Se han creado algunas fases de prueba del compilador para entender su funcionamiento. Después de analizarlo hemos decidido que para facilitar la integración con la Generación Hardware y debido a que necesitaríamos otro acercamiento para Mercurium se realizaría la generación del *kernel* mediante una aplicación que desarrollaríamos nosotros mismos desde zero.

El *kernel* generado deberá ser transformado por la herramienta de sintetización de C a VHDL. Esta herramienta no permite utilizar todas las operaciones del estándar C++ y por lo tanto hace que sea complejo poder adaptar la aplicación a cualquier tipo de problema (código). La aplicación ha sido diseñada para que sea escalable de una manera simple. Inicialmente desarrollamos la aplicación para que construya un *kernel* genérico. Posteriormente, se ha ampliado para otros tipos de problemas lo cual ha añadido versatilidad a la aplicación. Como se puede comprobar en la sección 5.2, añadir soporte para otros problemas resulta sencillo y no requiere modificar el código de la aplicación.

2.2.3 Integración

1. Integración de la Generación Hardware y Generación Kernel

2. Integración con el resto del proyecto del BSC

Para la integración de la Generación Hardware y la Generación Kernel se ha creado una aplicación que une la salida de la Generacion Kernel con la entrada de la Generacion Hardware. Primero se generan los *kernels* y posteriormente se utilizan para generar el *bitstream* correspondiente. Para la integración con el resto del proyecto del BSC se ha creado otra aplicación que añade una interfaz de uso muy simple para unificar todo el *toolchain*.

2.2.4 Herramientas

Las herramientas que se mencionan a continuación no están relacionadas directamente con el hardware reconfigurable. Las herramientas que sí lo están se describen en la sección 3.

Eclipse Se ha utilizado como IDE principal.[10]

PyDev *Plugin* para eclipse para la programación en Python.[11]

Git & Bitbucket Se ha utilizado Git como DCVS y Bitbucket como *host* de nuestro repositorio.[12, 13]

2.2.5 Lenguajes de programación

Estos son los lenguajes de programación que se han utilizado además de *shell script*.

Python Para el desarrollo de las herramientas se ha utilizado Python 2.7.[14, 15]

C/C++ Para el desarrollo de la la fase de Mercurium se ha utilizado C++ y los *kernels* generados están escritos en C/C++.[16]

2.2.6 Fuentes consultadas

Entre las más destacadas tenemos:

- IEE Xplore Digital Library[17]
- ACM Digital Library[18]
- Google Scholar[19]

3

Entorno de trabajo

A continuación se describen las infraestructuras hardware y software que se han utilizado:

3.1 OmpSs

OmpSs[20] es un modelo de programación desarrollado por el BSC con el objetivo de extender OpenMP con nuevas directivas para el soporte de paralelismo asíncrono y sistemas heterogéneos. También, como se advierte en su página web, puede ser utilizado para añadir nuevas directivas extendiendo otras APIs para aceleradores como CUDA[21] o OpenCL[22].

Las directivas OmpSs que se utilizan para generar el *kernel* son las relacionadas con dependencia de datos y las que especifican el dispositivo destino de la ejecución de los *kernels*. En la sección 5.2.1 se muestran en detalle.

La implementación que existe de este modelo de programación está basada en Mercurium como *front-end compiler*.

3.2 Mercurium

Mercurium es una infraestructura de compilación fuente-a-fuente. Actualmente soporta C y C++. Principalmente se utiliza para la implementación de OpenMP en el entorno de Nanos. No obstante, debido al carácter extensible que tiene, también se

ha utilizado para implementar otros modelos de programación o transformaciones de compilación.[23, 24]

El programa usa una arquitectura de *plugin* escritos en C++ los cuales representan las fases del compilador. Según la configuración, los *plugin* son cargados dinámicamente por el compilador.

3.3 Nanos

Nanos se utiliza como librería en tiempo de ejecución (*runtime*) en el código de los ARM. Se trata de un *runtime* diseñado para entornos donde interviene el paralelismo. Principalmente se utiliza para la sincronización de entornos paralelos, como por ejemplo la sincronización entre *threads*, y la gestión de los mismos *threads*. De esta manera consigue complementarse con OmpSs.

Al igual que Mercurium es extensible mediante el uso de *plugin*.

3.4 Zynq All-Programmable Soc

Los *Zynq All-Programmable SoC* son una familia de *System on a chip (SoC)* desarrollada por Xilinx. Destaca por el rendimiento que tiene en relación al bajo consumo que hace. Esta arquitectura tiene dos *cores* ARM-A9 además del chip de FPGA de la familia *Zynq*.[25]

En la figura 3.1 se muestra la arquitectura de los *Zynq All Programmable SoC*.

3.4.1 ZedBoard

Uno de los *SoC* utilizados ha sido la *Zedboard*. Se trata de un *SoC* asequible de bajo precio que integra un chip FPGA *Zynq-702* con el cual se han llevado a cabo las primeras pruebas.

3.4.2 ZC702

La *ZC702* es el *SoC* objetivo del proyecto para el cual se generan los *bitstreams*. A pesar de que integra el mismo chip FPGA que la *ZedBoard*, la *ZC702* dispone de muchos más recursos como memoria, frecuencia de procesador lógico, etc.

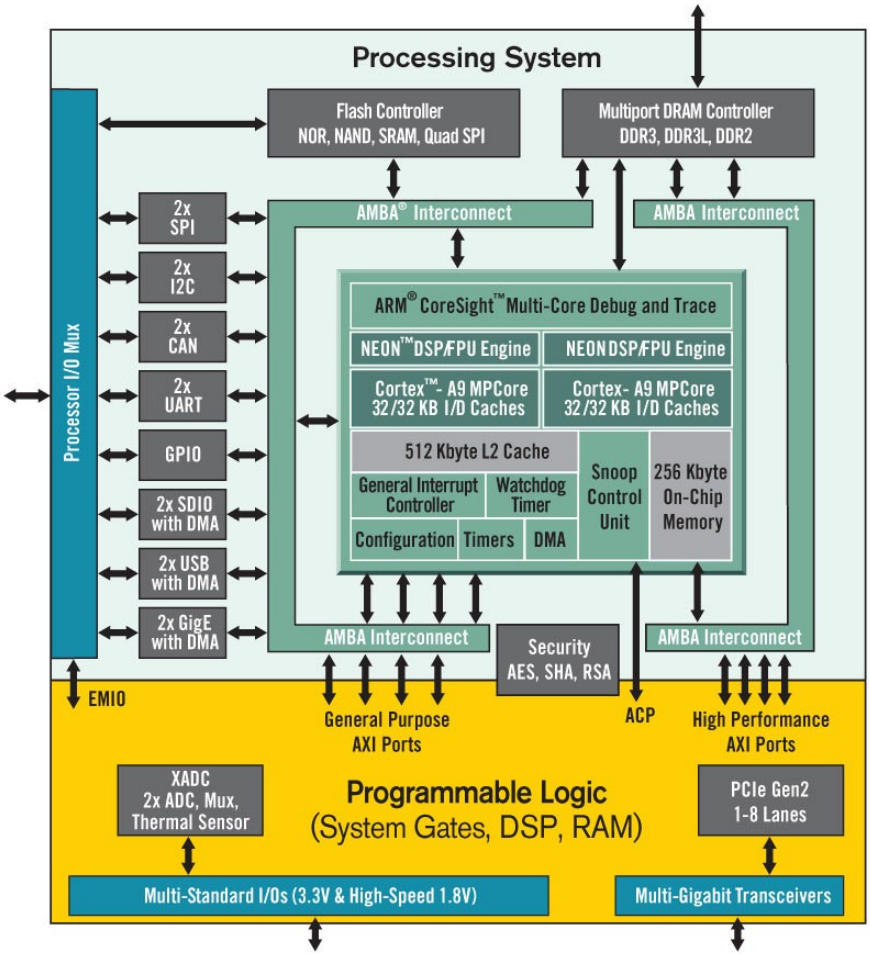

Figura 3.1: Zynq 7000 overview - Arquitectura Zynq All Programmable SoC , Fuente: Página oficial de Xilinx.

3.5 Vivado & ISE Suite

Vivado y ISE Suite pertenecen a la suite *Vivado System Design Edition*. Se trata de software propietario de Xilinx. Con el entorno que proveen este conjunto de herramientas el usuario es capaz de desarrollar todo tipo de aplicaciones para las FPGA. Integran todo el software necesario para el uso del hardware reconfigurable en sus productos.

En el proyecto se utilizan las siguientes herramientas de la suite:

VivadoHLS Para la sintetización del *kernel* en C/C++ a VHDL.

PlanAhead Para la síntesis, implementación y generación del *bitstream*.

XPS Para la configuración del *bitstream*. Crea un archivo MHS.

EDK Para generar aplicaciones *standalone* y cargar el *bitstream* en el *SoC*.

VivadoHLS pertenece a Vivado mientras que PlanAhead, XPS y EDK pertenecen a ISE Suite.

4

Diseño de la plataforma y Uso

4.1 Esquema General

Ompss2fpga integra GrizzlyH fpgacxx como se muestra en la figura 4.1. fpgacxx es un alias que llama a Mercurium con una configuración específica, y que forma parte del proyecto del BSC donde se integra nuestro proyecto.

A partir del código C/C++ con directivas OmpSs y mediante el uso de de la plataforma creada, ompss2fpga, conseguimos generar el binario del ARM utilizando fpgacxx y el *bitstream* de la FPGA utilizando GrizzlyHG.

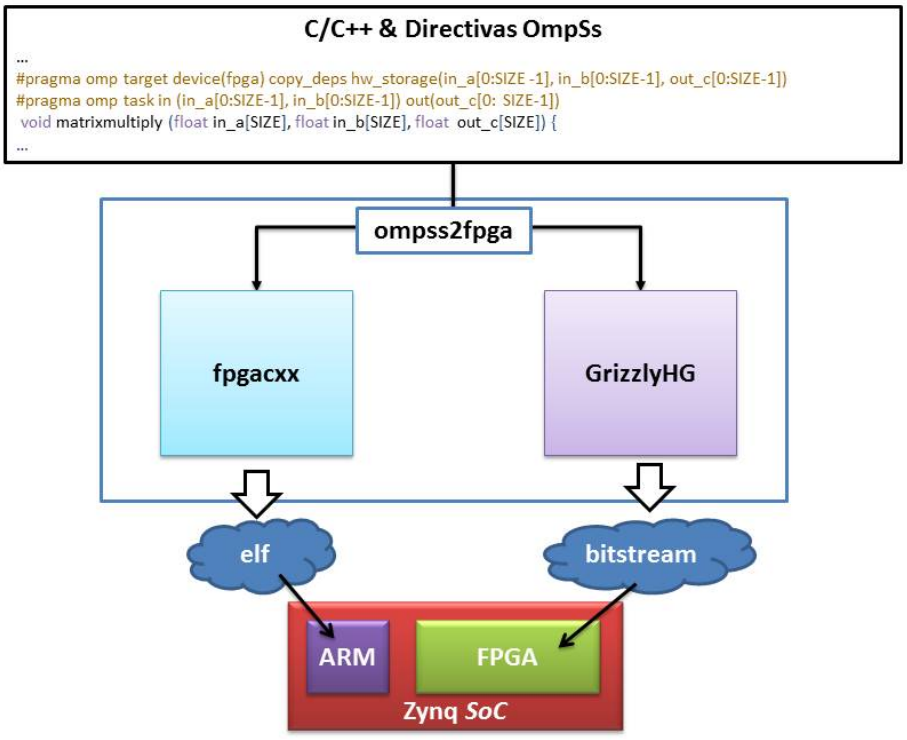

Figura 4.1: Ompss2fpga - Diagrama Funcional

4.1.1 *Syntax* de la línea de comando

Ompss2fpga se ha diseñado para que se pueda ejecutar via línea de comando. La sintaxis de la línea de comando se muestra a continuación. No debe confundirse con la mostrada en el apartado 4.3.4 que corresponde a la sintaxis de GrizzlyHG.

```
usage: ompss2fpga [-h] [--wrapper] [--fpga-options OPTIONS]
                  [--grizzly-options OPTIONS]
                  [ompss_code [ompss_code ...]]

positional arguments:
  ompss_code            Path to input code file with pragmas

optional arguments:
  -h, --help            show this help message and exit
  --wrapper             If present grizzly will execute only kernel
      generation
  --ompss_code          Path to input code file with pragmas
  --fpga-options OPTIONS
                        Specify options for fpga. Between brackets.
  --grizzly-options OPTIONS
                        Specify options for grizzly. Between brackets. If
                            not present takes defaults from grizzly/etc/
                            defaults.cfg. For more info type '
                            grizzlyLauncher -h'
```

Lo siguientes ejemplos muestran como se utiliza esta sintaxis.

Generación Kernel & Hardware para *cores* heterogéneos

A continuación se muestra la llamada para la generación del *bitstream* con *cores* heterogéneos a partir de los códigos MxM.cpp y covariance.cpp.

```
ompss2fpga codigos/MxM.cpp codigos/covariance.cpp
```

Generación Kernel & Hardware para *cores* heterogéneos con parámetros

A continuación se muestra la llamada para la generación del bitstream con *cores* heterogéneos a partir de los códigos ompss_mmult_core.cpp, y covariance_core.cpp cambiando el directorio donde se guardan los kernels generados a "/tmp/kernels".

```
ompss2fpga  codigos/ompss_mmult_core.cpp codigos/covariance_core.cpp --fpga-
    options "--no-optimize"  --grizzly-options "-o /tmp/kernels"
```

Otro ejemplo con *cores* heterogéneos, 2 *cores* a partir del código matrix_mult.cpp y 1 *core* a partir del código cholesky.cpp.

```
ompss2fpga.py  matrix_mult.cpp cholesky.cpp --grizzly-options "-coreN 2 -
    coreN2 1"
```

4.2 Generación del binario ARM: fpgaccxx

Esta parte está incluida en el proyecto del BSC pero el desarrollo de la misma no se incluye en nuestro proyecto, lo realiza el BSC.

La función de fpgacxx es el generar el binario elf, es decir, el binario que se ejecuta en el ARM.

4.3 Generación del bistream: GrizzlyHG

4.3.1 Esquema general

GrizzlyHG4.2 es la aplicación que integra la Generación Kernel con la Generación Hardware.

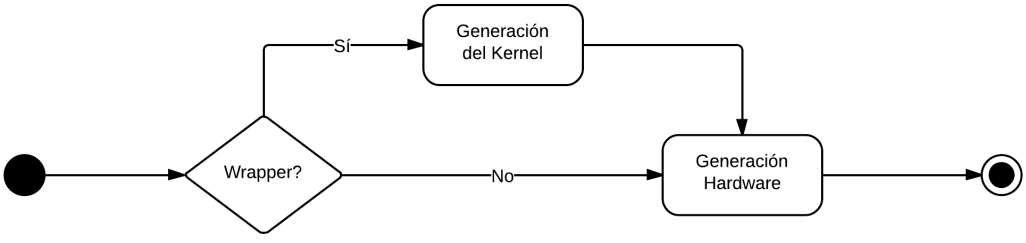

Figura 4.2: GrizzlyHG - Diagrama Funcional

Como se puede observar en la figura4.2, la Generación Kernel es opcional. GrizzlyHG ofrece la posibilidad de generar únicamente el *bitstream* a partir de un *kernel* ya existente.

Cuando se realiza la Generación Kernel, el *kernel* que genera forma parte de la entrada de la Generación Hardware.

4.3.2 Funcionalidad Interna

Para la generación del *kernel* se sigue el proceso de la figura 4.3. La entrada de la aplicación de Generación Kernel son códigos en C o C++ con directivas OmpSs y para cada uno de ellos se sigue el proceso mostrado.

