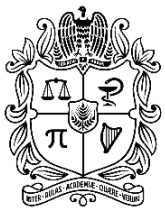


UNIVERSIDAD
NACIONAL
DE COLOMBIA

Optimización de algoritmos para computación científica sobre arquitecturas heterogéneas

Luis Fernando Castaño Londoño

Universidad Nacional de Colombia
Facultad de Ingeniería y Arquitectura
Departamento de Ingenierías Eléctrica, Electrónica y Computación
Manizales, Colombia
Septiembre 2021

Optimización de algoritmos para computación científica sobre arquitecturas heterogéneas.

Luis Fernando Castaño Londoño

Tesis o trabajo de grado presentada(o) como requisito parcial para optar al título de:
Doctor en Ingeniería - Automática

Director:
Ph.D. Gustavo Osorio

Línea de Investigación:
Diseño electrónico
Grupo de Investigación:
Percepción y Control Inteligente

Universidad Nacional de Colombia
Facultad de Ingeniería y Arquitectura
Departamento de Ingenierías Eléctrica, Electrónica y Computación
Manizales, Colombia
Septiembre 2021

Agradecimientos

Agradezco a Sandra, Lucía y mi familia en general por todo el amor, apoyo y paciencia. Agradezco a mi director de tesis Gustavo Osorio Londoño por el apoyo, paciencia y sus valiosos aportes. Agradezco a los docentes y compañeros del grupo de trabajo acadÃ©mico en Percepción y Control Inteligente por sus valiosos aportes. Agradezco a mis compañeros del Laboratorio de Sistemas de Control y Robótica por sus valiosos aportes. Agradezco al programa de Becas Estudiante Sobresalientes de Postgrado de la Universidad Nacional de Colombia por el apoyo recibido a través de la beca otorgada.

Resumen

Optimización de algoritmos para computación científica sobre arquitecturas heterogéneas

Un esquema muy usado en la computación científica se conoce como computación con esténcil. Es el núcleo central de algoritmos de álgebra lineal, ecuaciones diferenciales parciales (EDP) y procesamiento de imágenes. Sin embargo, el desempeño de los algoritmos basados en esténcil, está limitado por la notable diferencia entre el máximo rendimiento de procesamiento y el máximo ancho de banda de memoria en los sistemas multinúcleo y unidades de computación gráfica (GPU). Por esta razón el estudio de métodos para su optimización ha sido de gran interés. Algunos métodos se basan en la optimización del empleo de memoria, sobre los cuales se han desarrollado diversos trabajos en sistemas basados en CPU y arquitecturas heterogéneas. Debido a que con estos métodos de optimización persisten limitaciones en el rendimiento, algunos autores han propuesto esquemas para sistemas basados en arreglos de compuertas programables en campo (FPGA). En esta tesis doctoral se presentan dos metodologías para la optimización de arquitecturas basadas en FPGA para la computación con esténcil. Para algunas arquitecturas el diseño se realiza a nivel de hardware con base en el modelo de Glushkov utilizando VHDL. En otros casos se realiza codiseño hardware/software utilizando herramientas de síntesis de alto nivel. Como casos de estudio se propone la implementación y evaluación de rendimiento de una arquitectura basada en esténcil para la aproximación a la solución de problemas de propagación de calor modelados con la ecuación de calor unidimensional y la ecuación de Laplace bidimensional. Se proponen transformaciones en las arquitecturas y códigos basados en esténcil para el mejoramiento del desempeño en la ejecución del algoritmo con relación a una implementación base. En el caso de implementación con la herramimenta de síntesis de alto nivel se definen parámetros asociados al tamaño del dominio de la solución y directivas de optimización, para la determinación del efecto en el desempeño de la ejecución del algoritmo basado en computación con esténcil. Se define una superficie de respuesta para determinar los valores óptimos de los parámetros con los cuales se obtiene la menor latencia para la implementación propuesta.

Palabras clave: computación heterogénea, FPGA, computación con esténcil, ecuación de calor, ecuación de Laplace, síntesis de alto nivel.

Abstract

Algorithm optimization for scientific computing on heterogeneous architectures

A scheme widely used in scientific computing is known as stencil computation. It is the central kernel of linear algebra algorithms, partial differential equations (PDE) and image processing. However, the performance of stencil-based algorithms is limited by the remarkable difference between maximum throughput and maximum memory bandwidth in multi-core systems and graphics computing units (GPUs). For this reason the study of methods for its optimization has been of great interest. Some methods are based on optimizing the use of memory, on which various jobs have been developed in CPU-based systems and heterogeneous architectures. Because these optimization methods persist with performance limitations, some authors have proposed schemes for systems based on programmable field gate arrays (FPGA). In this thesis, two methodologies for the optimization of FPGA-based architectures for stencil computing are presented. For some architectures the design is done at the hardware level based on the Glushkov model using VHDL. In other cases, hardware/software co-design is carried out using high-level synthesis tools. As a case study, the implementation and performance evaluation of a stencil-based architecture is proposed for the approximation to the solution of heat propagation problems modeled with the one-dimensional heat equation and the two-dimensional Laplace equation.

Keywords: heterogeneous computing, FPGA, stencil computation, heat equation, Laplace equation, high level synthesis.

Contenido

1. Introducción

El desarrollo de sistemas basados en FPGA ha tenido transformaciones muy importantes en los últimos años. Los retos en la implementación de diferentes tipos de algoritmos y aplicaciones llevaron al surgimiento de herramientas y estudios orientados al mejoramiento del flujo de diseño. Se encuentra una gran variedad de literatura que trata aspectos relacionados con los formatos de representación numérica, la complejidad de los diferentes niveles de abstracción, y el empleo de herramientas de desarrollo para facilitar el flujo de diseño.

En esta tesis se realiza una exploración de algunos de estos aspectos a través del diseño e implementación de diferentes arquitecturas para la optimización de algoritmos de computación con esténcil sobre sistemas heterogéneos basados en FPGA. Los aceleradores basados en FPGA han sido usados como alternativa a los sistemas de cómputo basados en CPU o GPU, dado que estos dispositivos han demostrado ofrecer mejor desempeño con menor consumo de energía [47, 56]. Estos sistemas toman ventaja del paralelismo inherente a través de la combinación de diversas técnicas de optimización del flujo de datos.

La determinación del tipo de representación numérica para los datos ha sido un aspecto de diseño de gran peso en el desarrollo de aplicaciones sobre sistemas basados en FPGA. Dentro de los temas abordados en los trabajos encontrados en la literatura se encuentran el uso de formatos de representación numérica y la implementación de operaciones aritméticas sobre FPGA como en [25, 8, 5, 24, 2, 62]. Una visión amplia de la aritmética computacional es desarrollada en [14], donde se presenta una descripción completa de aspectos teóricos y prácticos para la implementación de circuitos aritméticos sobre sistemas embebidos.

Con base en una serie de trabajos sobre la implementación de operaciones empleando formatos de representación estándar, el empleo de formatos de representación personalizados estuvo justificado por la flexibilidad del diseño e implementación de circuitos aritméticos sobre FPGA. Esto fue soportado por la evaluación del rendimiento y la comparación de resultados numéricos en términos de precisión empleando diferentes formatos. En [51] se presenta la evaluación del desempeño de tres tipos de FPGAs para formatos de punto flotante de 64, 32 y 24 bits. Por otra parte, el desarrollo de módulos de punto flotante parametrizados se encuentran en trabajos como los presentados en [3]. En este trabajo se hace relación al uso de mixto de

formatos o la variación de la precisión dependiendo de los requerimientos de los algoritmos. De esta forma se plantea que el mejoramiento del rendimiento en la ejecución de algoritmos, puede lograrse con el empleo de formatos de punto fijo o formatos flexibles de punto flotante. Trabajos similares pueden encontrarse en [19, 57, 22, 17, 15, 49, 41, 55, 16, 23].

Algunos autores evidenciaron en sus trabajos una tendencia creciente hacia el empleo del formato de punto flotante como en [15]. La consolidación de la computación heterogénea y la incursión definitiva de los FPGA en el campo de la computación científica, ha llevado hacia el empleo de formatos de punto flotante estándar compatibles con otros tipos de sistema de cómputo. De esta forma se cuenta actualmente con una gran variedad de librerías, núcleos de propiedad intelectual y herramientas que permiten la implementación de sistemas basados en FPGA, en las que se facilita el empleo de diferentes formatos de representación numérica y la aritmética computacional. Algunas de estas herramientas fueron desarrolladas para el mejoramiento del flujo de diseño, en la medida que los principales fabricantes de FPGA han enfocado el mercado tanto en términos de desarrollo de hardware como de software.