Inicialmente, Finder agrupa en diferentes *Task* todos los *pragmas* OmpSs y las funciones a acelerar. Cada *Task* está compuesta por una función a acelerar y los *pragmas* OmpSs relacionados.

En un mismo código podemos tener tantas *Task* como se quieran. El resto de la aplicación itera sobre la agrupación de *Task* creada por Finder. En cada iteración se utiliza una única *Task* y solo se itera una vez sobre la misma. Para cada una se le aplican tres fases diferentes:

La primera fase consiste en extraer toda la información de los *pragmas* y de la cabecera de la función. Se obtienen los tipos e identificadores de las variables, tamaños de dimensiones, si son de entrada o salida, etc.

La segunda fase consiste en escribir el código del *kernel* resultante. Cada *kernel* generado consiste en el *Scatter* de los datos utilizados por la función del código original, la llamada a dicha función y el *Gather* para retornar los resultados de la función ejecutada.

La ultima fase añade un archivo *tcl* que indica los parámetros de compilación para ese *kernel*. Este archivo sera utilizado en la Generación Hardware para convertir el código C/C++ del *kernel* en código HDL.

En la figura 4.4 se muestra el proceso para la generación del hardware. El proceso es el mismo independientemente de si partimos de un *kernel* ya generado o se genera utilizando la Generación Kernel.

La Generación Hardware empieza con la transformación de los *kernels* a código HDL obteniendo los *cores*.Posteriormente se crea un proyecto de PlanAhead partiendo de uno de los *templates* elegidos. Los *cores* generados se añaden al proyecto configurando la interconexión según el *template* de conexión elegido. Para especificar la interconexión se utiliza el archivo de configuración MHS del proyecto.

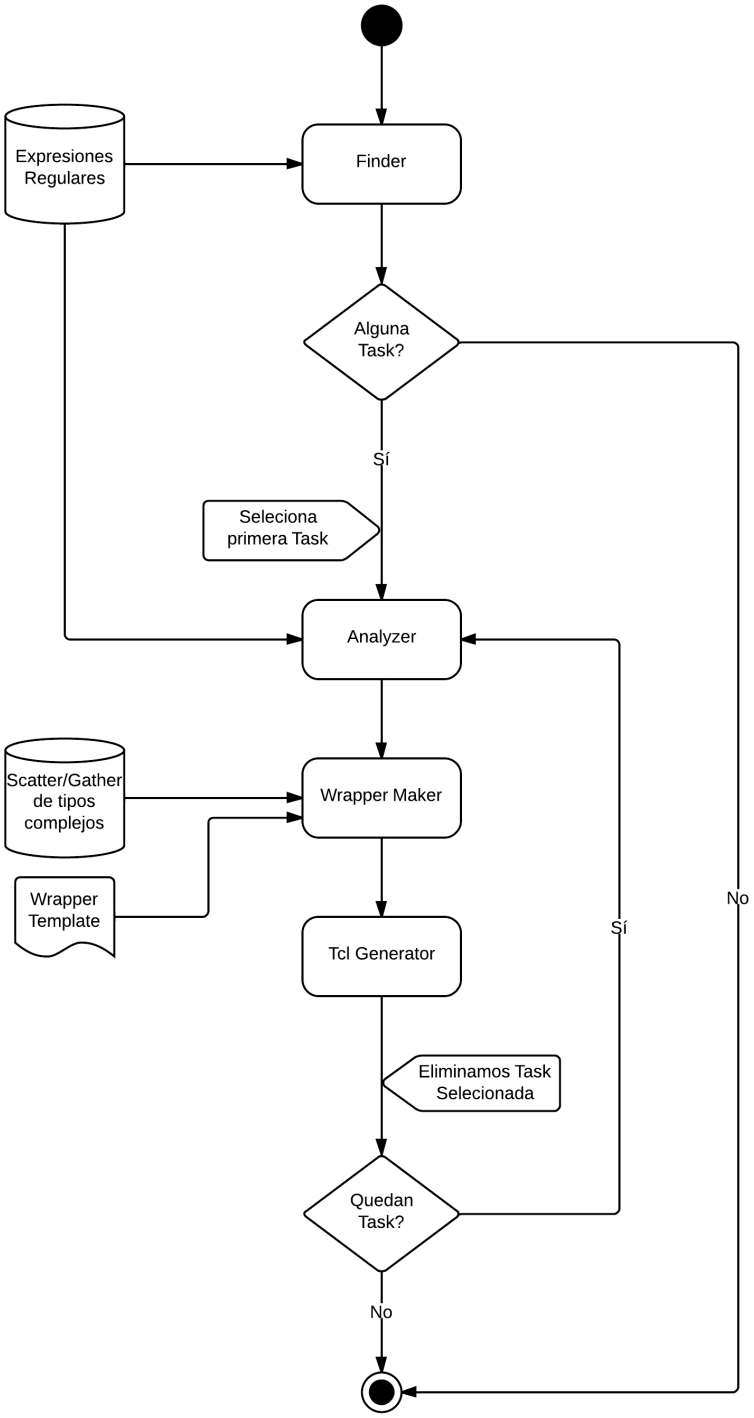

Figura 4.3: Generación Kernel - Diagrama Funcional

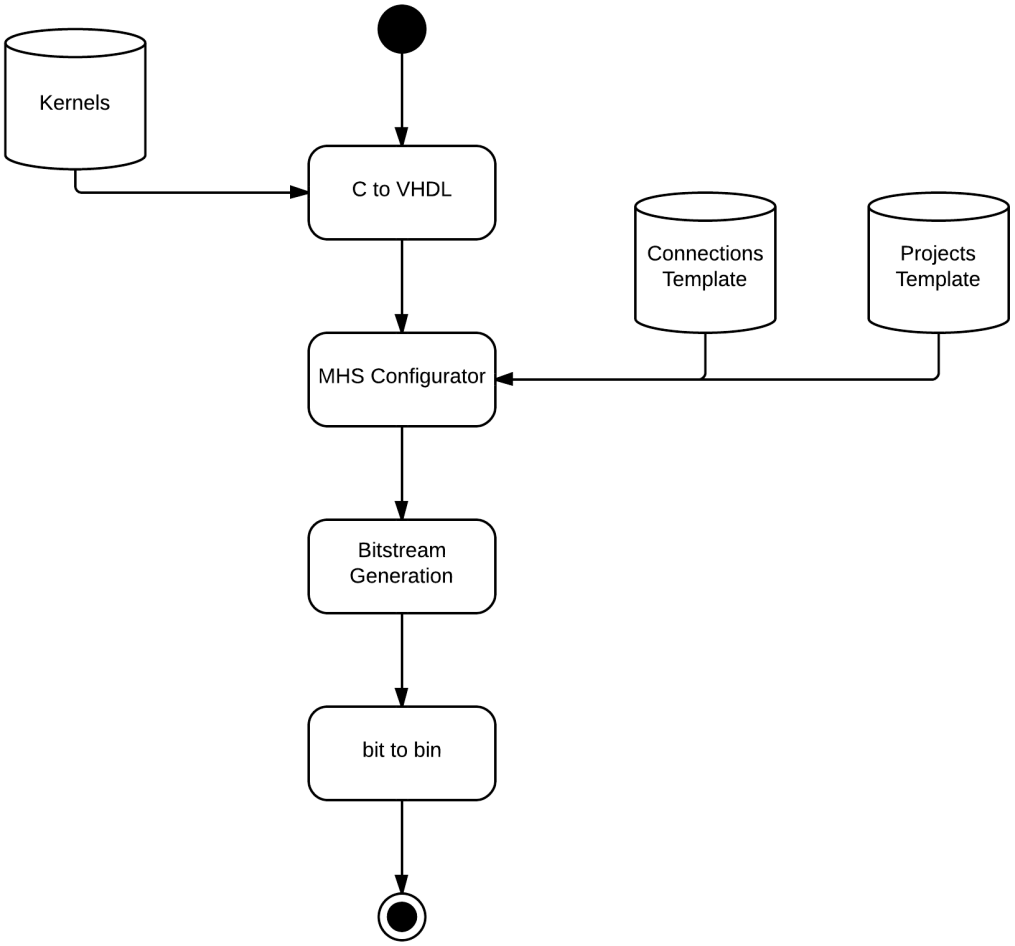

Figura 4.4: Generación Hardware - Diagrama Funcional

Una vez realizada la configuración de conexiones se procede a la síntesis e implementación del proyecto para acabar con la generación del *bitstream*.

Como último paso se convierte el *bitstream* generado a binario.

La implementación de GrizzlyHG se muestra en detalle en el capítulo de Implementación5.

4.3.3 Estructura de directorios

En la figura 4.5 se muestra la estructura de directorios de la plataforma construida.

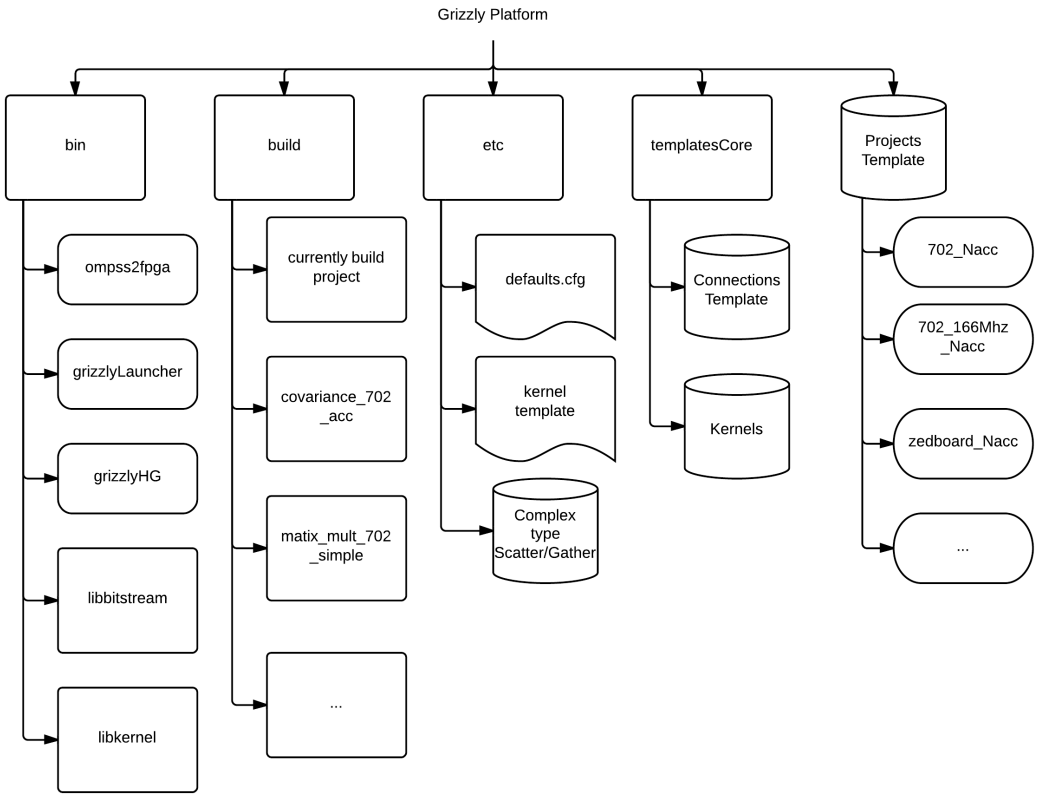

Figura 4.5: Plataforma Grizzly - Estructura

Se divide en:

bin Binarios para ejecutar la plataforma y librerías creadas.

build Almacena todos los proyectos que se crean. Se guardan aquí automáticamente.

etc Archivos de configuración y "Base de datos" para el *Scatter/Gather* de tipos complejos.

templatesCore Tenemos por separado los *kernels* y los *templates* de conexiones.

templatesProject *Templates* de proyectos.

4.3.4 *Syntax* de la línea de comando

GrizzlyHG se ha diseñado para que se pueda ejecutar via línea de comando. La sintaxis de la línea de comando se muestra a continuación.

Dispone de 3 modos:

wrapper Únicamente para Generación Kernel.

bitstream Únicamente para Generación Hardware.

wrapperbitstream Generación del Kernel seguido de Generación Hardware. La Generación Hardware sintetiza los *kernels* creados durante la Generación del Kernel.

A pesar de que todos utilizan una sintaxis muy parecida, dependiendo del modo se disponen de diferentes parámetros.

A continuación se muestra la sintaxis inicial para la selección de un modo en GrizzlyHG:

```
usage: grizzlyHG [-h] {wrapper,bitstream,wrapperbitstream} ...

positional arguments:
  {wrapper,bitstream,wrapperbitstream}
                        sub-command help
    wrapper             Create wrapper
    bitstream           Generate bitstream
    wrapperbitstream    Create wrapper and generate bitstream for that
        wrapper

optional arguments:
  -h, --help            show this help message and exit
```

Generación Kernel: wrapper

La sintaxis del modo wrapper es la siguiente:

```
usage: grizzlyHG wrapper [-h] [-code_file2 NAME] [-t TEMPLATE] [-o OUTPUT]
                         code_file

positional arguments:
  code_file         Path to input code file with pragmas

optional arguments:
  -h, --help        show this help message and exit
  -code_file2 NAME  Path to input code file with pragmas
  -t TEMPLATE       Template to use for wrapper
  -o OUTPUT         Where to write the folder containing wrapper and script.
      tcl
```

Generación Kernel para un código

A continuación se muestra la llamada para generar el kernel del código MxM.cpp.

```
grizzlyHG wrapper MxM.cpp
```

Si queremos indicar parámetros como por ejemplo cambiar el *template* (-t) a new_template y cambiar el directorio donde se guardarán los kernels generados (-o) a "/tmp/kernels" la llamada sería:

```
grizzlyHG wrapper MxM.cpp -t new_template -o /tmp/kernels
```

Generación Kernel para dos códigos

La llamada para generar el kernel de los códigos MxM.cpp y covariance.cpp es la siguiente:

```
grizzlyHG wrapper MxM.cpp -code_file2 covariance.cpp
```

Generación Hardware: bitstream

A continuación se muestra la sintaxis del modo bitstream:

```
usage: grizzlyHG bitstream [-h] [-proj PROJECT] [-coreN NUM]
                           [-connect CONNECTION] [-core2 NAME] [-coreN2 NUM]
                           [-connect2 CONNECTION]
                           core

positional arguments:
  core                  Core to use [Has to be on templatesCore/codeCore]

optional arguments:
  -h, --help            show this help message and exit
  -proj PROJECT         Project to use [Has to be on templatesProject folder
      ]
  -coreN NUM            How many cores [Has to be on templatesProject folder
      ]
  -connect CONNECTION   Connections to use [Has to be on templatesCore/mhs
      folder]
  -core2 NAME           Core to use [Has to be on templatesCore/codeCore]
  -coreN2 NUM           (Useful if -core2 present) How many cores [Has to be
       on templatesProject folder]
  -connect2 CONNECTION  (Useful if -core2 present) Connections to use [Has
      to be on templatesCore/mhs folder]
```

Los *kernels* deben estar dentro del directorio templatesCore/codeCore o indicar la ruta absoluta. Las conexiones deben estar dentro de templatesCore/mhs o indicar la ruta absoluta.