Los aceleración de algoritmos basados en esténcil utilizando FPGA, ha sido realizada tanto con técnicas de diseño convencional de lógica de transferencia entre registros (RTL) como con herramientas de diseño de alto nivel (HLS). Arquitecturas basadas en arreglos como las desarrolladas en [43, 26, 44], implican la necesidad de un número considerable de FPGA para la simulación de problemas de tamaños que pueden ser tratados sobre CPU o GPU con un desempeño aceptable a un menor costo. El uso de herramientas de diseño de alto nivel ha permitido la superación de algunos retos relacionados con la representación numérica y la complejidad del flujo de diseño. En [47] Schmitt *et al.* demuestran la posibilidad de emplear un esquema de computación con esténcil para una malla de 4096x4096 en una sola FPGA.

En esta tesis se presenta el diseño, implementación y evaluación de desempeño de diferentes arquitecturas para la ejecución en FPGA de algoritmos basados en esténcil empleando el esquema de diferencias finitas, desarrolladas con técnicas de diseño RTL y con herramientas de síntesis de alto nivel (HLS). Las contribuciones de la tesis son las siguientes:

1. Se analiza el desempeño en la ejecución de un código basado en esténcil implementado en una arquitectura heterogénea basada en FPGA utilizando diseño convencional RTL y una herramienta de síntesis de alto nivel.

2. Se proponen transformaciones en las arquitecturas y códigos basados en esténcil para el mejoramiento del desempeño en la ejecución del algoritmo con relación a una implementación base.

3. Para el caso de implementación con la herramimenta de síntesis de alto nivel se definen parámetros asociados al tamaño del dominio de la solución y directivas de optimización, para la determinación del efecto en el desempeño de la ejecución del algoritmo basado en computación con esténcil.

4. Se define una superficie de respuesta para determinar los valores óptimos de los parámetros con los cuales se obtiene la menor latencia para la implementación propuesta.

5. Se desarrolla una metodología que puede ser empleada en la implementación, evaluación y optimización automática del algoritmo.

El documento está organizado como se describe a continuación. En el Capítulo 2 se presenta el concepto de la computación con esténcil y se hace referencia a algunos trabajos desarrollados sobre técnicas de optimización sobre sistemas basados en CPU y en arquitecturas heterogéneas. Se presenta el codiseño hardware/software para la implementación en FPGA de dos arquitecturas base para la aproximación a la solución numérica de problemas de propagación de calor modelados con la ecuación de calor unidimensional y la ecuación de Laplace bidimensional.

En el Capítulo 3 se presenta el diseño e implementación de dos arquitecturas paralelas desarrolladas a partir de una arquitectura base, para la aceleración del tiempo de ejecución del algoritmo basado en esténcil utilizando una sola FPGA. Para todas las arquitecturas desarrolladas se presenta el diseño a nivel de transferencia entre registros, basado en el modelo de Glushkov. La implementación de estas arquitecturas es realizada utilizando el sistema de desarrollo comercial ZedBoard.

Adicionalmente, se presentan los resultados de la evaluación del desempeño del sistema en términos del tiempo de ejecución del algoritmo con las arquitecturas implementadas. Se muestra la comparación de los tiempos de procesamiento y los resultados numéricos con los obtenidos de la ejecución del algoritmo sobre un microprocesador. La evaluación de rendimiento es realizada para la arquitectura base y las dos arquitecturas paralelas. La aceleración obtenida con las arquitecturas paralelas es calculada con relación a un sistema basado en CPU, a un núcleo de ARM del SoC-FPGA y a la arquitectura base. Se muestra que es posible realizar la ejecución del algoritmo basado en esténcil en menor tiempo. La comparación de las arquitecturas implementadas, permite observar cómo las variaciones propuestas en el camino de datos influye en la reducción del tiempo de ejecución del algoritmo. Se muestra que uno de los aspectos del camino de datos que más influye en el rendimiento es la estructura de la memoria. Además se muestra que el uso del arreglo de registros introducido en el camino de datos permite tomar ventaja de la localidad espacial y temporal de los datos, reduciendo la cantidad de recursos utilizados y de operaciones de transferencia de memoria.

En el Capítulo 4 se presenta una estrategia para la implementación de algoritmos basados en esténcil en SoC-FPGA utilizando una herramienta de síntesis de alto nivelVivado HLS. El problema de optimización es trabajado en dos frentes: gestión de memoria y paralelización de ciclos. Para lograr la tarea, se propone un desenrollado manual en el bucle interno a partir de la división de la malla en bloques rectangulares a lo largo de la dimensión y con el fin de reducir la latencia del bucle intermedio. Adicionalmente, se realiza el uso de directivas de partición de memoria on-chip y de paralelización tipo pipeline para incrementar la tasa de transferencia . El rendimiento es evaluado de acuerdo con la cantidad de divisiones y las particiones de memoria on-chip en términos de latencia, consumo de energía, uso de recursos del FPGA y aceleración. Se obtiene una aceleración de aproximadamente 6.7× con relación a la CPU utilizada como referencia, con un consumo de potencia de aproximadamente 3.6 vatios. Finalmente son presentadas las conclusiones y trabajo futuro.

2. Computación con esténcil sobre arquitecturas heterogéneas

Un esquema muy usado en la computación científica se conoce como computación con esténcil [18, 38]. Los algoritmos iterativos de computación con esténcil están presentes en aplicaciones científicas y de ingeniería, tales como: integración numérica de ecuaciones diferenciales parciales, [28, 58, 36], procesamiento de imágenes [9, 42], simulación Particle-in-Cell [42], procesamiento de grafos [32], entre otros.

La computación con esténcil involucra una variable $v_t(r)$ definida para un conjunto de puntos discretos $\Omega = \{r\}$ distribuidos uniformemente en un arreglo regular y un número de pasos de tiempo de simulación $T = \{t\}$. El algoritmo consiste en el recorrido iterativo realizado sobre Ω para actualizar el valor de $v_t(r)$, empleando un método de aproximación numérica en función de los valores de los puntos vecinos y del punto de interés [18, 38]. El pseudocódigo del esquema basado en esténcil se define como se muestra en Algoritmo 1 [18].

Algoritmo de descripción 1: Pseudocódigo del algoritmo de computación con esténcil. (Naive)

```
1 foreach t ∈ T do
2 |   foreach r ∈ Ω do
3 |   |   v_{t+1}(r) ← f({v_t(r') | r' ∈ vecinos(r)})
4 |   end
5 end
```

La características de un esténcil están determinadas por la geometría del conjunto de puntos vecinos y el punto de interés, según la cual se define la dimensionalidad, el orden, el tamaño y la huella [11, 18]. El esquema basado en esténcil es muy eficiente para la aproximación a la solución numérica de EDP empleando el método de diferencias finitas explícito [4]. Cada punto de la malla es actualizado a partir de la contribución ponderada de un subconjunto de sus vecinos, en el dominio del espacio y el dominio del tiempo [11].

2.1. Aceleración de algoritmos de computación con esténcil

Los algoritmos de computación con esténcil tienen un ancho de banda constante, es decir, la cantidad de bytes transferidos de la memoria a la CPU por operación de punto flotante (B/F) no depende del tamaño de la memoria caché [34]. Generalmente la implementación del esquema basado en esténcil presenta bajo rendimiento cuando se emplean ciclos anidados sobre arquitecturas multiprocesador [54]. Esto se debe principalmente a las limitaciones de ancho de banda [34, 43, 52]. Por esta razón, la aceleración de códigos basados en esténcil ha sido ampliamente estudiada [63, 13, 59, 30, 60, 6, 36, 34, 48, 35].

Una de las limitaciones más importantes del cálculo del esténcil es su baja intensidad operacional, la cual se define como la cantidad de operaciones de punto flotante que son realizadas con relación a la catidad de datos que son accedidos desde la memoria DRAM [61]. Esta característica se traduce en un bajo desempeño en sistemas basados en CPU o GPU [59, 43, 11].

Algunas técnicas de optimización han sido desarrolladas para sistemas basados en CPU o GPU buscando superar las limitaciones de desempeño explotando la localidad espacial y temporal de los datos [34, 1, 7, 53, 11]. Estos algoritmos de optimización utilizan un conjunto de regiones obtenidas a través de la partición del dominio Ω en paralelogramos espacio-tiempo para optimizar el uso de la memoria caché [4, 34]. En la técnicas conocida como sesgado de tiempo o *time-skewing* la partición es realizada como se muestra en la Figura **2-1**a [53]. Aunque las particiones están definidas considerando la dependencia espacial y temporal de los cálculos, este algoritmo tiene limitaciones por las dependencias entre los bloques generados. Por otra parte la técnica llamada *cache oblivious* que se muestra en la Figura **2-1**b, establece las particiones de tal forma que no se requiere información de la jerarquía de memoria [11]. Sin embargo, eisten limitaciones de desempeño que se mantienen a pesar del uso de los métodos de optimización de los algoritmos basados en esténcil [43].

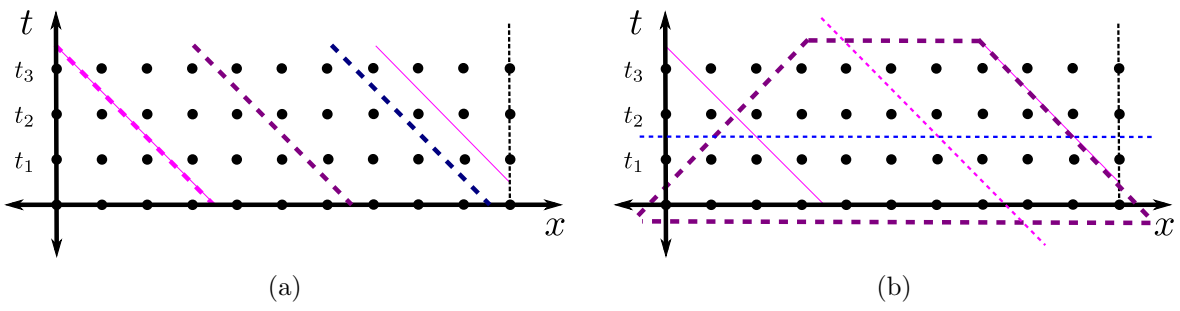


Figura 2-1.: Regiones definidas según la técnica de optimización: a)time skewing, b) cache oblivious.

Para superar algunas de estas limitaciones se han propuesto aceleradores basados en FPGA, buscando mejorar el rendimiento de algoritmos basados en esténcil con un bajo consumo de energía [33, 59, 13, 12, 40, 50, 27, 47, 56, 43, 45, 26]. Los sistemas basados en FPGA toman ventaja del paralelismo inherente a través de la combinación de diversas técnicas de optimización del flujo de datos. Los FPGA tienen un gran número de registros, lo que facilita la transferencia de datos entre las iteraciones de un cálculo sin la necesidad de acceder a una memoria externa. Esto conduce a un aumento en la intensidad operacional y la velocidad de procesamiento [59].

Los sistemas basados en FPGA generalmente se implementan mediante un lenguaje de descripción de hardware (HDL) [59, 37, 27, 21]. Sin embargo, los diseños basados en HDL requieren un amplio conocimiento del hardware [59]. Para subir el nivel de abstracción de los diseños y facilitar la implementación, se han desarrollado diferentes herramientas de síntesis de alto nivel (HLS) [37, 46, 39, 31, 63, 20]. Las herramientas HLS permiten ignorar los detalles del hardware, pero a menudo ofrecen soluciones menos eficientes en comparación con las obtenidas mediante HDL [37]. En estos casos, es necesario volver a escribir manualmente el código para optimizar aspectos como el acceso a la memoria [20].

En la literatura se encuentran diferentes intentos de mejorar el rendimiento de las soluciones HLS. Por ejemplo, en [13], se ha explorado un conjunto de opciones de diseño para acomodar un gran conjunto de restricciones. En algunos de estos trabajos logran un alto rendimiento al evadir el bloqueo espacial y restringir el tamaño de entrada. Por otro lado, el bloqueo espacial y temporal se combinan para evitar restricciones de tamaño de entrada como en [63].

Es bien sabido que uno de los cuellos de botella en las soluciones HLS es el acceso a los datos [20, 10]. De esta forma, es necesario optimizar la gestión de la memoria. En [20], la teoría de grafos se utiliza para optimizar el uso de los bancos de memoria. En [10], se propone una partición no uniforme de la memoria de tal manera que se minimice el número de bancos de memoria. La canalización de bucles es otro método clave para la optimización en HLS [30]. Sin embargo, el nivel de rendimiento de las soluciones puede no ser óptimo cuando aparecen dependencias de memoria complejas. En [29, 31, 30], se mejoran las capacidades de canalización de bucles para manejar dependencias de memoria inciertas.

2.2. Caso de estudio: ecuación de calor unidimensional.

Considérese la ecuación diferencial parcial (EDP) que se muestra en (2-1).

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \qquad 0 < x < L \tag{2-1}$$

Esta ecuación es empleada para modelar la distribución de calor en una barra de longitud L, donde $u(x,t)$ representa la temperatura en el punto x en el instante t, y α es la difusividad térmica del material. Por esta razón se conoce como la ecuación de calor. Dadas unas condiciones iniciales y condiciones de frontera como se muestra en (2-2), la solución de la ecuación muestra la variación de la temperatura en el dominio espacio-tiempo.

$$\begin{cases} u(x,0) = & f(x) \\ u(0,t) = & 0 \\ u(L,t) = & 0 \end{cases} \tag{2-2}$$

Una aproximación a la solución numérica de esta ecuación se obtiene empleando el método de diferencias finitas explícito. Los dominios del espacio y el tiempo son discretizados definiendo un número de puntos J y N respectivamente. De esta forma el dominio de la solución es dividido en una malla de $J \times N$ puntos como se muestra en la Figura **2-2**, donde x_j y t_n se definen como se muestra en (2-3) y (2-4).

$$t_n = n\Delta t, \qquad \Delta t = t_F/N, \quad n = 0, 1, ..., N \tag{2-3}$$

$$x_j = j\Delta x, \qquad \Delta x = L/J, \quad j = 0, 1, ..., J \tag{2-4}$$

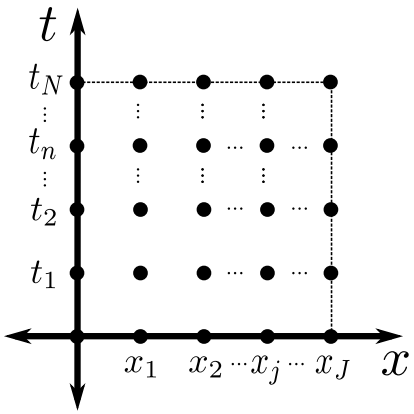


Figura 2-2.: Discretización de los ejes de espacio y tiempo en una malla uniforme de JxN puntos.

La aproximación a la solución numérica de la ecuación (2-1) se obtiene como en (2-5) a partir de las sustituciones que se muestran en (2-6) y (2-7), considerando una distribución uniforme de los puntos en el dominio de la solución.

$$u_j^{n+1} = u_j^n + \alpha(u_{j+1}^n - 2u_j^n + u_{j-1}^n) \tag{2-5}$$

$$u_t(x_j, t_n) \approx \frac{(u_j^{n+1} - u_j^n)}{\Delta t} \tag{2-6}$$

$$u_{xx} \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{2-7}$$

La expresión (2-5) puede ser representada gráficamente como la plantilla o esténcil de cuatro puntos que se muestra en la Figura **2-3**.

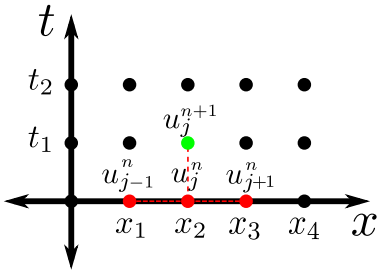


Figura 2-3.: Plantilla de cuatro puntos para la aproximación a la solución numérica de la ecuación de calor por el método de diferencias finitas explícito.

La ejecución iterativa del algoritmo basado en esténcil se realiza como se muestra en la Figura **2-4** para las primeras cuatro iteraciones.

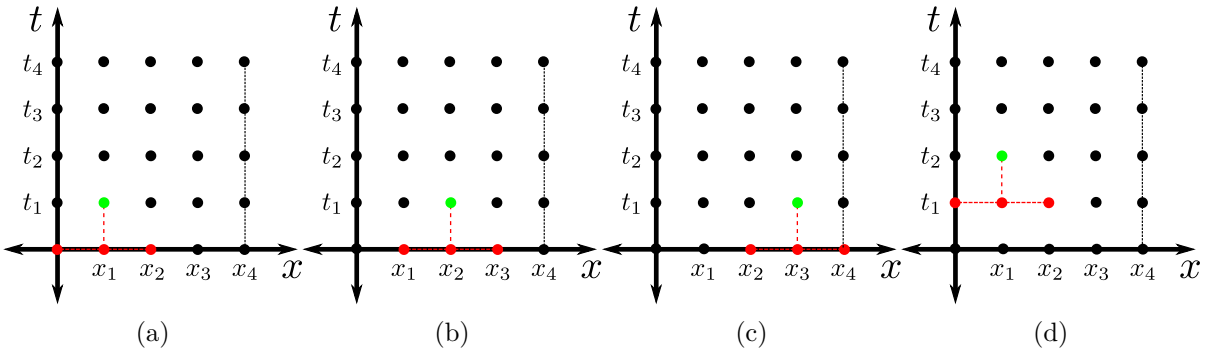


Figura 2-4.: Primeras cuatro iteraciones de la ejecución secuencial del algoritmo basado en esténcil para una malla de 5x5.

El barrido en el dominio de la solución se realiza de la forma que se muestra en Algoritmo 2.

Algoritmo de descripción 2: Pseudocódigo del algoritmo de computación con esténcil utilizando la arquitectura base.