Generación Hardware para un kernel

La llamada para generar el bitstream a partir de un *kernel* existente, matrix_multiply_core, del cual queremos 3 aceleradores (-coreN) y conectarlos utilizando la conexión "simple"(-connect) es la siguiente:

```
grizzlyHG bitstream matrix_multiply_core -coreN 3 -connect simple
```

Generación Hardware para dos *kernels*

Si queremos generar el bitstream para dos cores diferentes a partir de kernels existentes, uno con el nombre de matrix_multiply_core y el otro con el nombre de covariance_core, la llamada quedaría de la siguiente manera:

```
grizzlyHG bitstream matrix_multiply_core -core2 covariance_core
```

Si quisieramos tener dos *cores* de covariance_core en vez de uno como viene por defecto, la llamada quedaría de la siguiente manera:

```
grizzlyHG bitstream MxM -core2 covariance -coreN2 2
```

Para cada *core* se pueden especificar diferentes opciones. Para el primer *core* se utilizan las opciones "-coreN" y "-connect", para el segundo *core* (-core2) se utilizan "-coreN2" y "-connect2".

Generación Kernel & Generación Hardware: wrapperbitstream

A continuación se muestra la sintaxis para el modo wrapperbitstream:

```
usage: grizzlyHG wrapperbitstream [-h] [-proj PROJECT] [-coreN NUM]
                                  [-connect CONNECTION] [-code_file2 NAME]
                                  [-coreN2 NUM] [-connect2 CONNECTION]
                                  [-t TEMPLATE]
                                  code_file

positional arguments:
  code_file             Path to input code file with pragmas

optional arguments:
  -h, --help            show this help message and exit
  -proj PROJECT         Project to use [Has to be on templatesProject folder
      ]
  -coreN NUM            How many cores [>0]
  -connect CONNECTION   Connections to use [Has to be on templatesCore/mhs
      folder]
  -code_file2 NAME      Path to input code file with pragmas
  -coreN2 NUM           (Useful if -code_file2 present) How many cores [Has
      to be on templatesProject folder]
  -connect2 CONNECTION  (Useful if -code_file2 present) Connections to use [
      Has to be on templatesCore/mhs folder]
  -t TEMPLATE           Template to use for wrapper
```

Generación del Kernel y Generación Hardware para un código

Para realizar la generación del *bitstream* a partir del código MxM.cpp se hace la llamada de la siguiente manera:

```
grizzlyHG wrapperbitstream MxM.cpp
```

Generación del Kernel y Generación Hardware para dos código

La siguiente llamada muestra como generar un *bitstream* con *cores* heterogéneos a partir de los códigos MxM.cpp y covariance.cpp.

```
grizzlyHG wrapperbitstream  codigos/MxM.cpp -code_file2  codigos/covariance.
    cpp
```

Las opciones para los *kernels* generados son idénticas a las mostradas en la sección de del modo bitstream.

5

Implementación

A continuación se detallan los elementos introducidos en el capítulo anterior referentes a GrizzlyHG.

5.1 GrizzlyHG

GrizzlyHG es la aplicación resultante de integrar la Generación Kernel y la Generación Hardware. A partir de los códigos a acelerar genera el *bitstream* correspondiente. Inicialmente genera el/los *kernels* y después los utiliza para generar el *bitstream*.

Ofrece tres modos distintos:

Generación del kernel A partir de los códigos de entrada genera los *kernels*

Generación del hardware A partir de los *kernels* genera el *bitstream*

Generación del kernel y del hardware A partir de los códigos de entrada crea los *kernels* para generar el *bitstream* correspondiente

Ademas GrizzlyHG tiene una interfaz que provee de diferentes parámetros para la generación del *bitstream,* son los mostrados anteriormente en la sección 4.3. Ofrece unos valores por defecto para los parámetros opcionales. Estos valores se encuentran en etc/defaults.cfg y pueden ser editados por el usuario según le convenga.

5.2 Generación Kernel

A continuación se describe la implementación del proceso de la generación kernel mostrada en la figura 5.1. Inicialmente, para el archivo (código) de entrada se buscan todas las *task* que hay en ese archivo utilizando el Finder. Cada *task* está compuesta por 2 *pragmas* OmpSs y la función a la cual hacen referencia esos *pragmas*. Para cada *task* se extraerá información de la función y los *pragmas* mediante el Analyzer. Para crear el *kernel* se utilizará WrapperGen. Finalmente, TclGenerator añade el archivo *tcl* de compilación correspondiente.

Para cada *task* se genera un *kernel* y un archivo *tcl*.

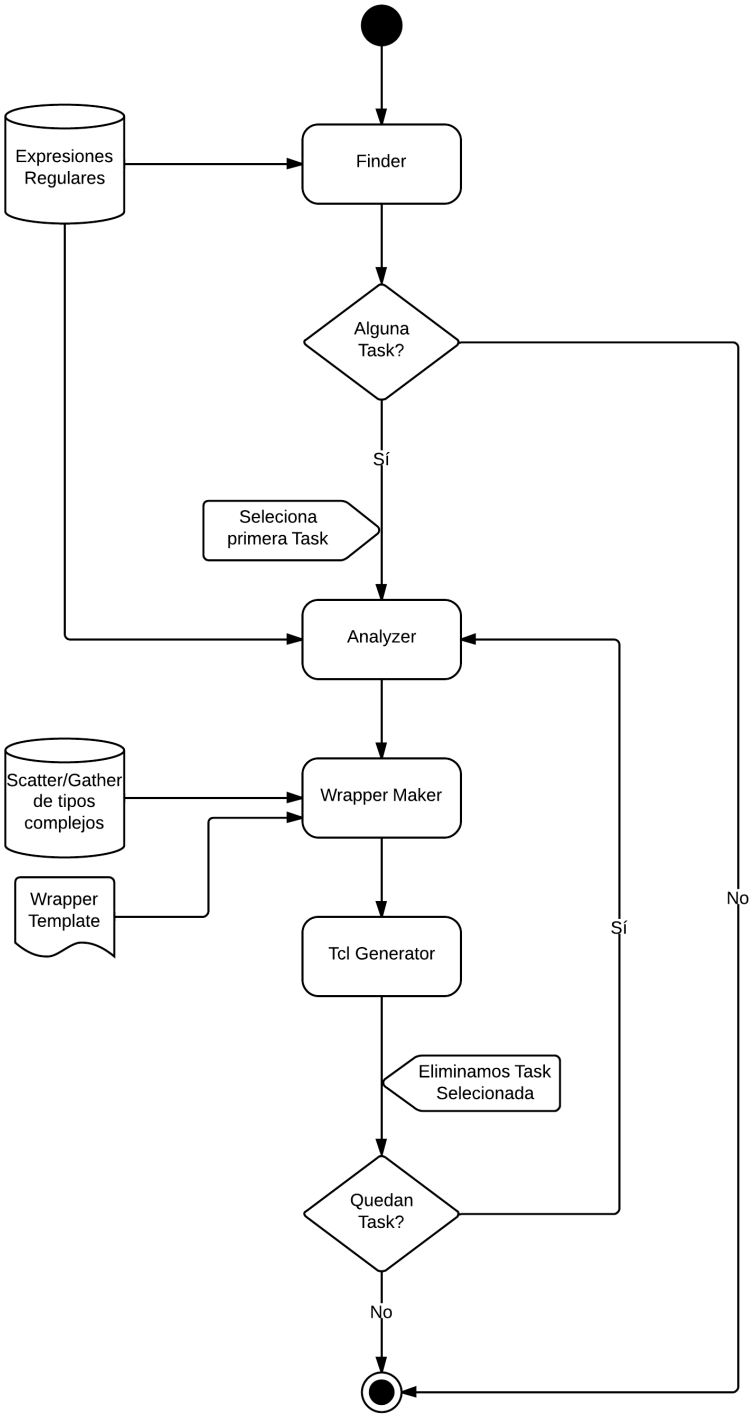

Figura 5.1: Generacion Kernel - Diagrama Funcional

5.2.1 Finder

Mediante expresiones regulares encuentra:

Includes Inclusión de librerías.

Defines Definiciones de pares identificador-*token*.

Task Cada *task* constituye un elemento a acelerar. Está compuesta por *pragmas* OmpSs y la función a la que hacen referencia.

Para identificar los *includes*, *defines*, *pragmas* y funciones se utilizan expresiones regulares.

Includes

Las expresiones regulares relativas a los *includes* buscan el siguiente patrón:

```
1 #include "archivo"
2 #include <archivo>
```

Defines

Las expresiones regulares relativas a los *defines* buscan el siguiente patrón:

```
1 #define identificador token
```

Task

Las expresiones regulares relativas a las *Task* buscan el siguiente patrón:

```
1 #<pragma OmpSs que especifica el target>
2 #<pragma OmpSs que especifica la task>
3 <cabecera de la funcion a acelerar>
4 <cuerpo de la funcion a acelerar>
```

Hay dos maneras de especificar las directivas OmpsSs. A continuación se describen ambos.

Modo In/Out: Este modo utiliza las cláusulas "copy_in(...)" y "copy_out(...)". La cláusula "copy_in(...)" se utiliza para indicar los conjutos de datos que pueden necesitar ser transferidos al dispositivo antes de la ejecucción del código asociado. La cláusula "copy_out(...)" se utiliza para indicar los conjuntos de datos que pueden necesitar ser transferidos desde el dispositivo después de la ejecucción del código asociado.

```
1
2 #pragma omp target device(fpga) copy_in(...) copy_out(...) [↩
     hw_storage(...)]
3 #pragma omp task [in(...) out(...)]
4 <cabecera funcion>
5 {
6   <cuerpo funcion>
7 }
8
9 Nota: si aparece "in(...) out(...)" en el segundo pragma unicamente↩
      se utiliza para comprobar la coherencia con el primer pragma.
```

Modo deps: Este modo utiliza la cláusula "copy_deps". La cláusula "copy_deps" indica que en caso de que el código asociado tengo alguna cláusula de dependencia las cláusulas "in(..)" y "out(...)" serán consideradas como "copy_in(...)" y "copy_out(...)" respectivamente.

```
1
2 #pragma omp target device(fpga) copy_deps [hw_storage(...)]
3 #pragma omp task in(...) out(...)
4 <cabecera funcion>
5 {
6   <cuerpo funcion>
7 }
```

La cláusula "hw_storage(...)" se ha añadido recientemente a OmpSs para poder especificar la cantidad de memoria a crear en hardware. Anteriormente no existía pero para poder realizar la automatización ha sido necesario su inclusión.

Si en el *pragma* que especifica el dispositivo objetivo no aparece la cadena "fpga", *device(fpga)*, la aplicación no genera ningún *kernel* para esa *task*.

Los caracteres "(...)" representan una lista de identificadores. En el caso de que el identificador tenga más de una dimensión, como *arrays*, matrices, etc, se debe indicar el tamaño de la siguiente manera:

```
1 identificador [<indice primer elemento>:<indice ultimo elemento>]
2
3 Ejemplo:
4 copy_in(in_a[0:SIZE-1], in_b[0:SIZE-1])
```

Con esto podemos reducir la cantidad de datos a enviar entre el ARM y la FPGA. Si no necestiamos enviar todos los elementos del *array* basta con indicar los índices de la siguiente manera:

```
1 Array A de N elementos
2 A[0:N-1] - Todo el array
3 A[p:r]   - Del elemento p al elemento r, ambos incluidos
```

Task de ejemplo:

En el código que se muestra a continuación podemos ver un ejemplo de *Task*.

```
1  #pragma omp target device(fpga) copy_deps hw_storage(in_a[0:SIZE↩
       -1], in_b[0:SIZE-1], out_c[0:SIZE-1])
2  #pragma omp task in (in_a[0:SIZE-1], in_b[0:SIZE-1]) out(out_c[0:↩
       SIZE-1])
3  void matrixmultiply (float in_a[SIZE], float in_b[SIZE], float ↩
       out_c[SIZE])
4  {
5      int i, j, k;
6
7      for ( i = 0; i < M; i++ )
8      {
9          for ( j = 0; j < N; j++ )
10         {
11             out_c[i*N + j] = 0;
12             #pragma AP PIPELINE II=1
13             for ( k = 0; k < N; k++ )
14             {
15                 out_c[i*N + j] += in_a[i*N + k] * in_b[k *N +j];
16             }
17         }
18     }
19 }
```

5.2.2 Analyzer

Extrae la información de cada *task*. A partir de los *pragmas* y la cabecera de la función crea una estructura de datos que contiene la información útil para la generación del *kernel*.

Se extraen mediante expresiones regulares:

- Parámetros de entrada
- Parámetros de salida

- Tamaños de dimensión
- Tipo del parámetro
- Nombre de la función

La estructura creada contiene:

1. Nombre de la función
2. Lista ordenada de parámetros de entrada - El orden es el mismo que el de la cláusula hw_storage.
3. Lista ordenada de parámetros de salida - El orden es el mismo que el de la cláusula hw_storage.

Para cada parámetro tiene la siguiente información:

1. Identificador
2. Tipo
3. Dimensión

Modo deps

A continuación se muestra un código de ejemplo de una *Task* siguiendo el Modo deps:

```
1 #pragma omp target device(fpga) copy_deps hw_storage(array_a[0:N↩
    -1], array_b[0:N-1], array_out[0:N-1])
2 #pragma omp task in(array_a[0:N-1], array_b[0:N-1]) out(array_out↩
    [0:N-1])
3 void matrix_multiply(float array_a[N], float array_b[N], float ↩
    array_c[N])
4 {
5   ...
6 }
```

De *hw_storage,* situado en el primer *pragma,* se extrae el orden en el que se debe realizar el empaquetado y desempaquetado de los parámetros. Esto se realiza debido a que pueden existir dependencias entre ellos. También se extraen los índices de los parámetros multidimensionales.

Del segundo *pragma* se extrae qué parámetros son de entrada y cuales de salida.

De la cabecera de la función se obtiene el nombre de la función y el tipo de los parámetros.

Modo In/Out

A continuación se muestra un código de ejemplo de una *Task* siguiendo el Modo In/Out:

```
1 #pragma omp target device(fpga) copy_in(array_a[0:N-1], array_b[0:N↩
     -1]) copy_out(array_out[0:N-1]) hw_storage(array_a[0:N-1], ↩
     array_b[0:N-1], array_out[0:N-1])
2 #pragma omp task in(array_a[0:N-1], array_b[0:N-1]) out(array_out↩
     [0:N-1])
3 void matrix_multiply(float array_a[N], float array_b[N], float ↩
     array_c[N])
4 {
5   ...
6 }
```

Del primer *pragma* se extrae:

1. De *hw_storage* el orden en el que se debe realizar el empaquetado y desempaquetado de los parámetros.

2. Los índices de los parámetros multidimensionales.

3. Que parámetros son de entrada y cuales de salida.

Del segundo *pragma* no se extrae información nueva.

De la cabecera de la función se extrae el nombre de la función y el tipo de los parámetros.

Para ambos modos se realiza la comprobación de que todos los parámetros que aparecen en la cabecera de la función han sido especificados en los *pragmas* y, que no haya ninguna incoherencia en los tamaños multidimensionales especificados en el primer *pragma* con los del segundo. Es decir, si no hubiéramos definido en el segundo *pragma* el parámetro array_b ni como de entrada ni de salida, nos saldría un error. Otro ejemplo, si en el primer *pragma* los índices de *array_a* son "0" y "N-2", nos saldría una advertencia diciendo que no concuerdan con lo que aparece en el segundo *pragma*, "0" y "N-1".

5.2.3 WrapperMaker

A partir de la estructura de datos creada por Analyzer construimos el *kernel*. Para construirlo utilizamos un *template* como base para añadir versatilidad. El *template* diseñado está basado en la implementación manual que se ha realizado de numerosos *kernels*.

Este *kernel* es necesario para poder introducir el código a acelerar en el hardware reconfigurable. Se necesita manejar la descarga de datos entre en el ARM y la FPGA y obtenerlos a traves de una estructura de datos concreta, es por ello que se requiere esta capa software adicional, el *kernel*.