Input: valores iniciales, condiciones de frontera, tamaño de malla
Output: temperatura en la iteración N

```
1 Loop 1: for n ← 0 to N − 1 do
2 |   Loop 1.1: for j ← 1 to J − 2 do
3 |   |   u_j^{n+1} ← (1 − 2α)u_j^n + α(u_{j+1}^n + u_{j−1}^n)
4 |   end
5 end
```

2.2.1. Sistema basado en FPGA para la aproximación a la solución numérica de la ecuación de calor unidimensional

Las arquitecturas desarrolladas son implementadas en el *ZedBoard Zynq Evaluation and Development Kit* utilizando el entorno *Vivado Design Suite - HLx Edition* versión 2018.3. En la Figura **2-5** se muestra el sistema de desarrollo utilizado.

Figura 2-5.: Sistema de desarrollo *ZedBoard Zynq Evaluation and Development Kit*

En este sistema de desarrollo un microprocesador ARM Cortex-A9 y un FPGA de la familia Artix-7 están integrados en un solo dispositivo conocido como *System-on-Chip* FPGA (SoC-FPGA). El ARM actúa como el procesador maestro sobre el cual se ejecuta la aplicación principal. El código fuente de la aplicación es realizado en C y es ejecutado bajo un sistema operativo *stand-alone*. La descripción del algoritmo basado en esténcil es realizada en VHDL y empaquetada en un IP personalizado. El microprocesador interactúa con el IP a través de

una interfaz de comunicación AXI. El ARM trabaja con un reloj de 667 MHz y el FPGA trabaja con un reloj de 100 MHz. El diagrama de bloques del sistema implementado se muestra en la Figura **2-6**.

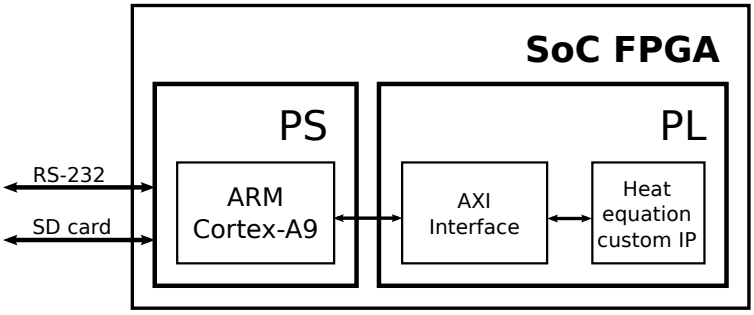


Figura 2-6.: Diagrama de bloques del sistema implementado un *ZedBoard Zynq Evaluation and Development Kit* bajo el entorno *Vivado Design Suite*.

El programa principal es ejecutado usando la consola de un terminal serial, a través de la cual se solicita la definición del tamaño de la malla para la generación de los valores iniciales. Estos datos son enviados hacia el IP, al igual que las condiciones de frontera y los comandos de configuración y control. Por otra parte los resultados numéricos, las señales de estado y el contador de desempeño son leídos por el programa principal desde el IP. Los datos son almacenados en un archivo de texto en formato decimal de 15 cifras significativas. Estos datos son representados en un formato personalizado de punto flotante de 32 bits con redondeo al valor más cercano, compatible con el formato estándar. El diseño RTL del bloque IP es realizado con base en el modelo de Glushkov, según el cual el sistema digital está conformado principalmente por un camino de datos y una unidad de control como se muestra en el diagrama de bloques de la Figura **2-7**.

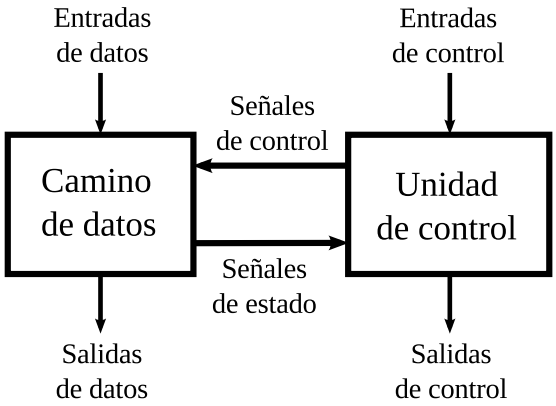


Figura 2-7.: Diagrama de bloques general conformado por el camino de datos y la unidad de control (modelo de Glushkov).

En esta arquitectura base (A1) el camino de datos está conformado por elementos de proce-

saminento (EP), registros, multiplexores, y memorias RAM. La secuencia para la ejecución del algoritmo basado en esténcil es coordinado por la unidad de control, la cual es descrita en VHDL como una máquina de estados finitos. Un contador binario es utilizado para medir la latencia total del cálculo de todos los puntos de la malla. El diagrama de bloques de esta arquitectura se muestra en la Figura **2-8**.

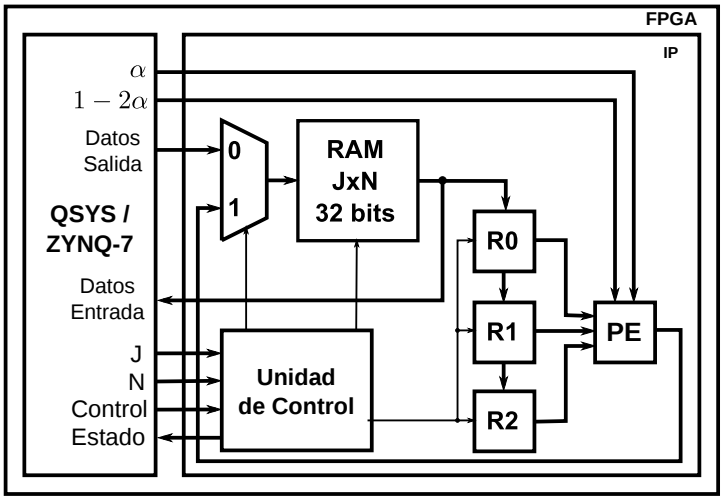


Figura 2-8.: Diagrama de bloques de la arquitectura base.

El circuito correspondiente al EP para realizar la operación con esténcil definida en (2-5) es implementado con base en el diagrama de bloques que se muestra en la Figura **2-9**. Las operaciones de punto flotante en los EP son descritas como circuitos combinacionales, por lo que la latencia de la operación con esténcil es de un ciclo de reloj.

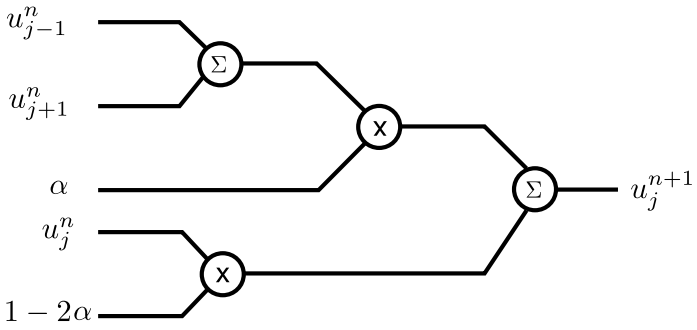


Figura 2-9.: Diagrama de bloques del elemento de procesamiento.

Para evitar el cambio de signo en la operación, la resta es realizada en el término $1-2\alpha$, considerando que éste valor puede ser definido como una constante. El diseño de este circuito es realizado teniendo en cuenta que la operación de la suma requiere el uso de operaciones lógicas más simples que en el caso de la multiplicación. Esto implica la reducción de la cantidad de recursos físicos necesarios para la implementación, sin una variación significativa

del tiempo de respuesta del circuito. La ejecución del algoritmo se realiza de acuerdo con el diagrama de flujo que se muestra en la Figura **2-10**. La sección del algoritmo que se ejecuta en el FPGA se muestra dentro de la línea punteada.