En la figura **??** podemos ver como interviene el kernel para poder convertir el código a acelerar en un core para que pueda ser utilizado por las herramientas de hardware. La Generación Kernel extrae las partes del código a acelerar y genera el kernel correspondiente de cada una de ellas. Posteriormente las herramientas hardware lo utilizarán para generar el core acelerador para así poder integrarlo con el resto del hardware.

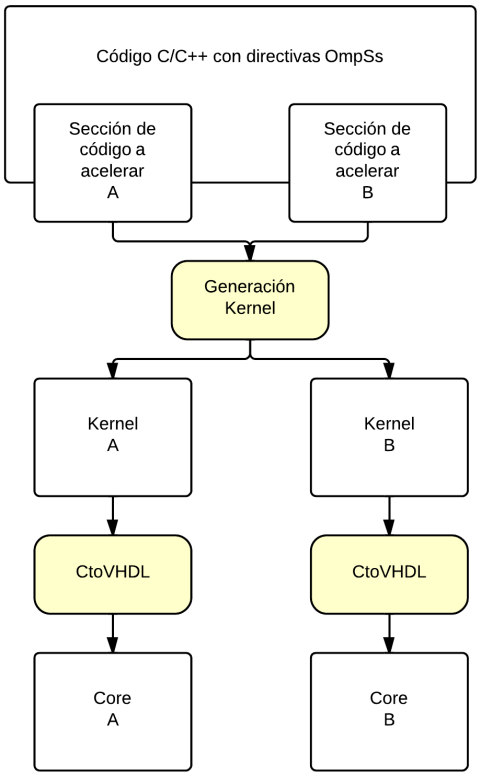

Figura 5.2: Capas software para obtener el core - El kernel posibilita el pasar del código al core

El *kernel* tiene una estructura en particular, se detalla a continuación, para poder integrar el código a acelerar con el hardware.

```
/*
 * Wrapper to handle conetion between core and system
 *
*/
#INCLUDES#

#DATA#

void #function_name#_wrapper (hls::stream<#stream_type#> &#stream_in_id#,
    hls::stream<#stream_type#> &#stream_out_id#) {
    #pragma HLS interface ap_ctrl_none port=return
    #pragma HLS resource variable=#stream_in_id# core=AXI4Stream
    #pragma HLS resource variable=#stream_out_id# core=AXI4Stream

    /* Input */
#SCATTER#

#GATHER_DECLARATION#

    /* Call funtion */
    #function_name#(#param_list#);

    /* Output */
#GATHER#
}
```

El *kernel* incluye las secciones:

Includes *Include* del código original y otros *includes* si fuera necesario.

Data *Defines* y otras variables globales.

Scatter Desempaquetado de datos provenientes del *stream* de entrada.

Gather declaration Declaración de los parámetros que se utilizan para el *gather* y aparecen en la llamada a la función.

Gather Empaquetado de datos para añadirlos al *stream* de salida.

WrapperMaker utiliza estas secciones para añadir el código generado automáticamente. Como se puede observar, el código generado corresponde al empaquetado y el desempaquetado de datos, y la llamada a la función original a acelerar.

Empaquetado y Desempaquetado de datos

Consiste en añadir o extraer datos de un *stream* el cual es de tipo entero. Se puede utilizar otro tipo distinto pero hemos elegido éste ya que es el mas versátil y el comúnmente utilizado por aplicaciones. Todos los datos que el *core* deba recibir desde el ARM se recibirán mediante enteros a través del *stream* de entrada. Lo mismo ocurre con los que se deban enviar pero utilizando el *stream* de salida.

Dado que las aplicaciones utilizan diferentes tipos se necesita añadir una conversión a/desde entero. Esta conversión es simple si ambos tipos ocupan la misma cantidad de *bytes* en memoria. Sin embargo, a veces se requieren otros tipos que no cumplen esta regla. Si el tipo es simple y pertenece al estándar de C, la plataforma puede generar el *kernel*. Se han incluido conversiones para los tipos estándar simples. Si el tipo es complejo, es decir, es una *struct* o alguna estructura de datos con diferentes campos la conversión no es trivial.

La cantidad de tipos complejos que pueden tener las aplicaciones es prácticamente infinito. La plataforma debe ser escalable para todas las aplicaciones, por ello se ha creado una interfaz muy simple para que el usuario pueda extender la plataforma para los tipos que haya creado el mismo. Debe crear un archivo JSON que indique como se debe realizar el empaquetado y desempaquetado de ese tipo. El archivo debe guardarse en el directorio etc/complex_data con el nombre <nombre del tipo>.json.

La estructura del archivo JSON debe ser la siguiente:

```
{
    "type": "<nombre de la estructura/tipo>",
    "structure": [{
        "type": "<tipo del primer elemento>",
        "name": "<identificador del primer elemento>",
    }, {
        "type": "<tipo del segundo elemento>",
        "name": "<identificador del segundo elemento>",
    }, {
        ...
    }],
    "scather": "<Desempaquetado de la estructura>",
    "gather": "<Empaquetado de la estructura>"
}
```

El tipo creado por el usuario puede tener tantos elementos como precise.

Por ejemplo hemos añadido el empaquetado y desempaquetado para el tipo "cint32_t". Se trata de un tipo de dato que se utiliza para números complejos. Contiene dos variables de tipo "int32_t", una para almacenar la parte real y la otra la parte imaginaría. El tipo "int32_t" es del estándar de C. El archivo que incluye la información para realizar el empaquetado y desempaquetado de datos es etc/complex_data/cint32_t.json y su contenido se muestra a continuación:

```
{
    "type": "cint32_t",
    "structure": [{
        "type": "int32_t",
        "name": "real"
    }, {
        "type": "int32_t",
        "name": "imag"
    }],
    "scather": "
    {ident}{stream_in_type} foo_real = {stream_in_name}.read();\n
    {ident}{stream_in_type} foo_imag = {stream_in_name}.read();\n
```

```
    {ident}{name}{row}{col}.real = {castsc[0]}foo_real;\n
    {ident}{name}{row}{col}.imag = {castsc[1]}foo_imag;\n
    ",

    "gather": "
    {ident}int32_t foo_real = {name}{row}{col}.real;\n
    {ident}int32_t foo_imag = {name}{row}{col}.imag;\n
    {ident}{stream_out_name}.write({castga[0]}foo_real);\n
    {ident}{stream_out_name}.write({castga[1]}foo_imag);\n
    "
}
```

Como se puede ver la interfaz añadida para que sea escalable incluye *tokens* como *ident, stream_in_type, stream_in_name* entre otros. Los *tokens* son remplazados por su valor correspondiente según el contexto. La lista de la interfaz completa se describe en la siguiente sección.

Interfaz para el empaquetado y desempaquetado de tipos complejos

A continuación se muestra la interfaz creada con la cual el usuario puede hacer compatible cualquier tipo de dato que utilice su aplicación con la plataforma. Estos *tokens* deben ir entre "{" "}".

Formato:

ident Para identar el código correctamente. Esta identación se añade además de la que ya incluye la generación. Por si el usuario quiere crear bucles identados u otros casos.

Identificadores:

name Identificador de la variable del tipo que estamos realizando el empaquetado/desempaquetado.

stream_in_name Identificador del *stream* de entrada.

stream_out_name Identificador del *stream* de salida.

Tipos:

stream_in_type Tipo del *stream* de entrada.

stream_out_type Tipo del *stream* de salida.

***Cast*:**

Se añade esta opción debido a que podría ser que el tipo del *stream* y el tipo del elemento coincidieran y no se necesitara el *cast*. Al añadir esta opción, si en algún momento se cambia el tipo del *stream*, no se necesita modificar el archivo JSON ya que la plataforma comprueba los tipos, en caso de que se requiera añadirá el *cast* necesario donde se encuentre este *token*.

castsc *Cast* para el desempaquetado.

castga *Cast* para el empaquetado.

Multidimensionales:

En caso de que los tipos se encuentren en *arrays* de una dimensión o más de una se sustituirán por la variable correspondiente. En caso contrario la plataforma lo ignorará para tratarlo como un escalar.

row Filas, primera dimension.

col Columnas, segunda dimension.

El uso de esta interfaz se puede ver en los ejemplos de JSON que hemos creado incluidos en el anexo en la sección A

5.2.4 TclGenerator

Dado que queremos automatizar la generación del bitstream debemos enviar las ordenes a los programas de alguna manera que no requiera el uso de la interfaz

gráfica. Los programas utilizados disponen de un modo *tcl* para trabajar con comandos que no hacen uso de su interfaz. De esta manera mediante las directrices *tcl* conseguimos automatizar el proceso.

TclGenerator crea un archivo que contendrá las directrices *tcl* para la sintetización de C a VHDL. Dado que las directrices son diferentes para cada *kernel* generado, es más adecuado generarlo en la Generación Kernel.

La estructura es la que se muestra a continuación:

```
############################################################
## This file is generated automatically by TaskTcl
## Edit as you need
############################################################

open_project <nombre proyecto> -reset
set_top <nombre funcion>
add_files <nombre archivo>
add_files -tb <nombre archivo>
open_solution "<nombre solucion>" -reset
set_part {xc7z020clg484-1}
create_clock -period 10

csynth_design
export_design -evaluate vhdl -format pcore
exit
```

Por defecto a <nombre proyecto>le asignamos el nombre de la función original para mantener la coherencia, es el nombre del directorio que tendrá el *core*. Del mismo modo, <nombre solución>es siempre "solution", ha sido escogido arbitrariamente. *set_part* indica la arquitectura para la cual se creará el *core*. En nuestro caso corresponde a la *ZC702*. Los *cores* generados también son compatibles para la *Zedboard* dado que tiene el mismo chip que la *ZC02*.

5.3 Generación Hardware

A partir de la experiencia en la generación manual de hardware para diferentes ejemplos, hemos observado que podríamos crear un *template* del proyecto de PlanAhead y expandirlo para adaptarlo según las necesidades. El proyecto de

PlanAhead contiene toda la configuración del *SoC*, sobretodo, todas las conexiones y configuraciones relacionadas con la parte reconfigurable.

Para generar el hardware utilizamos una base y uno o más *cores*. La base, o *template*, consiste en un proyecto de PlanAhead. Inicialmente teníamos diferentes *templates* para cada *SoC* y tipo de aplicación pero no era escalable. Posteriormente creamos *templates* genéricos para cada *SoC*, la *Zedboard* y la *ZC702*, que pueden ser utilizados para cualquier aplicación.

En el proceso de la Generación Hardware, primero se convierten los *kernels* a cores en VHDL para que puedan ser procesados por las herramientas de síntesis. Para ello se utiliza VivadoHLS para transformar el *kernel* escrito en C/C++ a VHDL, CtoVHDL. Una vez tenemos el *core* generado se integra con el proyecto de PlanAhead configurando las conexiones con los diferentes elementos del *SoC*. Para realizar esta integración se añade el *core* al proyecto y se editan las configuraciones dentro del archivo MHS. Finalmente se ejecuta BitstreamGeneration para realizar la síntesis, la implementación y la generación del *bitstream* utilizando PlanAhead. Adicionalmente se añade la conversión del *bitstream* generado a formato binario utilizando la herramienta bit2bin.

5.3.1 CtoVHDL

Para realizar la conversión de código C a VHDL se utiliza la herramienta VivadoHLS. Además del *kernel*, se le añade como entrada el archivo *tcl* correspondiente al *kernel* que queremos convertir. Obtenemos el *core* en un lenguaje HDL para que pueda ser procesado por las herramientas de síntesis que ejecutaremos para acabar generando el *bitstream*. La duración de este proceso depende de la complejidad del código. La duración de la sintetización de los *kernels* utilizados se puede ver en la sección 6.3.

El archivo *tcl* es el mostrado anteriormente en el apartado 5.2.4.

5.3.2 MHSconfigurator

La configuración de las conexiones se realiza mediante la edición del archivo MHS del proyecto PlanAhead. En este archivo se encuentran los diferentes elementos que se configuraran en el *SoC* y como se conectarán entre si.

Añadir o modificar nuevas conexiones de manera automática resulta algo complicado dado que cada aplicación puede tener necesidades distintas. El usuario puede añadir nuevas maneras de realizar estas conexiones tal y como se explica a continuación. El MHSConfigurator utiliza dos archivos en los cuales se indican qué elementos se deben añadir y cómo se deben modificar las conexiones, son los archivos *code* y *connection*. Si el usuario tiene unas necesidades distintas para su aplicación a las incluidas, tan solo debe añadir un directorio con estos dos archivos, editados según requiera, dentro del directorio templatesCore/mhs. Cada *core* puede tener una configuración distinta como hemos visto anteriormente.

MHSconfigurator instancia un par *code-connection* para cada core que se vaya a incluir. Una vez instanciados, añade todos los *code* al MHS base del proyecto y posteriormente edita todas las conexiones según se especifica en cada *connection*.

A continuación se muestra un fragmento del archivo MHS con la configuración base para la *ZC702*.

```
 PARAMETER VERSION = 2.1.0


 PORT PS_CLK = PS_CLK, DIR = I, CLK_FREQ = 33333333, SIGIS = CLK
 PORT PS_SRSTB = PS_SRSTB, DIR = I
...
 PORT DDR_Addr = DDR_Addr, DIR = IO, VEC = [14:0]


BEGIN processing_system7
 PARAMETER INSTANCE = zynq_ps7
 PARAMETER HW_VER = 4.01.a
 PARAMETER C_EN_DDR = 1
 PARAMETER C_FCLK_CLK0_FREQ = 166666672
 PARAMETER C_FCLK_CLK1_FREQ = 200000000
 PARAMETER C_FCLK_CLK2_FREQ = 41666668
 PARAMETER C_FCLK_CLK3_FREQ = 41666668
...
 PARAMETER C_INTERCONNECT_S_AXI_ACP_MASTERS = %
     C_INTERCONNECT_S_AXI_ACP_MASTERS%
...
 BUS_INTERFACE M_AXI_GP0 = xd_dm_axi_interconnect_gm0
```

```
 BUS_INTERFACE S_AXI_ACP = xd_dm_axi_interconnect_acp
...
 PORT Irq_F2P = %Irq_F2P%
END

BEGIN axi_interconnect
 PARAMETER INSTANCE = xd_dm_axi_interconnect_gm0
 PARAMETER HW_VER = 1.06.a
 PORT INTERCONNECT_ACLK = zynq_ps7_FCLK_CLK0
 PORT INTERCONNECT_ARESETN = zynq_ps7_FCLK_RESET0_N
END

BEGIN axi_interconnect
 PARAMETER INSTANCE = xd_dm_axi_interconnect_acp
 PARAMETER HW_VER = 1.06.a
 PARAMETER C_AXI_SUPPORTS_USER_SIGNALS = 1
 PORT INTERCONNECT_ACLK = zynq_ps7_FCLK_CLK0
 PORT INTERCONNECT_ARESETN = zynq_ps7_FCLK_RESET0_N
END
```

En el se pueden observar como aparecen los elementos de interconexion *axi_interconnect,* frecuencias de relojes, interrupciones en *Irq_F2P*, etc. Los campos que aparecen con %<cadena>% indican al MHSConfigurator dónde debe editar los parámetros de conexión.