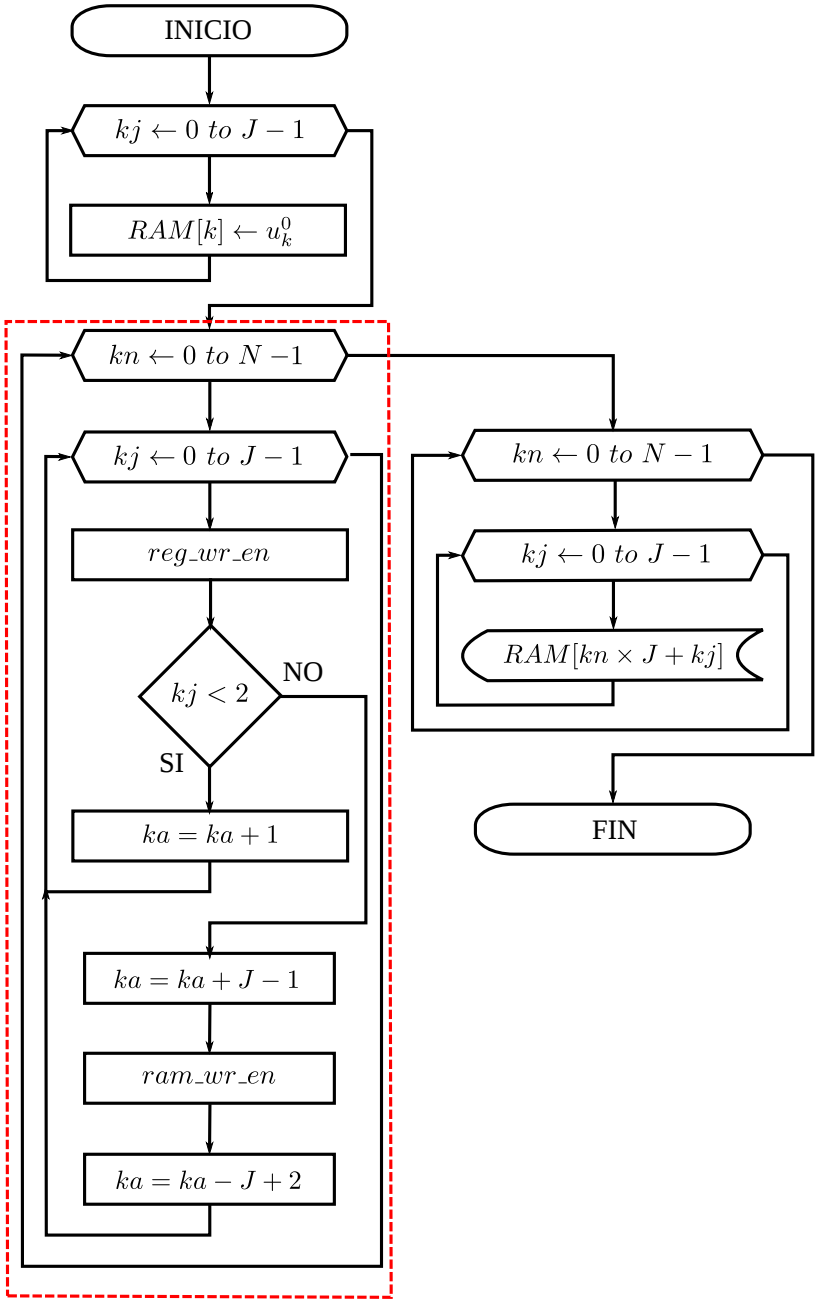


Figura 2-10.: Diagrama de flujo de la unidad de control. La parte de la secuencia ejecutada por el procesador maestro se muestra por fuera de la línea punteada. La parte de la secuencia que se ejecuta en el FPGA se muestra dentro de la línea punteada.

La parte de la secuencia ejecutada por el procesador maestro que se muestra por fuera de la línea punteada realiza las siguientes operaciones:

1. Definir los valores de J y N y escribirlos en los registros correspondientes en el IP
2. Inicializar la memoria RAM del IP, escribiendo las condiciones de frontera en las posiciones de memoria correspondientes
3. Generar el vector de valores iniciales y escribirlos en las primeras J posiciones de la RAM
4. Enviar la señal de que indica a la unidad de control el inicio de la secuencia de procesamiento
5. Esperar la señal de estado de la unidad de control que indica que ha finalizado la secuencia de procesamiento
6. Leer los datos de la memoria RAM y escribirlos en el archivo de texto

La parte que se encuentra entre la línea segmentada corresponde a la secuencia ejecutada por la unidad de control. Los contadores k_j y k_n son empleados para el control de los ciclos. El contador k_a es empleado para el direccionamiento de la RAM.

La conexión en cascada de los registros permite un flujo continuo de datos desde la memoria. Con cada ciclo de reloj es realizada la carga de un dato en el registro $R0$. En la medida que se obtiene cada término u_j^{n+1} a la salida de EP, el valor es almacenado en la RAM a través del multiplexor. De esta forma los resultados obtenidos están disponibles para las iteraciones siguientes hasta completar el cálculo de todos los puntos de la malla, sin necesidad de operaciones de transmisión con el procesador maestro. Cuando el proceso termina la señal de estado es desactivada para indicar al procesador la finalización. El diagrama de transición de estados se muestra en la Figura **2-11**.

La descripción de los estados para la máquina de estados finitos correspondiente a la unidad de control es la siguiente:

e_0: estado inicial, control de inicio. Inicializa el contador de desplazamiento en el eje x (k_j), el contador de iteraciones (k_n) y el contador de direccionamiento de memoria (k_a). Activa la habilitación de escritura y direccionamiento externo de la memoria RAM. Selecciona el multiplexor para fuente de datos externa (desde el procesador maestro). Esto permite el almacenamiento del vector de condiciones iniciales en las primeras J posiciones. Si la señal de inicio es activa pasa a e_1, de lo contrario permanece en e_0. La señal de estado permanece inactiva para indicar al procesador maestro que el sistema está disponible.

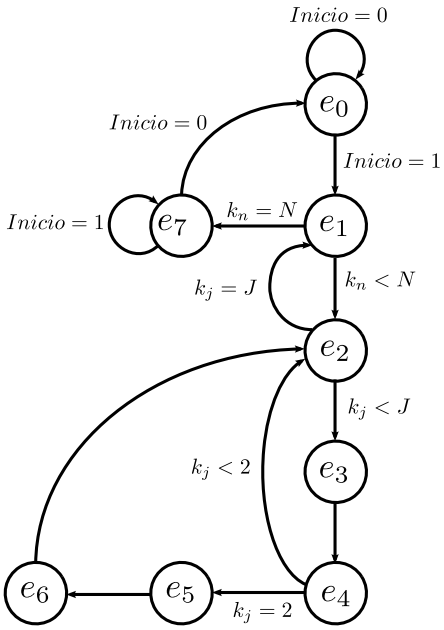


Figura 2-11.: Diagrama de transición de estados de la unidad de control.

e_1: control de iteraciones. Compara k_n con el número de iteraciones (N). Selecciona el multiplexor para la fuente de datos interna. Si k_n es menor que N pasa a e_2, de lo contrario pasa a e_7. Se activa la señal de estado que indica al procesador maestro que el sistema está ocupado procesando.

e_2: control de barrido en el dominio del espacio. Compara k_j con el número de puntos en el eje x (J). Si k_j es menor que J pasa a e_3. Si es igual inicializa el contador k_j e incrementa k_n en 1. Pasa a e_1.

e_3: escritura en registros. Activa la habilitación de escritura en registros. Pasa a e_4.

e_4: control de carga inicial de registros. Se garantiza que para el cálculo del término u_1^{n+1} los registros están cargados con los términos u_0^n, u_1^n y u_2^n. Para esto, si k_j es menor que 2 incrementa en 1 los contadores k_j y k_a, pasa a e_2. De lo contrario incrementa k_a en $J-1$ y pasa a e_5.

e_5: escritura en memoria. Habilita la escritura en memoria para el almacenamiento del término $u_{k_j-1}^{k_n+1}$ en la dirección k_a. Pasa a e_6.

e_6: control de k_a y k_j. Decrementa k_a en $J-2$ e incrementa k_j en 1. Pasa a e_2.

e_7: control de finalización. Si la señal de inicio continúa activa, permanece en e_7. De lo contrario pasa a e0.

La latencia de esta máqiuna de estados en ciclos de reloj (n_{CLK}) requeridos para la ejecución del algoritmo se obtiene empleando la expresión 2-8.

$$n_{CLK} = N(5J-2)+2 \tag{2-8}$$

Este valor se obtiene de la cantidad de veces que la unidad de control se encuentra en cada estado según se muestra en la Tabla **2-1**.

Tabla 2-1.: Latencia para cada estado de la unidad de control a partir de la señal de inicio.

Estado	Latencia
e_0	1
e_1	$N-1$
e_2	$(J+1)\times(N-1)$
e_3	$J\times(N-1)$
e_4	$J\times(N-1)$
e_5	$(J-2)\times(N-1)$
e_6	$(J-2)\times(N-1)$
e_7	1

2.3. Caso de estudio: ecuación de Laplace bidimensional

Supóngase Ω como un dominio de R^2 con contorno definido como $\partial\Omega$. La EDP que se muestra en (2-9) es considerada elíptica para todos los puntos $(x, y) \in \Omega$.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{2-9}$$

Puede ser empleada para modelar la distribución estacionaria de temperatura en una región bidimensional dadas unas condiciones de frontera y se describe como se muestra en (2-10), donde u_{xy} representa la temperatura en el punto (x, y).

$$\begin{cases} u_{xx} + u_{yy} = 0 \\ u = g(x,y), \quad \forall (x,y) \in \partial\Omega \end{cases} \tag{2-10}$$

Una aproximación a la solución numérica de esta ecuación se obtiene empleando el método de diferencias finitas. La región Ω es discretizada en las dos dimensiones definiendo un número de puntos I y J respectivamente. De esta forma el dominio de la solución consiste en una malla de $I \times J$ puntos como se muestra en la Figura **2-12**, donde x_i y y_j se definen como se muestra en (2-11) y (2-12).

$$x_i = i\Delta x, \quad \Delta x = L_x/I, \ i = 0, 1, ..., I \tag{2-11}$$

$$y_j = j\Delta y, \quad \Delta y = L_y/J, \ j = 0, 1, ..., J \tag{2-12}$$

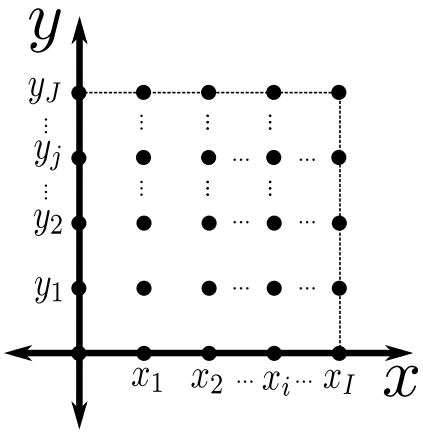


Figura 2-12.: Discretización de los ejes espaciales en una malla uniforme de IxJ puntos.

La aproximación a la solución numérica de la ecuación (2-9) se obtiene como en (2-13) a partir de las sustituciones que se muestran en (2-14) y (2-15), considerando una distribución uniforme de los puntos en el dominio de la solución.

$$u_{ij} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) \tag{2-13}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{(u_{i+1,j} - 2u_{ij} + u_{i-1,j})}{(\Delta x)^2} \tag{2-14}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{(u_{i,j+1} - 2u_{ij} + u_{i,j-1})}{(\Delta y)^2} \tag{2-15}$$

La expresión (2-13) es representada gráficamente como la plantilla o esténcil de cinco puntos que se muestra en la Figura **2-13**.

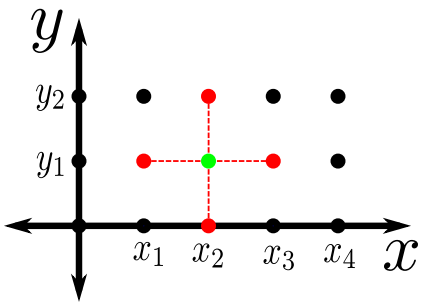


Figura 2-13.: Plantilla de cinco puntos para la aproximación a la solución numérica de la ecuación de Laplace por el método de diferencias finitas.

El recorrido de todos los puntos internos de la malla es realizado un número determinado de iteraciones aplicando la expresión 2-13 de la forma que se muestra por 2-16.

$$u_{ij}^{n+1} = \frac{1}{4}(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}) \tag{2-16}$$

La implementación de esta aproximación a la solución numérica de la ecuación de Laplace en dos dimensiones es conocida como el algoritmo de Jacobi [7], el cual se describe para un número determinado de iteraciones N como se muestra en Algoritmo 3.

Algoritmo de descripción 3: Pseudocódigo del algoritmo basado en esténcil para la aproximación a la solución numérica de la ecuación de Laplace usando diferencias finitas.

```
1 Loop 1: for n ← 0 to N − 1 do
2 |   Loop 1.1: for j ← 1 to J − 1 do
3 |   |   Loop 1.1.1: for i ← 1 to I − 1 do
4 |   |   |   u_{i,j}^{n+1} ← 0.25(u_{i+1,j}^{n} + u_{i−1,j}^{n} + u_{i,j+1}^{n} + u_{i,j−1}^{n})
5 |   |   end
6 |   end
7 end
```

2.4. Sistema basado en FPGA para la aproximación a la solución numérica de la ecuación de Laplace bidimensional

El esquema general del sistema implementado involucra el uso de un microprocesador ARM Cortex-A9 que actúa como maestro, sobre el cual se ejecuta la aplicación. El diagrama de bloques del sistema implementado se muestra la Figura **2-14**.

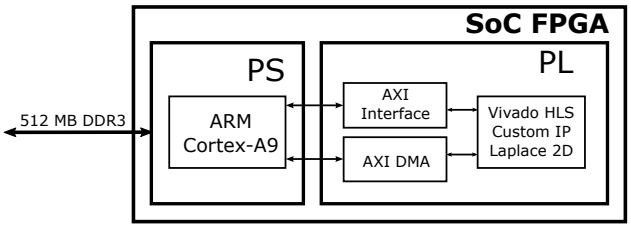


Figura 2-14.: Diagrama de bloques del sistema implementado en un *ZedBoard Zynq Evaluation and Development Kit* bajo el entorno *Vivado Design Suite*.

El procesador interactúa a través de una interfaz AXI con un IP creado en *Vivado HLS*.

El IP se encarga de la ejecución del algoritmo basado en esténcil. Las arquitecturas desarrolladas son implementadas sobre un sistema de desarrollo *ZedBoard Zynq Evaluation and Development Kit* bajo el entorno *Vivado Design Suite*. El código fuente de la aplicación principal es realizado en C y ejecutado bajo PetaLinux sobre el PS. La comunicación es realizada empleando una interfaz AXI y acceso directo a memoria (DMA). La aplicación principal funciona como un programa maestro que se usa a través de una consola de terminal como interfaz de usuario. El código fuente incluye la generación de los valores iniciales y las condiciones de frontera, los cuales son almacenados en la memoria RAM. Después de la definición de la cantidad de iteraciones, se realiza la ejecución del algoritmo a través de un llamado a la función correspondiente al IP. Una vez terminada la ejecución, los resultados quedan disponibles en la memoria RAM para ser leídos o almacenados en un archivo de texto en formato decimal con 15 cifras significativas. La implementación secuencial del código se realiza como referencia para la evaluación del desempeño de las implementaciones con paralelización. El código de la función principal en *Vivado HLS* para la aproximación a la solución numérica de la ecuación de Laplace en dos dimensiones se basa en el pseudocódigo mostrado en Algoritmo 4 utilizando un arreglo de dos dimensiones.

Algoritmo de descripción 4: Pseudocódigo del algoritmo de computación con esténcil con arreglo bidimensional para N iteraciones.

Input: valores iniciales, condiciones de frontera, tamaño de malla, numero de iteraciones
Output: temperatura en la iteración N

```
1  Loop 1: for n ← 0 to N − 1 do
2  |  Loop 1.1: for j ← 1 to J − 1 do
3  |  |  Loop 1.1.1: for i ← 1 to I − 1 do
4  |  |  |  v[j, i] ← 0.25 (u[j, i + 1] + u[j, i − 1] + u[j + 1, i] + u[j − 1, i])
5  |  |  end
6  |  end
7  |  Loop 1.2: for j ← 1 to J − 1 do
8  |  |  Loop 1.2.1: for i ← 1 to I − 1 do
9  |  |  |  u[j, i] ← v[j, i]
10 |  |  end
11 |  end
12 end
```

Debido a que los datos llegan a la función del esténcil como un vector, el código es modificado como se muestra en Algoritmo 5.

Algoritmo de descripción 5: Modificación de la operación con esténcil para los datos tomados como vector.

Input: valores iniciales, condiciones de frontera, tamaño de malla, numero de iteraciones
Output: temperatura en la iteración N

```
1  Loop 1: for n ← 0 to N − 1 do
2      Loop 1.1: for j ← 1 to J − 1 do
3          Loop 1.1.1: for i ← 1 to I − 1 do
4              s1 ← u[j ∗ X_I + i + 1]
5              s2 ← u[j ∗ X_I + i − 1]
6              s3 ← u[(j + 1) ∗ X_I + i]
7              s4 ← u[(j − 1) ∗ X_I + i]
8              v[j ∗ X_I + i] ← 0.25 (s1 + s2 + s3 + s4)
9          end
10     end
11     Loop 1.2: for j ← 1 to J − 1 do
12         Loop 1.2.1: for i ← 1 to I − 1 do
13             u[j ∗ X_I + i] ← v[j ∗ X_I + i]
14         end
15     end
16 end
```

El tamaño máximo que se puede usar para una malla cuadrada, está determinado por la cantidad de memoria *on-chip* del dispositivo. Considerando el uso formato de punto flotante simple y que el algoritmo requiere dos arreglos para almacenar los valores de las dos últimas iteraciones, el tamaño de la malla se calcula como se muestra en (2-17).

$$mesh\_size_{max} = \sqrt{\frac{RAM\_Blocks \times 18000}{32 \times 2}} \tag{2-17}$$

3. Aceleración del algoritmo basado en esténcil para la ecuación de calor unidimensional

La implementación y optimización de algoritmos basados en esténcil requiere consideración de la dependencia de datos. En este sentido, se propone la implementación de diferentes formas de optimización del flujo de datos a nivel de hardware, aprovechando la localidad espacial y temporal de los datos. Una opción consiste en el empleo de un vector aprovechando la localidad espacial. En este caso, la ejecución del algoritmo basado en esténcil se realiza como se muestra en la Figura **3-1**.