Un ejemplo de archivos de configuración *code* y *connection* que incluye la plataforma son los siguientes:

Code (fragmento)

```
BEGIN axi_dma
 PARAMETER INSTANCE = xd_dm_axidma_%I%
 PARAMETER HW_VER = 6.03.a
 PARAMETER C_BASEADDR = 0x400%I%0000
 PARAMETER C_HIGHADDR = 0x400%I%FFFF
...
 PARAMETER C_INTERCONNECT_S_AXI_LITE_MASTERS = zynq_ps7.M_AXI_GP0
 BUS_INTERFACE S_AXI_LITE = xd_dm_axi_interconnect_gm0
 BUS_INTERFACE M_AXI_SG = xd_dm_axi_interconnect_acp
 BUS_INTERFACE M_AXI_MM2S = xd_dm_axi_interconnect_acp
 BUS_INTERFACE M_AXI_S2MM = xd_dm_axi_interconnect_acp
```

```
 BUS_INTERFACE M_AXIS_MM2S = xd_dm_axidma_%I%_M_AXIS_MM2S
 BUS_INTERFACE S_AXIS_S2MM = %CORE_FUNCTION_NAME%_%I%_p_hls_var_out
...
 PORT mm2s_prmry_reset_out_n = xd_dm_axidma_%I%_mm2s_prmry_reset_out_n
 PORT mm2s_cntrl_reset_out_n = xd_dm_axidma_%I%_mm2s_cntrl_reset_out_n
 PORT s2mm_prmry_reset_out_n = xd_dm_axidma_%I%_s2mm_prmry_reset_out_n
 PORT s2mm_sts_reset_out_n = xd_dm_axidma_%I%_s2mm_sts_reset_out_n
 PORT mm2s_introut = xd_dm_axidma_%I%_mm2s_introut
 PORT s2mm_introut = xd_dm_axidma_%I%_s2mm_introut
END

BEGIN %CORE_FUNCTION_NAME%
 PARAMETER INSTANCE = %CORE_FUNCTION_NAME%_%I%
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE out_stream_v = %CORE_FUNCTION_NAME%_%I%_p_hls_var_out
 BUS_INTERFACE in_stream_v = xd_dm_axidma_%I%_M_AXIS_MM2S
 PORT aclk = zynq_ps7_FCLK_CLK0
 PORT aresetn = xd_dm_axidma_%I%_s2mm_sts_reset_out_n
END
```

Este archivo también incluye %<cadena>%, en este caso se utiliza para añadir el nombre del *core*, %CORE_FUNCTION_NAME%, y el número de instancia, %I%. El número de instancia se añade para el caso de los múltiples *cores* homogéneos.

Este archivo indica qué se debe añadir al archivo MHS. En este caso un *axi_dma* para la transmisión de datos y un *core*.

Connection

```
%C_INTERCONNECT_S_AXI_ACP_MASTERS%
xd_dm_axidma_%I%.M_AXI_SG & xd_dm_axidma_%I%.M_AXI_MM2S & xd_dm_axidma_%I%.
    M_AXI_S2MM
%Irq_F2P%
xd_dm_axidma_%I%_mm2s_introut & xd_dm_axidma_%I%_s2mm_introut
```

En *connection* se indican qué conexiones se deben modificar en el archivo MHS base. En este caso %C_INTERCONNECT_S_AXI_ACP_MASTERS% y %Irq_F2P%.

Estas adiciones/modificaciones se realizan para cada *core*, si tenemos 2 de ellos se harán 2 adiciones (*code*) y 2 modificaciones (*connection*). En la figura5.3 se muestra la arquitectura resultante para dos cores aceleradores.

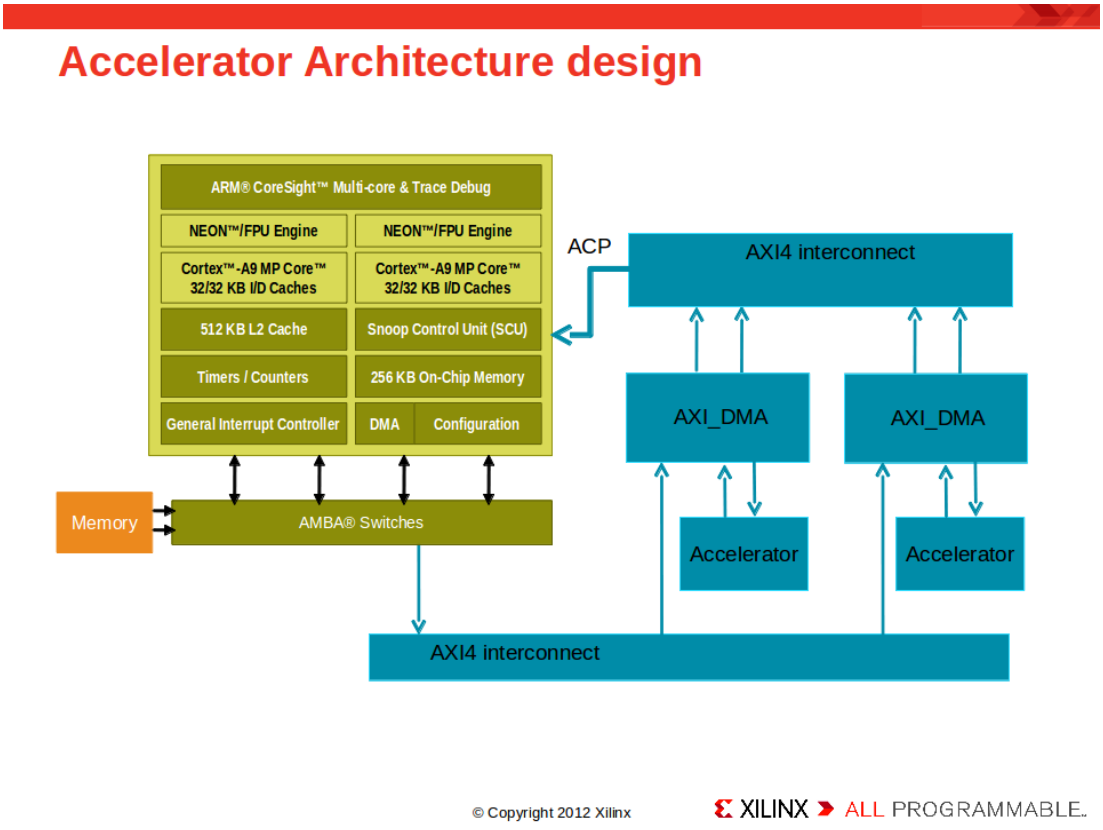

Figura 5.3: Diseño arquitectura aceleradora - Diagrama de conexión del ARM con 2 *cores*

5.3.3 BitstreamGeneration

En BitstreamGeneration se realiza la síntesis y la implementación de todo el proyecto y finalmente se genera el *bitstream*. Todo el proceso se realiza utilizando PlanAhead. Este proceso es el que más dura en tiempo de procesado y depende enteramente de la complejidad del proyecto. La duración en los proyectos que hemos realizado se puede ver en la sección 6.3. Una vez finalizado este proceso obtenemos el .bit para configurar el hardware reconfigurable del *SoC*.

Las instrucciones *tcl* que se utilizan son las siguientes:

```
# planAhead -mode tcl -source bitgen.tcl
open_project /edk/template/template.ppr
# Create a top VHDL
make_wrapper -files [get_files /edk/template/template.srcs/sources_1/edk/
    system/system.xmp] -top -fileset [get_filesets sources_1] -import
# Clean bitstream, implementation, synthesis
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
reset_run impl_1
reset_run synth_1
# Launch synthesis, implementation, bitstream generation
launch_runs synth_1
wait_on_run synth_1
launch_runs impl_1
wait_on_run impl_1
launch_runs impl_1 -to_step Bitgen
wait_on_run impl_1
# Export hardware to sdk
#export_hardware [get_files /edk/template/template.srcs/sources_1/edk/system
    /system.xmp] [get_runs impl_1] -bitstream
#launch_sdk -bit /edk/template/template.sdk/SDK/SDK_Export/hw/system_stub.
    bit -workspace /edk/template/template.sdk/SDK/SDK_Export -hwspec /edk/
    template/template.sdk/SDK/SDK_Export/hw/system.xml
close_project
exit
```

Como se puede observar las últimas líneas para exportar el hardware y abrir el SDK están comentadas. Si quisiéramos generar el *elf* de los ARM por nosotros mismos lo descomentaríamos. Actualmente generar el *elf* de los ARM se realiza mediante la aplicación del BSC, fpgacxx.

5.3.4 Bit2bin

La herramienta bit2bin es externa a este proyecto, no ha sido creada por nosotros. Se realiza la llamada a bit2bin con el .bit como entrada. El archivo binario se genera para cargar el *bitstream* directamente al *SoC* sin necesidad de utilizar un programa como el EDK.

6

Resultados

Una vez finalizado el desarrollo de la plataforma hemos obtenido los resultados que se muestran a continuación. Se han escogido tres aplicaciones que sirven como ejemplo. La plataforma es capaz de procesar prácticamente cualquier tipo de aplicación debido a su escalabilidad y versatilidad.

Si algún tipo de aplicación no es soportada actualmente se puede solventar añadiendo uno o más de los siguientes ítems:

1. Conversiones de tipos complejos
2. *Template* del *kernel*
3. *Template* del proyecto
4. *Code* y *connection*

La plataforma genera el *bitstream* y el ejecutable de los ARM correctamente. Sin embargo, para generar el ejecutable de los ARM de los *SoC* es necesario una librería la cual no disponemos debido a su carácter confidencial. La librería se encarga de la configuración del *AXI DMA* en la transmisión de datos entre el ARM y el *core*. Al no disponer de esa librería no somos capaces de generar el *elf* de los ARM que genera fpgacxx.

La solución es construir nuestra propia aplicación *standalone* para ejecutar en el *SoC*. Dado que no es una tarea sencilla únicamente se ha realizado para Matrix Mulitply.

Todos los demás *bitstreams* generados han sido probados en el BSC en la *ZC702* donde sí disponen de esta librería.

6.1 Entorno de experimentación

Los resultados se han obtenido utilizando una ZC702 conectada a un ordenador a traves de un cable ethernet utilizando un terminal remoto. Se ha utilizado Mercurium 1.99.0, Nanos++ 0.7a. Para la compilación hardware se ha utilizado Xilinx ISE Design 14.4 y Vivado HLS 2012.4. Todos los codigos que corren en el ARM se han compilado con arm-xilinx-linux-gnueabi-g++ (Sourcery CodeBench Lite 2011.09-50) 4.6.1 y arm-xilinx-linux-gnueabi-gcc (Sourcery CodeBench Lite 2011.09-50) 4.6.1 con el parámetro de optimización "-O3". No se ha realizado ninguna vectorización de manera manual ni ninguna otra optimización en el código que corre en los ARM. Se ha realizado *cross-compile* para las aplicaciones en un procesador de 64 bits utilizando linux con suficientes recursos para generar *bitstreams*.

6.2 Aplicaciones Testeadas

A continuación se muestran las aplicaciones que hemos utilizado para testear la plataforma.

En la sección "Código" se muestra el fragmento de código de las aplicaciones que utiliza GrizzlyHG para generar el *kernel*. Los códigos originales son mayores e implementan todo el problema. Solo se adjunta el fragmento útil para la generación del *kernel*.

6.2.1 Matrix Multiply

La aplicación realiza la multiplicación de dos matrices de 32x32 elementos de coma flotante. También se ha probado la versión de 64x64 elementos de coma flotante.

Código

```
#pragma omp target device(fpga) copy_deps hw_storage(in_a[0:SIZE-1], in_b[0:SIZE-1], out_c[0:SIZE-1])
#pragma omp task in (in_a[0:SIZE-1], in_b[0:SIZE-1]) out(out_c[0:SIZE-1])
void matrixmultiply (float in_a[SIZE], float in_b[SIZE], float out_c[SIZE])
{
    int i, j, k;

    for ( i = 0; i < M; i++ )
    {
        for ( j = 0; j < N; j++ )
        {
            out_c[i*N + j] = 0;
                        #pragma AP PIPELINE II=1
            for ( k = 0; k < N; k++ )
            {
                out_c[i*N + j] += in_a[i*N + k] * in_b[k *N +j];
            }
        }
    }
}
```

Kernel generado

```
1  /*
2   * Wrapper function to handle conetion between kernel accelerator ↩
       and system
3   *
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8  #include <sys/time.h>
9  #include <string.h>
10 #include "/home/edu/PFC/codigos_entrada/matrix_mult/↩
       ompss_mmult_core.cpp"
11 #include <hls_stream.h>
12
13
14 #define M_  32
15 #define N_  32
16 #define SIZE_ 1024
17 #define M N_
18 #define N M_
19 #define SIZE SIZE_
20
21
22 void matrixmultiply_wrapper (hls::stream<int> &in_stream, hls::↩
       stream<int> &out_stream) {
23     #pragma HLS interface ap_ctrl_none port=return
24     #pragma HLS resource variable=in_stream core=AXI4Stream
25     #pragma HLS resource variable=out_stream core=AXI4Stream
26
27     /* Input */
28         int i;
29         int j;
30
```

```
31          float in_a[SIZE-1-0 + 1];
32          for ( i = 0; i <= SIZE-1; i++){
33                  #pragma HLS pipeline II=2
34                  int foo = in_stream.read();
35                  in_a[i] = *(float*)&foo;
36          }
37
38          float in_b[SIZE-1-0 + 1];
39          for ( i = 0; i <= SIZE-1; i++){
40                  #pragma HLS pipeline II=2
41                  int foo = in_stream.read();
42                  in_b[i] = *(float*)&foo;
43          }
44
45
46
47          float out_c[SIZE-1-0 + 1];
48
49
50      /* Call funtion */
51      matrixmultiply(in_a, in_b, out_c);
52
53      /* Output */
54          for ( i = 0; i <= SIZE-1; i++){
55                  #pragma HLS pipeline II=2
56                  float foo = out_c[i];
57                  out_stream.write(*(int*)&foo);
58          }
59
60  }
```

Resultado

En la figura 6.1 se muestra la comprobación del funcionamiento del *bitstream* para la multiplicación de 63x64 elementos. El código que corre en el ARM ejecuta la multiplicación de matrices en el mismo ARM para tener los resultados de la

multiplicación y posteriormente los comprueba con el resultado que recibe de la multiplicación de matrices realizado en la FPGA. Como se observa en la última línea el resultado ha sido satisfactorio.

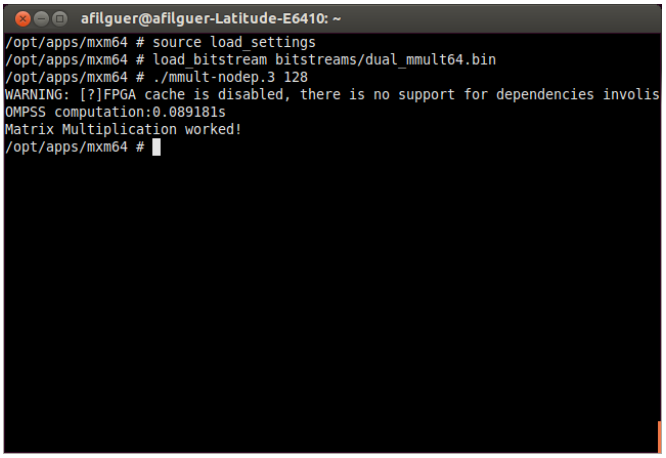

Figura 6.1: Resultado Matrix Multiply - Comprobación funcional del *bitstream* generado para Matrix Multiply

6.2.2 Covariance

La aplicación realiza el cálculo de la covariancia entre diferentes elementos.