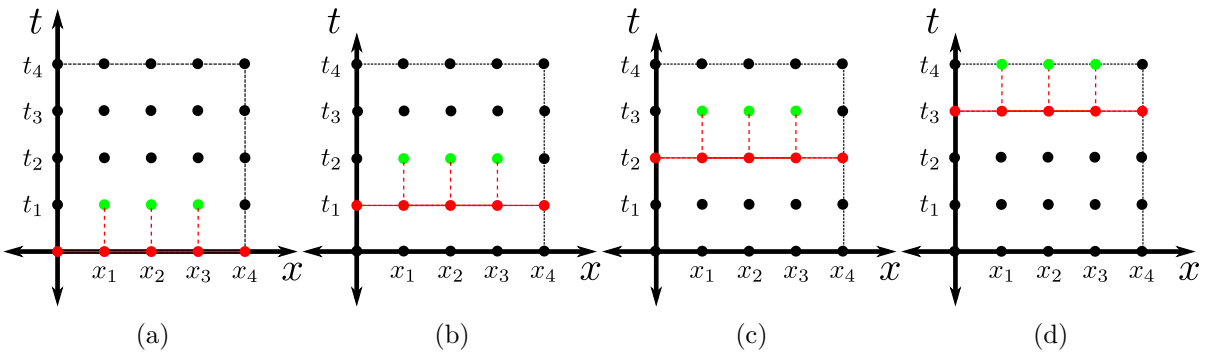


Figura 3-1.: Primeras cuatro iteraciones de la ejecución del algoritmo basado en esténcil para una malla de 5x5 empleando un vector.

Para la implementación de este esquema se propone una arquitectura con camino de datos conformado por seis EP y ocho registros para el procesamiento de mallas de $8 \times N$, como se muestra el diagrama de bloques de la Figura **3-2**.

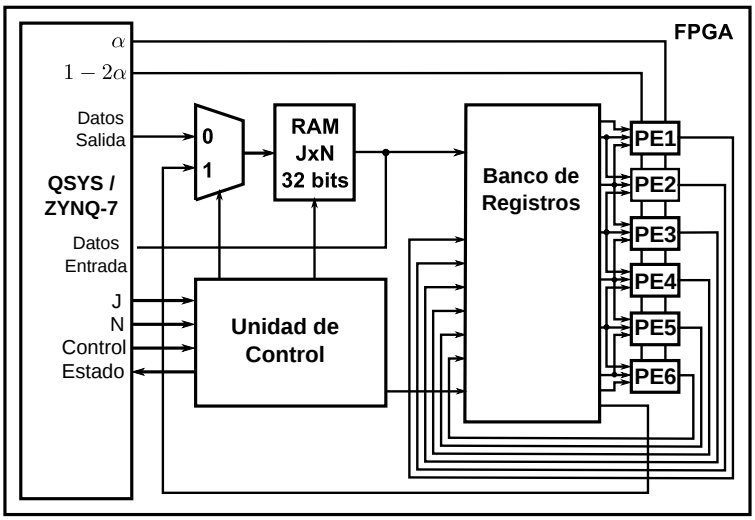


Figura 3-2.: Diagrama de bloques de la arquitectura implementada para mallas de $8 \times N$ con procesamiento concurrente.

El diagrama de flujo resultante para el diseño de la unidad de control es mostrado en la Figura **3-3**.

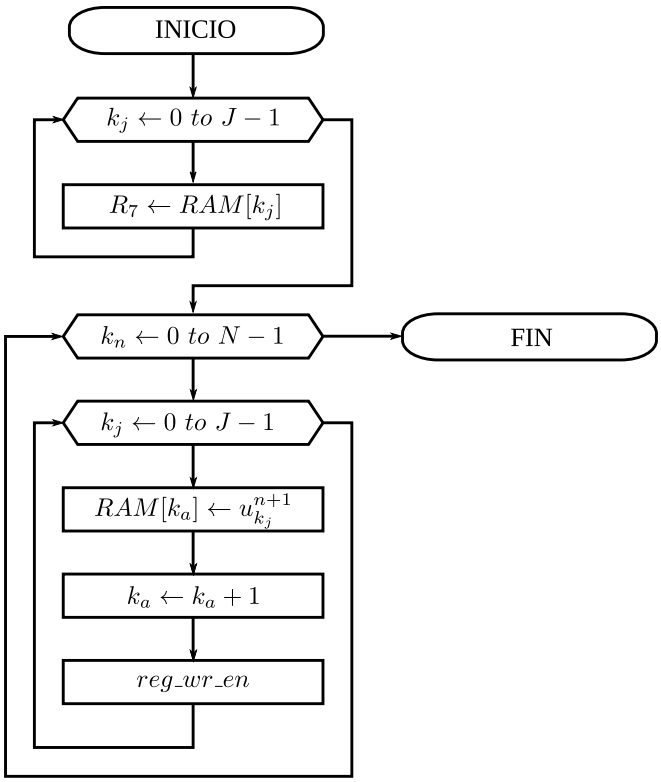


Figura 3-3.: Diagrama de flujo de la unidad de control para la ejecución del algoritmo empleando procesamiento concurrente y almacenamiento secuencial.

Con el diseño del camino de datos se reduce el número de estados requeridos para la ejecución del algoritmo, ya que los términos u_j^{n+1} son obtenidos de forma concurrente en un solo ciclo de reloj. Sin embargo, el almacenamiento de los datos se realiza de forma secuencial dado que existe una sola RAM, generando un cuello de botella. Por otra parte, la salida de los EP son guardadas de forma simultánea en el banco de registros a través de multiplexores internos. De esta forma los valores están disponibles para calcular los términos de la siguiente iteración sin acceder a la RAM.

La secuencia es realizada como se describe a continuación, considerando que en el estado inicial los valores iniciales están almacenados en las primeras posiciones de la RAM:

1. Se cargan los valores iniciales en los registros. Los resultados de las operaciones están disponibles al terminar la carga de los registros
2. Se almacenan los resultados en la memoria RAM
3. Se almacenan los resultados en el banco de registros. Los resultados de las operaciones están disponibles al terminar la carga de los registros. Se vuelve al paso anterior.

El diagrama de transición de estados se muestra en la Figura **3-4**. Esta unidad de control tiene un estado más que la arquitectura base, sin embargo el número de veces que pasa por algunos de los estados es menor.

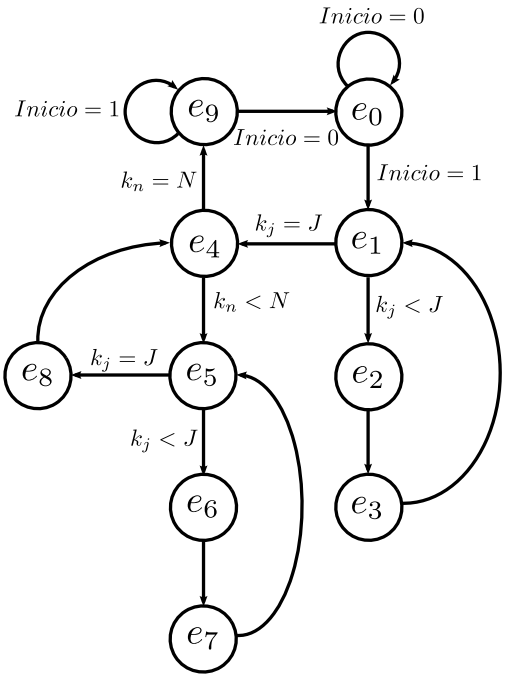


Figura 3-4.: Diagrama de transición de estados de la unidad de control para la ejecución del algoritmo empleando procesamiento concurrente.

La descripción para cada uno de los estados es la siguiente:

e_0: estado inicial, control de inicio. Inicializa los contadores k_j, k_n y k_a. Activa la habilitación de escritura y direccionamiento externo de la memoria RAM. Selecciona el multiplexor para fuente de datos externa, para el almacenamiento del vector de condiciones iniciales en las primeras J posiciones. Si la señal de inicio es activa pasa a $e1$, de lo contrario permanece en $e0$. La señal de estado permanece inactiva para indicar al procesador maestro que el sistema disponible.

e_1: control de carga de valores iniciales en el banco de registros. Si k_j es menor que J pasa a e_2, de lo contrario pasa a e_4. Se activa la señal de estado que indica al procesador maestro que el sistema esta ocupado procesando.

e_2: escritura en registros. Activa la habilitación de escritura en registros. Pasa a e_3.

e_3: control de contadores k_a y k_j. Incrementa k_a y k_j en 1. Pasa a e_2.

e_4: control de iteraciones. Compara k_n con el número de iteraciones (N). Selecciona el multiplexor para la fuente de datos interna. Si k_n es menor que N pasa a e_5, de lo contrario pasa a e_9.

e_5: control del ciclo para almacenamiento en memoria. Compara k_j con J. Si es menor pasa a e_6. Si es igual inicializa el contador k_j y pasa a e_8.

e_6: escritura en memoria. Habilita la escritura en memoria para el almacenamiento del término $u_{k_j}^{k_n+1}$ en la dirección k_a. Pasa a e_7.

e_7: control de contadores k_a y k_j. Incrementa k_a y k_j en 1. Pasa a e_5.

e_8: escritura en registros. Activa la habilitación de escritura del banco de registros, incrementa k_n y pasa a e_4.

e_9: control de finalización. Si la señal de inicio continúa activa, permanece en e_9. De lo contrario pasa a $e0$.