Código

```
1  #pragma omp target device(fpga) copy_in(L,M,Cols,Rows,ColsDivM,↩
       row_indices[0:Rows-1],in[0:L-1]) copy_out(out[0:Rows-1][0:Cols↩
       -1]) hw_storage(in[0:MAX_L-1], row_indices[0:MAX_ROWS-1], out[0:↩
       MAX_ROWS-1][0:MAX_COLS-1])
2  #pragma omp task in(L, M, Cols, Rows, ColsDivM, row_indices[0:↩
       MAX_ROWS-1],in[0:MAX_L-1]) out(out)
3  void covariance(cint32_t const *in, int const *row_indices, ↩
       cint64_t out[MAX_ROWS][MAX_COLS], int L, int M, int Cols, int ↩
       Rows, int ColsDivM)
4  {
5  #pragma HLS inline
6
7    // initialize output matrix
8  #pragma HLS resource variable out core=RAM_2P
9  #if PP_UNROLL_FACTOR > 1
10 #  pragma HLS array_partition variable=out  cyclic factor=↩
       UNROLL_FACTOR dim=2
11 #endif
12
13  loop_init_out_outer:
14   for (ap_uint<BW_MAX_ROWS> row = 0; row < Rows; ++row)
15   loop_init_out_inner:
16     for (ap_uint<BW_MAX_COLS> col = 0; col < Cols; ++col)
17       {
18 #pragma HLS pipeline
19 #pragma HLS loop_flatten off
20     out[row][col].real = 0;
21     out[row][col].imag = 0;
22       }
23
```

```
24   // allocate covariance array
25   cint32_t ca[MAX_COLS];
26 #pragma HLS resource variable ca  core=RAM_2P
27 #pragma HLS array_partition variable=ca cyclic factor=UNROLL_FACTOR
28
29   // main loop
30  loop_main:
31   for (int l = 0; l < L; ++l)
32     {
33 #pragma HLS loop_tripcount max=4096
34
35       // delay memory terms in covariance array
36     loop_ca_copy:
37       for (ap_uint<BW_MAX_COLS> i = Cols-1; i >= M; --i)
38 #pragma HLS pipeline
39 #pragma HLS loop_tripcount max=132-6
40 #pragma HLS dependence variable=ca array
41     ca[i] = ca[i-M];
42       // copy new input element
43       int32_t real = in[l].real;
44       int32_t imag = in[l].imag;
45
46     loop_ca_init:
47       for (ap_uint<BW_MAX_COLS> m = 0; m < M; ++m)
48     {
49 #pragma HLS pipeline
50 #pragma HLS loop_tripcount max=6
51       ca[m].real = real - m;
52       ca[m].imag = imag - m;
53     }
54
55       // matrix multiplication
56       if (l >= ColsDivM)
57     innerLoop (out, ca, Cols, row_indices, Rows);
58     }
59 }
```

Kernel generado

```
1  /*
2   * Wrapper function to handle conetion between kernel accelerator ↩
       and system
3   *
4  */
5  #include <cassert>
6  #include <ap_int.h>
7  #include "/home/edu/PFC/codigos_entrada/covariance/covariance_core.↩
       cpp"
8  #include "/home/edu/PFC/codigos_entrada/covariance/complexint.h"
9  #include "/home/edu/PFC/codigos_entrada/covariance/covariance.h"
10 #include <hls_stream.h>
11
12
13 #define PP_UNROLL_FACTOR 2 // allow #if directives with the ↩
       UNROLL_FACTOR
14
15
16 void covariance_wrapper (hls::stream<int> &in_stream, hls::stream<↩
       int> &out_stream) {
17     #pragma HLS interface ap_ctrl_none port=return
18     #pragma HLS resource variable=in_stream core=AXI4Stream
19     #pragma HLS resource variable=out_stream core=AXI4Stream
20
21     /* Input */
22         int i;
23         int j;
24
25         int L;
26         L = in_stream.read();
27
28         int M;
29         M = in_stream.read();
```

```
30
31          int Cols;
32          Cols = in_stream.read();
33
34          int Rows;
35          Rows = in_stream.read();
36
37          int ColsDivM;
38          ColsDivM = in_stream.read();
39
40          int row_indices[MAX_ROWS-1-0 + 1];
41          for ( i = 0; i <= Rows-1; i++){
42                  #pragma HLS pipeline II=2
43                  row_indices[i] = in_stream.read();
44          }
45
46          cint32_t in[MAX_L-1-0 + 1];
47          for ( i = 0; i <= L-1; i++){
48                  #pragma HLS pipeline II=2
49                  int foo_real = in_stream.read();
50                  int foo_imag = in_stream.read();
51                  in[i].real = *(int32_t*)&foo_real;
52                  in[i].imag = *(int32_t*)&foo_imag;
53          }
54
55
56
57          cint64_t out[MAX_ROWS-1-0 + 1][MAX_COLS-1-0 + 1];
58
59
60      /* Call funtion */
61      covariance(in, row_indices, out, L, M, Cols, Rows, ColsDivM);
62
63      /* Output */
64          for ( i = 0; i <= Rows-1; i++){
65                  for ( j = 0; j <= Cols-1; j++){
```

```
66                          #pragma HLS pipeline II=2
67                          int64_t foo_real = out[i][j].real;
68                          int64_t foo_imag = out[i][j].imag;
69                          out_stream.write(foo_real);
70                          out_stream.write(foo_real >> 32);
71                          out_stream.write(foo_imag);
72                          out_stream.write(foo_imag >> 32);
73                  }
74          }
75
76  }
```

Resultado

En la figura 6.2 se muestra la comprobación del funcionamiento del *bitstream* para covariance. El código que corre en el ARM ejecuta el covariance en el mismo ARM para tener los resultados de referencia y posteriormente los comprueba con el resultado que recibe del covariance realizado en la FPGA. Como se observa en la última línea el resultado ha sido satisfactorio.

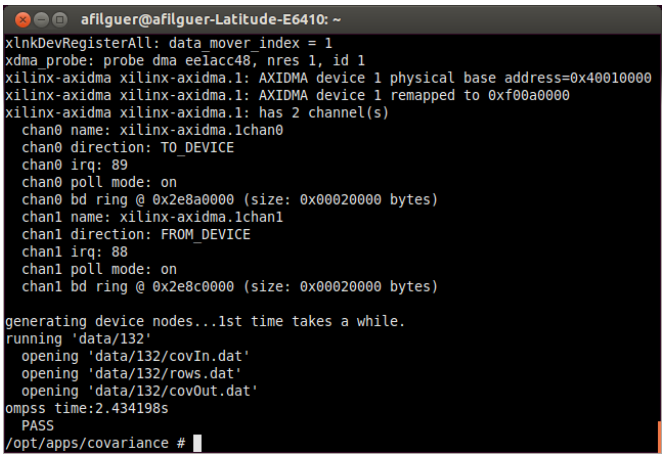

Figura 6.2: Resultado Covariance - Comprobación funcional del *bitstream* generado para Covariance

6.2.3 Cholesky

La aplicación implementa el problema de factorización de Cholesky . La descomposición de una matriz positiva definida (hermítica) en el producto de una matriz triangular y su transpuesta conjugada. Es útil para soluciones numéricas eficientes y simulaciones de Monte Carlo.

Código

```
1  template bool cpbtrf<cpbtrf_dim,double> (std::complex<double> A[↩
       cpbtrf_dim][cpbtrf_dim], std::complex<double> L[cpbtrf_dim][↩
       cpbtrf_dim]);
2
3
4      template <typename T, int Dim>
5  static void read_lower(std::complex<double> *in, std::complex<T> ↩
       out[Dim][Dim])
6  {
7  #pragma HLS inline
8      for (int col = 0; col < Dim; ++col){
9          for (int row = col; row < Dim; ++row)
10         {
11 #pragma HLS pipeline II=4
12             //out[row][col] = deserialize<std::complex<T> >(in++);
13             out[row][col] = *in++;
14             if (col != row)
15                 out[col][row] = std::complex<T>(0,0);
16         }
17     }
18 }
19
20 #pragma omp target device(fpga) copy_in(A[0:cpbtrf_dim-1][0:↩
       cpbtrf_dim-1]) copy_out(L[0:cpbtrf_dim-1][0:cpbtrf_dim-1]) ↩
       hw_storage(A[0:cpbtrf_dim-1][0:cpbtrf_dim-1], L[0:cpbtrf_dim↩
       -1][0:cpbtrf_dim-1])
```

```
21 #pragma omp task in(A[0:cpbtrf_dim-1][0:cpbtrf_dim-1]) out(L[0:↩
       cpbtrf_dim-1][0:cpbtrf_dim-1])
22 void cpbtrf_hw(std::complex<double>  A[cpbtrf_dim][cpbtrf_dim], std↩
       ::complex<double> L[cpbtrf_dim][cpbtrf_dim])
23 {
24     cpbtrf<cpbtrf_dim,cpbtrf_type>(A,L);
25 }
```

***Kernel* generado**

```
1  /*
2   * Wrapper function to handle conetion between kernel accelerator ↩
       and system
3   *
4  */
5  #include <iostream>
6  #include <iomanip>
7  #include <fstream>
8  #include <cassert>
9  #include <vector>
10 #include <complex>
11 #include <cstdlib>
12 #include "/home/edu/PFC/codigos_entrada/cholesky/cpbtrf.h"
13 #include "/home/edu/PFC/codigos_entrada/cholesky/cholesky_ompss.cpp↩
       "
14 #include "/home/edu/PFC/codigos_entrada/cholesky/timing.h"
15 #include <hls_stream.h>
16
17
18
19
20 void cpbtrf_hw_wrapper (hls::stream<int> &in_stream, hls::stream<↩
       int> &out_stream) {
21     #pragma HLS interface ap_ctrl_none port=return
22     #pragma HLS resource variable=in_stream core=AXI4Stream
```

```
23      #pragma HLS resource variable=out_stream core=AXI4Stream
24
25      /* Input */
26          int i;
27          int j;
28
29          std::complex<double> A[cpbtrf_dim-1-0 + 1][cpbtrf_dim-1-0 +↩
                1];
30          for ( i = 0; i <= cpbtrf_dim-1; i++){
31                  for ( j = 0; j <= cpbtrf_dim-1; j++){
32                          #pragma HLS pipeline II=2
33                          int read_real_2 = in_stream.read();
34                          int read_real_1 = in_stream.read();
35                          int read_imag_2 = in_stream.read();
36                          int read_imag_1 = in_stream.read();
37                          long long real, imag;
38                           real = read_real_1;
39                           real = (real << 32) + read_real_2;
40                           imag = read_imag_1;
41                           imag = (imag << 32) + read_imag_2;
42                           const double *double_real = ↩
                                 reinterpret_cast<const double *>(&real)↩
                                 ;
43                           const double *double_imag = ↩
                                 reinterpret_cast<const double *>(&imag)↩
                                 ;
44                           std::complex<double> complex(*double_real,↩
                                  *double_imag);
45                           A[i][j] = complex;
46                  }
47          }
48
49
50
51          std::complex<double> L[cpbtrf_dim-1-0 + 1][cpbtrf_dim-1-0 +↩
                1];
```

```
52
53
54      /* Call funtion */
55      cpbtrf_hw(A, L);
56
57      /* Output */
58          for ( i = 0; i <= cpbtrf_dim-1; i++){
59                  for ( j = 0; j <= cpbtrf_dim-1; j++){
60                          #pragma HLS pipeline II=2
61                          double real = L[i][j].real();
62                          double imag = L[i][j].imag();
63                          const long long *lreal = reinterpret_cast<↩
                                const long long *>(&real);
64                          const long long *limag = reinterpret_cast<↩
                                const long long *>(&imag);
65                          int ireal_1, ireal_2, iimag_1, iimag_2;
66                          ireal_1 = *lreal;
67                          ireal_2 = *lreal >> 32;
68                          iimag_1 = *limag;
69                          iimag_2 = *limag >> 32;
70                          out_stream.write(ireal_1);
71                          out_stream.write(ireal_2);
72                          out_stream.write(iimag_1);
73                          out_stream.write(iimag_2);
74                  }
75          }
76
77  }
```

Resultado

En la figura 6.3 se muestra la comprobación del funcionamiento del *bitstream* para cholesky. El código que corre en el ARM ejecuta el cholesky en el mismo ARM para tener los resultados de referencia y posteriormente los comprueba con el resultado que recibe del cholesky realizado en la FPGA. Como se observa en la última línea el resultado ha sido satisfactorio.

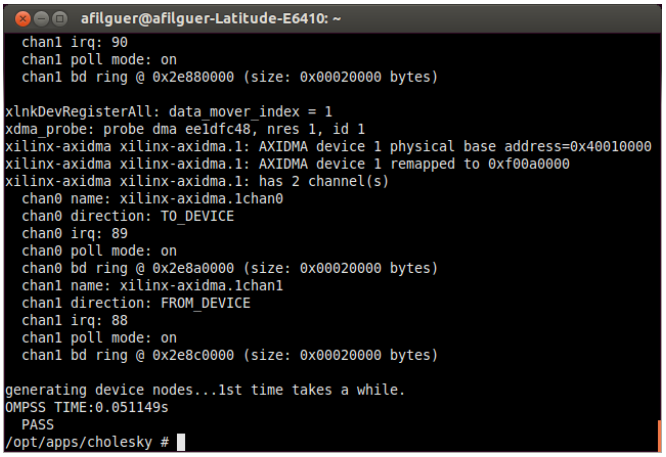

Figura 6.3: Resultado Cholesky - Comprobación funcional del *bitstream* generado para Cholesky

6.3 Tiempos de ejecución de GrizzlyHG

A continuación se muestran los tiempos reales de las ejecuciones de los ejemplos anteriores. Estas ejecuciones corresponden a los tiempos de compilación de los ejemplos anteriores mediante la plataforma GrizzlyHG.

El tiempo que utilizan las partes desarrolladas por nosotros es apenas inexistente comparado con el de las herramientas de sintetización e implementación de la Suite de Vivado. Así pues el tiempo añadido es apenas nulo mientras que el tiempo ahorrado de realizarlo manualmente es extremadamente grande. No se puede hablar de cifras en el caso de la generación manual dado que depende de la complejidad del sistema a construir y la experiencia del usuario.

Los tiempos que se muestran a continuación reflejan la media de 5 ejecuciones diferentes. Las especificaciones de la máquina utilizada se muestra en el anexo en la sección B. El ejemplo de MxM utilizado es de 32x32 elementos.

En la tabla 6.1 tenemos los tiempos para *cores* homogéneos, primera parte, y *cores* heterogéneos, segunda parte. Como bien se puede observar el tiempo de la Generación Hardware es mucho mayor que la Generación Kernel. También podemos ver como en el caso de Cholesky tanto la Generación Kernel como la Generación Hardware es mucho mayor que para los demás códigos. Esta diferencia se debe a su complejidad en el caso de la Generación Hardware por el alto porcentaje del hardware reconfigurable que ocupa. En el caso de la Generación Kernel esta

diferencia viene dada por la longitud de los *pragmas* a analizar utilizando expresiones regulares. Debido a las expresiones regulares utilizadas el tiempo crece con la longitud de manera cuadrática para las listas de identificadores.

	GrizzlyHG (s)	G. Kernel (s)	G. Hardware (s)
MxM: 1 core	2417.2581	0.0313	2417.2079
MxM: 2 cores	3090.9757	0.0315	3090.9442
Covariance: 1 core	2059.7585	0.0365	2059.7221
Covariance: 2 cores	2786.1909	0.0368	2786.1540
Cholesky: 1 core	3559.4818	22.9333	3536.5485
Cholesky: 2 cores	4720.8084	23.4787	4697.3296
MxM + Covariance	3675.5995	0.2768	3675.3226
MxM + Cholesky	5561.1903	24.6279	5536.5624
Covariance + Cholesky	5263.7569	25.5772	5238.1796

Tabla 6.1: Desglose de Tiempos Reales en GrizzlyHG

En la tabla 6.2 tenemos el desglose de la Generación Hardware. Como podemos observar lo que lleva más tiempo es la síntesis, implementación y generación del *bitstream* del sistema, es decir, la ejecución de PlanAhead. La segunda parte que más tiempo consume es la sintetización del *kernel*, CtoVHDL. Como se puede observar CtoVHDL y PlanAhead tardan más en función de la complejidad de la aplicación. En el caso de CtoVHDL en sistemas homogéneos no hay diferencia entre uno o dos *cores* dado que este proceso solo se realiza una vez. En el caso de sistemas heterogéneos se aplica primero a un *kernel* y después al otro, dado que son distintos. La diferencia entre *cores* viene dada por el porcentaje del hardware reconfigurable que utilizan. La generación del archivo MHS no varía entre diferentes sistemas homogéneos ni tampoco entre diferentes sistemas heterogéneos dado que no depende directamente de la complejidad del código. Por ultimo, bit2bin siempre tiene el mismo tiempo dado que es totalmente independientemente.

En las figuras 6.4 y 6.5 podemos ver de manera gráfica la información de las tablas. El tiempo de la plataforma viene dado por la Generación Hardware, principalmente de PlanAhead en primer lugar y CtoVHDL en segundo lugar.

En la figura 6.6 se puede ver gráficamente como afecta el uso de expresiones regulares para *parsear* el código en la Generación Kernel. En el caso de sistemas

	CtoVHDL (s)	G. MHS (s)	PlanAhead (s)	bit2bin (s)
MxM: 1 core	444.0166	0.0201	1970.8870	0.1091
MxM: 2 cores	412.8650	0.0196	2677.5081	0.0989
Covariance: 1 core	274.0973	0.0203	1784.9329	0.0990
Covariance: 2 cores	271.8784	0.0154	2513.6433	0.0995
Cholesky: 1 core	982.1113	0.0239	2553.8608	0.0992
Cholesky: 2 cores	983.9892	0.0124	3712.8145	0.1001
MxM Covariance	826.1751	0.0570	2823.2997	0.1020
MxM Cholesky	1549.8648	0.0300	3986.1462	0.1172
Covariance Cholesky	1425.4467	0.0316	3812.3231	0.1083

Tabla 6.2: Desglose de Tiempos Reales de G. Hardware

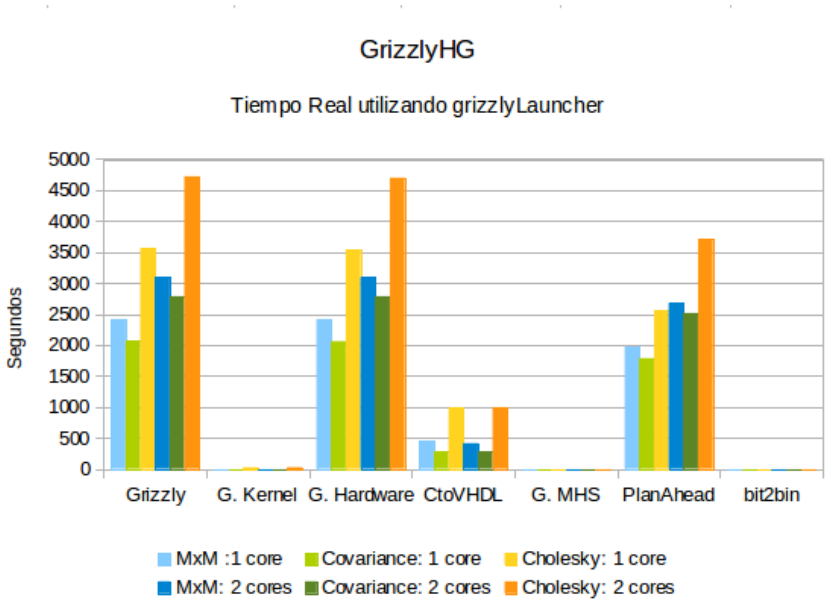

Figura 6.4: Gráfico: Desglose de tiempos reales en GrizzlyHG - Utilizando *cores* homogéneos de 1 y 2 cores

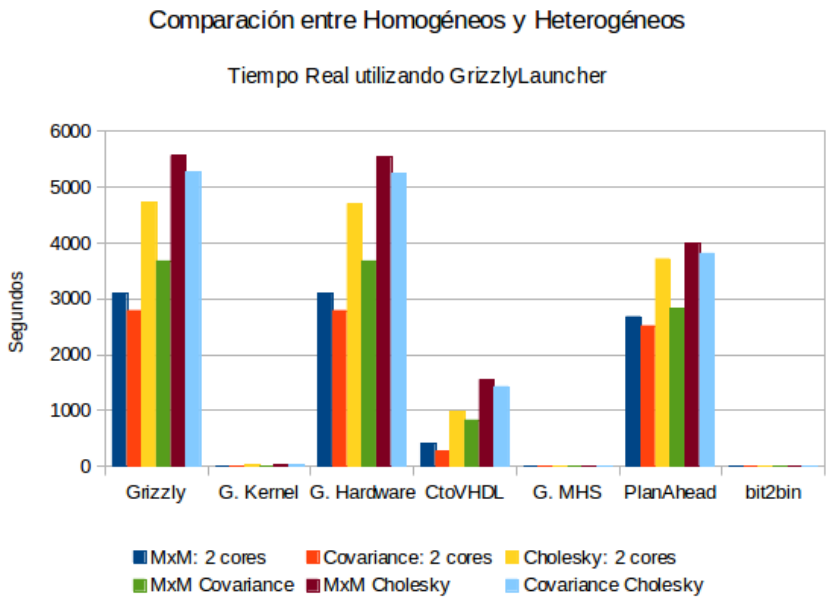

Figura 6.5: Gráfico: Desglose de tiempos reales en GrizzlyHG - Utilizando *cores* heterogéneos de 2 cores en total

donde interviene cholesky el tiempo es significantemente mayor debido a la longitud de la cadena a tratar con expresiones regulares. Si bien se podría reducir este tiempo aplicando otras medidas vemos que la mejora a conseguir no tendría un gran impacto en el tiempo total de la plataforma.

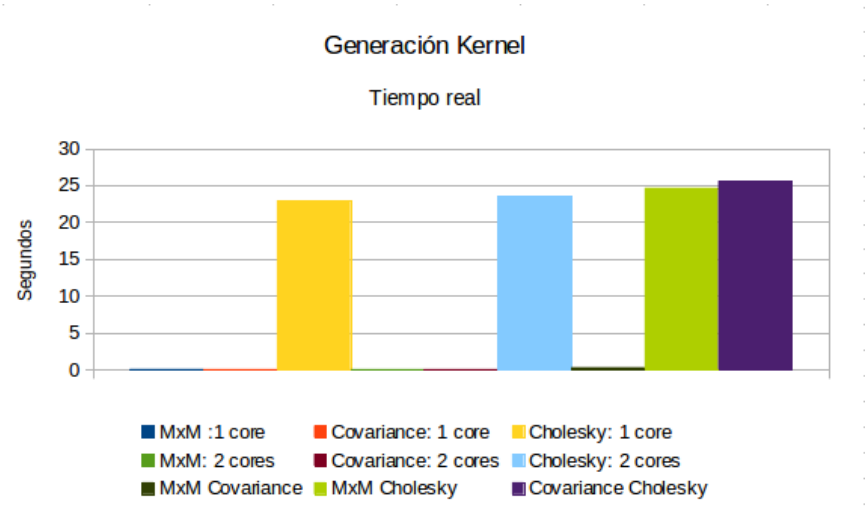

Figura 6.6: Gráfico: Desglose de tiempos reales en Generación Kernel -

7

Conclusiones

Con este proyecto se ha logrado construir una plataforma capaz de automatizar la generación hardware. Así pues el usuario puede generar el *bitstream* a partir de sus códigos en C/C++ tan solo añadiendo dos *pragmas* OmpSs.

Con el proyecto del BSC el usuario obtiene, además del *bitstream*, el binario para los ARM, con lo que puede acelerar sus códigos en las máquinas Zynq de manera totalmente automática. Esto conlleva las ventajas de que no se requieren conocimientos sobre hardware reconfigurable ni tener que dedicar tiempo y esfuerzo en generarlo.

La plataforma es un primer prototipo funcional que permite la generación automática de hardware a partir de un código en alto nivel, integrándose dentro de otro proyecto más grande que permite la aceleración de aplicaciones utilizando la FPGA de una Zynq. Esto lo demuestran las pruebas realizadas para varios códigos y varias plataformas hardware basadas en Zynq.

Por todo ello, podemos decir que los objetivos del proyecto se han cumplido y los resultados han sido importantes y positivos. Inicialmente el proyecto sólo contemplaba la automatización de la generación del hardware. No obstante, una vez acabada la generación del hardware se decidió extender el proyecto y añadir la generación del *kernel* para completar la automatización.

Actualmente no existe nada parecido en automatización de éste proceso. Esto nos ha llevado a que hayamos podido participar en dos publicaciones internacionales:

1. Filgueras, A., **Gil, Eduard**, Jiménez-González, D., Alvarez, C., Martorell, X., Langer, J., Noguera, J. & Vissers, K. "*OmpsSs@Zynq All-Programmable SoC Ecosystem*", 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2014)

2. Filgueras, A., **Gil, Eduard**, Alvarez, C., Jiménez-González, D., Martorell, X., Langer, J. & Noguera, J. "*Heterogeneous tasking on SMP/FPGA SoCs: the case of OmpSs and the Zynq*", workshop @VLSI-SOC: W1A(Special Session2): Are processors the NAND gats of the future? (2013)

Para la realización de este proyecto han sido útiles todos los conocimientos adquiridos a lo largo de la carrera pero cabe destacar algunas asignaturas.

Compiladores, CL Entender el funcionamiento de Mercurium

Programación Consciente de la Arquitectura, PCA Uso de punteros y conversiones de tipo. Escribir *shell scripts*.

Estructura de Computadores 2, EC2 Organización de datos en memoria. Teoría de punteros.

Administración de Sistemas Operativos, ASO Escribir *shell scripts*. Solución de errores del sistema Linux para el correcto funcionamiento de los *drivers* JTAG.

Planificación y Gestión de Proyectos y Sistemas Informáticos, PGPSI Organización y planificación del proyecto.

Proyecto de Redes de Computadores, PXC Organización del proyecto.

Arquitectura de Computadores, AC Entender como funciona el *SoC* y trabajar a bajo nivel.

Por último, mencionar que el hacer este proyecto me ha brindado la oportunidad de realizar un *internship* en Xilinx Ireland Inc.

7.1 Trabajo Futuro

La generación del *kernel* se habría implementado utilizado Mercurium de haber sabido si se disponía del suficiente tiempo. Decidimos desarrollar nosotros mismo una aplicación dado que no teníamos experiencia en Mercurium y la integración con la Generación Hardware sería más fácil.

No obstante realizar la generación del *kernel* en Mercurium aporta las ventajas de que Mercurium utiliza un *parser* para extraer los *tokens* del código y organiza un árbol para que sea más simple trabajar con la información. Si la sintaxis de OmpSs cambiase sería muy sencillo adaptarlo, cosa que en nuestra herramienta es más complejo. Además la robustez de Mercurium es mucho mayor y provee de las herramientas necesarias. Con el prototipo de la Generación Kernel hemos visto que realizarlo es posible y viable. Con todo ello queremos indicar que el no realizarlo con Mercurium ha sido por limitación de tiempo, al no estar previsto en los objetivos iniciales del pryecto la generación del kernel.

8

Planificación Temporal

El inicio del proyecto se sitúa en el 18/02/2013 y finaliza el 10/01/2014. Tiene una duración de 327 días (225 laborables), aproximadamente 11 meses. Se han dedicado 1208 horas repartidas en 169 días para su realización (ver Tabla 8.1).

Bloque	Horas	Días
Acercamiento	140	18
Desarrollo de la Plataforma	888	111
Reporte	180	40
Total	**1208**	**169**

Tabla 8.1

La planificación se divide en tres grandes bloques:

Acercamiento Pre-requisito para obtener el conocimiento previo necesario.

Desarrollo de la Plataforma Análisis, Construcción y Verificación de la plataforma.

Reporte Elaboración de la memoria y de la presentación del trabajo realizado.

Para el *Desarrollo de la Plataforma* se dedican un total de 888 horas repartidas en 111 días (ver Tabla 8.2).

8.1 Diagrama de Gant

En el gráfico 8.1 tenemos la planificación del *Acercamiento* y la *Generación Hardware*. En el gráfico 8.2 tenemos la planificación la *Generación Kernel*, la *Inte-*

Etapa	Horas	Días
Generación Hardware	408	51
Generación Kernel	416	52
Integración	64	8
Total	**888**	**111**

Tabla 8.2

gración y el *Reporte* (Redactar Memoria y Elaboración Presentación).

El soporte para *cores* heterogéneos era secundario y por ello se realiza una vez realizados los objetivos principales.

ID	Task Name	Duration	Start	Finish
1	**Acercamiento**	**140 hrs**	**Mon 2/18/13**	**Wed 3/13/13**
2	Test SoC Zedboard	1 day	Mon 2/18/13	Mon 2/18/13
3	Instalación Suite 14.2	2 days	Tue 2/19/13	Wed 2/20/13
4	Generar bitstream manualmente	10 days	Thu 2/21/13	Wed 3/6/13
5	Configurar SDK	2 days	Thu 3/7/13	Fri 3/8/13
6	Instalación Suite 14.3	2 days	Mon 3/11/13	Tue 3/12/13
7	Hello World Standalone	1 hr	Wed 3/13/13	Wed 3/13/13
8	Matrix Mult Standalone	3 hrs	Wed 3/13/13	Wed 3/13/13
9	**Generación hardware**	**408 hrs**	**Fri 4/5/13**	**Fri 6/14/13**
10	Configuración Conexiones	10 days	Fri 4/5/13	Thu 4/18/13
11	Compilación CtoVHDL del core	1 day	Fri 4/19/13	Fri 4/19/13
12	Generación Bitstream	3 days	Mon 4/22/13	Wed 4/24/13
13	Portar a Zynq 702	5 days	Thu 4/25/13	Wed 5/1/13
14	Soporte para otras aplicaciones	15 days	Mon 6/3/13	Fri 6/21/13
15	**Cores homogéneos**	**96 hrs**	**Thu 5/16/13**	**Fri 5/31/13**
16	Generar Bitstream manualmente	7 days	Thu 5/16/13	Fri 5/24/13
17	Automatización	5 days	Mon 5/27/13	Fri 5/31/13
18	Cores heterogéneos	5 days	Mon 11/11/13	Fri 11/15/13

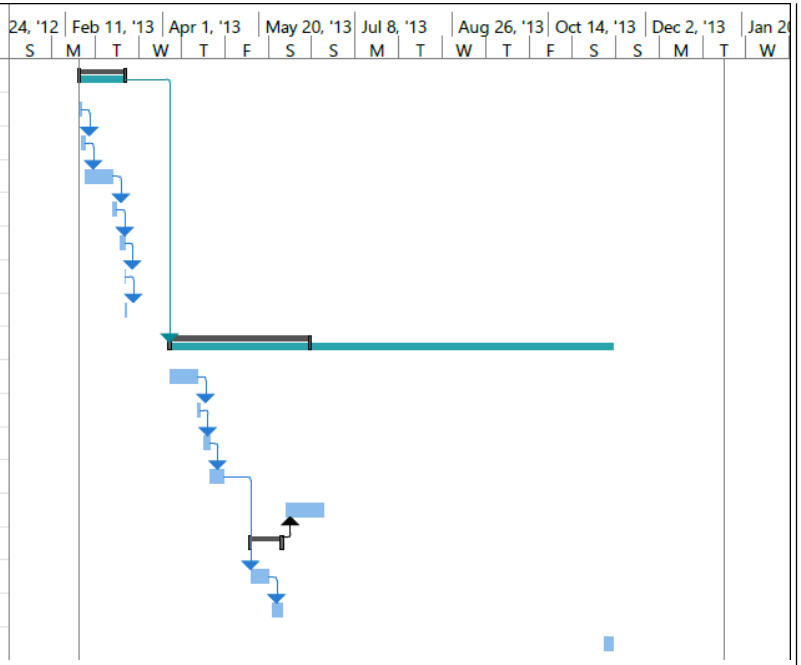

Figura 8.1: Diagrama Gant 1/2 - Acercamiento, Generación Hardware.

ID	Task Name	Duration	Start	Finish
23	**Generación kernel**	**416 hrs**	**Mon 8/19/13**	**Tue 10/29/13**
24	Evaluación Mercurium	10 days	Mon 8/19/13	Fri 8/30/13
25	Generación Kernel Base	30 days	Mon 9/2/13	Fri 10/11/13
26	Ampliación para Matrix Mult	1 day	Mon 10/14/13	Mon 10/14/13
27	Ampliación para Covariance	7 days	Tue 10/15/13	Wed 10/23/13
28	Ampliación para Cholesky	4 days	Thu 10/24/13	Tue 10/29/13
29	**Integración**	**64 hrs**	**Wed 10/30/13**	**Fri 11/8/13**
30	Adaptar input Generación Hardware	2 days	Wed 10/30/13	Thu 10/31/13
31	Unión Generación Hardware y Kernel	3 days	Fri 11/1/13	Tue 11/5/13
32	Integración con proyecto BSC	3 days	Wed 11/6/13	Fri 11/8/13
33	Redactar Memoria	140 hrs	Mon 11/18/13	Fri 1/3/14
34	Elaboración Presentación	5 days	Mon 1/6/14	Fri 1/10/14

Figura 8.2: Diagrama Gant 2/2 - Generación Kernel, Integración, Reporte.

8.2 Recursos

Para llevar a cabo las tareas se requieren dos perfiles, el de analista y el de programador. En las tareas planificadas se requiere ambos perfiles y por lo tanto no se puede asignar una tarea a un solo perfil. Dado que el carácter de este proyecto es de investigación se estima que el peso del analista es de un 70% (621,6 horas) y el del programador el 30% (266,4) en las tareas de desarrollo de la plataforma. El acercamiento únicamente es necesario para el analista, para el programador no se requiere.

Para nuestra planificación suponemos que el analista requiere el acercamiento. También está la posibilidad de que el analista ya tuviera los conocimientos necesarios, en ese caso hay que descontar las 140 horas correspondientes contempladas en la tabla.

El analista dedica un total de 761,6 horas mientras que el programador dedica un total de 266,4 horas (ver Tabla 8.3).

Perfil	Horas
Analista	761,6
Programador	266,4
Total	**1028**

Tabla 8.3

9

Costes económicos

Los costes económicos se dividen en:

- Costes Humanos
- Costes Hardware
- Costes Software
- Otros Costes

Los *SoC* y las licencias de software privativo de Xilinx han sido donados por Xilinx Inc. a la universidad. En el cálculo de costes se le aplica la cantidad de 0€.

Costes Humanos

Para la realización del proyecto se requieren dos perfiles:

- Analista
- Programador

El coste del analista se ha calculado según un sueldo medio de 60.000€ anuales (1826 horas), más el coste de la seguridad social (30%). Se encarga del 70% del peso de las tareas y del acercamiento.

El coste del programador se ha calculado según un sueldo medio de 50.000€ anuales (1826 horas), más el coste de la seguridad social (30%). Se encarga del 30% del peso de las tareas.

El total de los costes humanos es de **42.015,77€** (ver Tabla 9.1).

Concepto	Coste unitario (€/h)	Coste bruto (€)	Coste SS (€)	Coste total (€)
Analista	32,86	25.025,19	7.507,56	32.532,75
Programador	27,38	7.294,63	2.188,39	9.483,02
Total	-	32.319,82	9.696,35	**42.015,77**

Tabla 9.1: Costes humanos

Costes Hardware

Se estima que el hardware tiene una vida útil de 3 años, se les calcula la parte imputable correspondiente a la duración del proyecto.

La *ZC702* y la *Zedboard* han sido donada por Xilinx Inc. Cada uno de los *SoC* están valorados en 895$[26] y 395$[27] respectivamente.

El coste del hardware es de **279,28€** (ver Tabla 9.2).

Concepto	Coste (€)	Coste imputable (€)
Portátil	914	279,28
Zedboard	0	0
ZC702	0	0
Total	914	**279,28**

Tabla 9.2: Costes Hardware

Costes Software

El software utilizado ha sido la *Suite Vivado System Edition* de Xilinx Inc. y software libre.

El precio de las licencias software donadas por Xilinx Inc. está valorado en 4.795$[28].

El coste total del software es de **0€** (ver Tabla 9.3).

Concepto	Coste (€)	Coste imputable (€)
Software Libre	0	0
Licencia Vivado & ISE Suite	0	0
Total	0	**0**

Tabla 9.3: Costes Software

Otros Costes

Otros costes a tener en cuenta ha sido el de Internet. A lo largo de todo el proyecto Internet ha sido una herramienta indispensable. El coste del Internet puede variar dependiendo de la compañía proveedora de Internet utilizada. En nuestro caso en particular es el reflejado en la tabla.

El coste total de otros costes es de **566.61€** (ver Tabla 9.4).

Concepto	Coste (€)	Coste imputable (€)
Internet	51,51/mes	566.61
Total	51,51/mes	**566.61**

Tabla 9.4: Otros Costes

Costes Totales

El coste total del proyecto asciende a **42.861,66€** (ver Tabla 9.5).

Concepto	Coste (€)
Costes Humanos	42.015,77
Costes Hardware	279,28
Costes Software	0
Otros Costes	566.61
Coste Total	**42.861,66**

Tabla 9.5: Coste Total

References

[1] **OpenMP**. *http://openmp.org/wp/*, Marzo 2013. Citado en página 1.

[2] **Zedboard Reference Designs**. *http://zedboard.org/design/1521/11*, Marzo 2013. Citado en página 7.

[3] **Zedboard Documentation**. *https://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD*, Marzo 2013. Citado en página 7.

[4] **Vivado Suite Installation and Licensing**. *http://www.xilinx.com/support/documentation/sw_manuals/*, Marzo 2013. Citado en página 7.

[5] **Adept JTAG Drivers Documentation**. *http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT*, Marzo 2013. Citado en página 7.

[6] **Zedboard SDK Example**. *http://zedboard.org/content/zedboard-sdk-helloworld-example*, Marzo 2013. Citado en página 7.

[7] **Xilinx Command Line Tools User Guide.** *http://www.xilinx.com/support/documentation/sw_manuals/*, Abril 2013. Citado en página 8.

[8] **Vivado Design Suite Tcl Command Reference Guide**. *http://www.xilinx.com/support/documentation/sw_manuals/*, Abril 2013. Citado en página 8.

[9] **Vivado Design Suite User Guide**. *http://www.xilinx.com/support/documentation/sw_manuals/*, Abril 2013. Citado en página 8.

[10] **Eclipse**. *http://www.eclipse.org*, Marzo 2013. Citado en página 10.

[11] **PyDev**. *http://pydev.org*, Abril 2013. Citado en página 10.

[12] **Git**. *http://www.git-scm.com*, Agosto 2013. Citado en página 10.

[13] **Bitbucket DVCS**. *https://bitbucket.org*, Septiembre 2013. Citado en página 10.

[14] **Python**. *http://www.python.org*, Noviembre 2013. Citado en página 10.

[15] **Python2 Documentation**. *http://docs.python.org/2/*, Noviembre 2013. Citado en página 10.

[16] **Cplusplus**. *http://www.cplusplus.com*, Noviembre 2013. Citado en página 10.

[17] **IEEE Xplore Digital Library**. *http://ieeexplore.ieee.org*, Noviembre 2013. Citado en página 10.

[18] **ACM Digital Library**. *http://dl.acm.org/*, Noviembre 2013. Citado en página 10.

[19] **Google Scholar**. *http://scholar.google.com*, Noviembre 2013. Citado en página 10.

[20] **Barcelona Supercomputing Center. Ompss**. *http://pm.bsc.es/ompss*, Noviembre 2013. Citado en página 11.

[21] **CUDA. Nvidia Developer Zone**. *http://developer.nvidia.com/cuda*, Marzo 2013. Citado en página 11.

[22] **OpenCL. Khronos Group**. *http://www.khronos.org/opencl/*, Marzo 2013. Citado en página 11.

[23] **Barcelona Supercomputing Center. Mercurium**. *http://pm.bsc.es/mcxx*, Agosto 2013. Citado en página 12.

[24] **Barcelona Supercomputing Center. Mercurium**. *https://pm.bsc.es/projects/mcxx*, Agosto 2013. Citado en página 12.

[25] **Xilinx. Zynq-7000 All Programmable SoC**. *http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/*, Abril 2013. Citado en página 12.

[26] **ZC702 SoC Price**. *http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC702-G.htm*, Diciembre 2013. Citado en página 80.

[27] **Zedboard Price**. *http://www.zedboard.org/buy*, Diciembre 2013. Citado en página 80.

[28] **Vivado Design Tool Node-Locked License Price**. *http://www.xilinx.com/products/design-tools/vivado/index.htm*, Diciembre 2013. Citado en página 80.

Appendix A

Ejemplos JSON de empaquetado y desempaquetado de tipos complejos

Los ejemplos que se muestran a continuación los hemos creado para los tipos complejos que utilizan las aplicaciones mostradas en los resultados. Se puede observar el uso de la interfaz creada para ello.

A.1 cint32_t

```
{
    "type": "cint32_t",
    "structure": [{
        "type": "int32_t",
        "name": "real"
    }, {
        "type": "int32_t",
        "name": "imag"
    }],
    "scather": "
    {ident}{stream_in_type} foo_real = {stream_in_name}.read();\n
    {ident}{stream_in_type} foo_imag = {stream_in_name}.read();\n
    {ident}{name}{row}{col}.real = {castsc[0]}foo_real;\n
```

```
    {ident}{name}{row}{col}.imag = {castsc[1]}foo_imag;\n
    ",

    "gather": "
    {ident}int32_t foo_real = {name}{row}{col}.real;\n
    {ident}int32_t foo_imag = {name}{row}{col}.imag;\n
    {ident}{stream_out_name}.write({castga[0]}foo_real);\n
    {ident}{stream_out_name}.write({castga[1]}foo_imag);\n
    "
}
```

A.2 cint64_t

```
{
    "type": "cint64_t",
    "structure": [{
        "type": "int64_t",
        "name": "real"
    }, {
        "type": "int64_t",
        "name": "imag"
    }],
    "scather": "
    {ident}{stream_in_type} foo_real = {stream_in_name}.read();\n
    {ident}{stream_in_type} foo_real32 = {stream_in_name}.read();\n
    {ident}{stream_in_type} foo_imag = {stream_in_name}.read();\n
    {ident}{stream_in_type} foo_imag32 = {stream_in_name}.read();\n
    {ident}{name}{row}{col}.imag = {castsc[0]}&foo_imag;\n
    {ident}{name}{row}{col}.imag = ({name}{row}{col}.imag >> 32) + {
        castsc[0]}&foo_imag32;\n
    {ident}{name}{row}{col}.real = {castsc[1]}&foo_real;\n
    {ident}{name}{row}{col}.real = ({name}{row}{col}.real >> 32) + {
        castsc[1]}&foo_real32;\n
    ",

```

```
    "gather": "
    {ident}int64_t foo_real = {name}{row}{col}.real;\n
    {ident}int64_t foo_imag = {name}{row}{col}.imag;\n
    {ident}{stream_out_name}.write(foo_real);\n
    {ident}{stream_out_name}.write(foo_real >> 32);\n
    {ident}{stream_out_name}.write(foo_imag);\n
    {ident}{stream_out_name}.write(foo_imag >> 32);\n
    "
}
```

A.3 complex<double>

```
{
    "type": "std::complex<double>",
    "structure": [{
        "type": "double",
        "name": "real"
    }, {
        "type": "double",
        "name": "imag"
    }],
    "scather": "
    {ident}{stream_in_type} read_real_2 = {stream_in_name}.read();\n
    {ident}{stream_in_type} read_real_1 = {stream_in_name}.read();\n
    {ident}{stream_in_type} read_imag_2 = {stream_in_name}.read();\n
    {ident}{stream_in_type} read_imag_1 = {stream_in_name}.read();\n
    {ident}long long real, imag;\n
    {ident} real = read_real_1;\n
    {ident} real = (real << 32) + read_real_2;\n
    {ident} imag = read_imag_1;\n
    {ident} imag = (imag << 32) + read_imag_2;\n
    {ident} const double *double_real = reinterpret_cast<const double
        *>(&real);\n
    {ident} const double *double_imag = reinterpret_cast<const double
        *>(&imag);\n
```

```
22      {ident} std::complex<double> complex(*double_real, *double_imag)
            ;\n
23      {ident} {name}{row}{col} = complex;\n
24      ",
25
26      "gather": "
27      {ident}double real = {name}{row}{col}.real();\n
28      {ident}double imag = {name}{row}{col}.imag();\n
29      {ident}const long long *lreal = reinterpret_cast<const long long
            *>(&real);\n
30      {ident}const long long *limag = reinterpret_cast<const long long
            *>(&imag);\n
31      {ident}int ireal_1, ireal_2, iimag_1, iimag_2;\n
32      {ident}ireal_1 = *lreal;\n
33      {ident}ireal_2 = *lreal >> 32;\n
34      {ident}iimag_1 = *limag;\n
35      {ident}iimag_2 = *limag >> 32;\n
36      {ident}{stream_out_name}.write(ireal_1);\n
37      {ident}{stream_out_name}.write(ireal_2);\n
38      {ident}{stream_out_name}.write(iimag_1);\n
39      {ident}{stream_out_name}.write(iimag_2);\n
40      "
41  }
```

Appendix B

Especificaciones del portátil

A continuación se muestran las especificaciones del portátil utilizado para realizar este proyecto.

B.1 cpuinfo

```
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 58
model name  : Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
stepping    : 9
microcode   : 0x15
cpu MHz     : 1200.000
cache size  : 6144 KB
physical id : 0
siblings    : 8
core id     : 0
cpu cores   : 4
apicid      : 0
initial apicid  : 0
fpu     : yes
fpu_exception   : yes
cpuid level : 13
wp      : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
    pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
```

```
    rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
    nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2
    ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer
     aes xsave avx f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm
    tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms
bogomips    : 4390.14
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Hay 7 más igual que este pero solo incluimos el primero por razones de espacio.

B.2 meminfo

```
MemTotal:       16306168 kB
MemFree:         7044260 kB
Buffers:          560196 kB
Cached:          6645756 kB
SwapCached:            0 kB
Active:          4659996 kB
Inactive:        3934720 kB
Active(anon):    1555456 kB
Inactive(anon):   158644 kB
Active(file):    3104540 kB
Inactive(file):  3776076 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       3905532 kB
SwapFree:        3905532 kB
Dirty:               120 kB
Writeback:             0 kB
AnonPages:       1388756 kB
Mapped:           260008 kB
Shmem:            325344 kB
Slab:             433252 kB
SReclaimable:     317552 kB
SUnreclaim:       115700 kB
KernelStack:        4128 kB
PageTables:        29524 kB
```

```
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    12058616 kB
Committed_AS:    4525756 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      384460 kB
VmallocChunk:   34359267760 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:       67584 kB
DirectMap2M:    16601088 kB
```