El número total de ciclos de reloj n_{CLK} requeridos para la ejecución del algoritmo se obtiene empleando la expresión 3-1.

$$n_{CLK} = N(4J + 1) - J + 2 \tag{3-1}$$

Una forma de superar las limitaciones generadas por tener solo una memoria, es asociar cada EP a una RAM para almacenar los resultados de forma concurrente. De esta forma el ciclo interno es eliminado de la secuencia de la unidad de control, disminuyendo la latencia total. En la Figura **3-5** se muestra el diagrama de bloques de la arquitectura implementada para mallas de $8 \times N$ con procesamiento y almacenamiento concurrente, empleando seis EP y ocho memorias RAM.

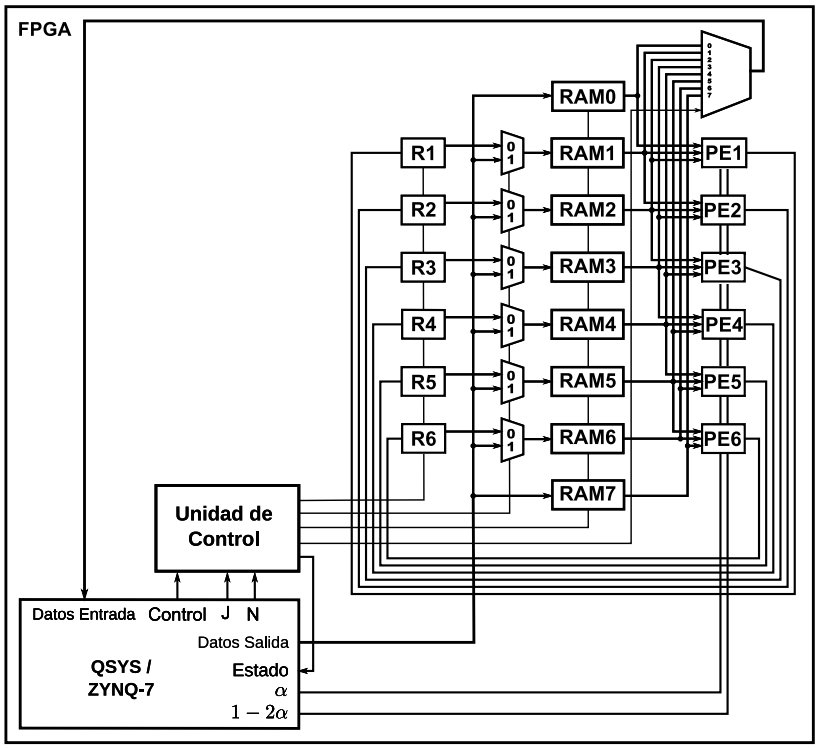


Figura 3-5.: Diagrama de bloques de la arquitectura implementada para mallas de $8 \times N$ con procesamiento y almacenamiento concurrente.

El diagrama de flujo para la ejecución del algoritmo empleando esta arquitectura se muestra en Figura **3-6**.

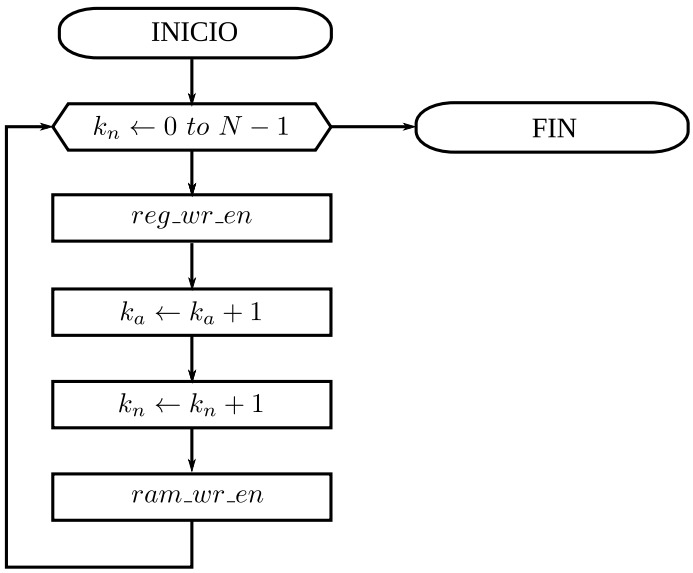


Figura 3-6.: Diagrama de flujo para la ejecución del algoritmo empleando procesamiento y almacenamiento concurrente.

El diagrama de transición de estados se muestra en la Figura **3-7**.

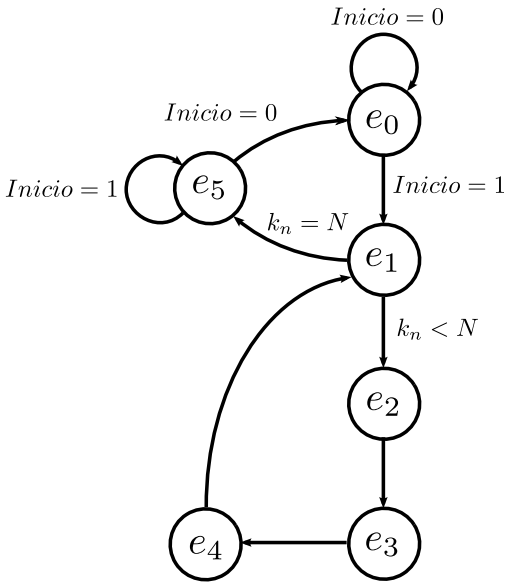


Figura 3-7.: Diagrama de transición de estados de la unidad de control para la ejecución del algoritmo empleando procesamiento y almacenamiento concurrente.

La descripción para cada uno de los estados es la siguiente:

e_0: estado inicial, control de inicio. Inicializa los contadores k_n y k_a. Activa la habilitación de escritura y direccionamiento externo de las memorias RAM. Selecciona los multiplexores en la fuente de datos externa, para el almacenamiento del vector de condiciones iniciales en la posición 0 de cada memoria. Si la señal de inicio es activa pasa a e_1, de lo contrario permanece en e_0. La señal de estado permanece inactiva para indicar al procesador maestro que el sistema disponible.

e_1: control de iteraciones. Compara k_n con el número de iteraciones (N). Selecciona el multiplexor para la fuente de datos interna. Si k_n es menor que N pasa a e_2, de lo contrario pasa a e_5. Se activa la señal de estado que indica al procesador maestro que el sistema esta ocupado procesando.

e_2: escritura en registros. Activa la habilitación de escritura en registros. Pasa a e_3.

e_3: control de contadores k_a y k_n. Incrementa k_a y k_n en 1. Pasa a e_4.

e_4: escritura en memoria. Habilita la escritura en las memorias para el almacenamiento del término $u_{k_j}^{k_n+1}$ en la dirección k_a. Pasa a e_1.

e_5: control de finalización. Si la señal de inicio continúa activa, permanece en e_5, de lo contrario pasa a e_0.

El número total de ciclos de reloj n_{CLK} requeridos para la ejecución del algoritmo se obtiene empleando la expresión 3-2. Como se puede observar, el número de ciclos de reloj necesarios para la ejecución del algoritmo es independiente de J.

$$n_{CLK} = 4N + 2 \tag{3-2}$$

3.1. Evaluación de desempeño

El desempeño de las arquitecturas implementadas es evaluado en términos del resultado numérico, el tiempo de ejecución del algoritmo basado en esténcil, la aceleración y el uso de recursos físicos en el FPGA. Los resultados numéricos obtenidos para diferentes tamaños de malla, se obtienen a partir de condiciones de frontera y valores iniciales generados en el programa principal. Los resultados de la aproximaión a la solución numérica de la ecuación de calor son leídos por el programa principal desde la memoria del circuito y almacenados en un archivo de texto usando formato decimal con 15 cifras significativas.

La aceleración es determinada a partir de la comparación de los tiempos de ejecución de las diferentes arquitecturas implementadas con referencia a la ejecución secuencial del algoritmo en un microprocesador. En cada arquitectura son medidos los tiempos de ejecución del algoritmo para diferentes tamaños de malla, teniendo en cuenta los tiempos correspondientes a la transferencia de datos desde el procesador, la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados en archivo de texto.

3.1.1. Evaluación en términos del resultado numérico

La evaluación del rendimiento en términos de la aproximación a la solución numérica se realiza en comparación con los resultados obtenidos con CPU. Los valores iniciales y las condiciones de frontera están definidos como se muestra en (3-3).

$$\begin{cases} u(x,0) = 2x, & x < 0.5 \\ u(x,0) = 2*(1-x), & 0.5 < x < 1 \\ u(0,t) = 0 \\ u(L,t) = 0 \end{cases} \tag{3-3}$$

La vista superior de las respuestas obtenidas con la arquitectura A_1 se muestra de la Figura **3-8**a a la Figura **3-8**f para mallas de 256 puntos en el eje x y 8, 16, 32, 64, 128, y 256 iteraciones respectivamente.

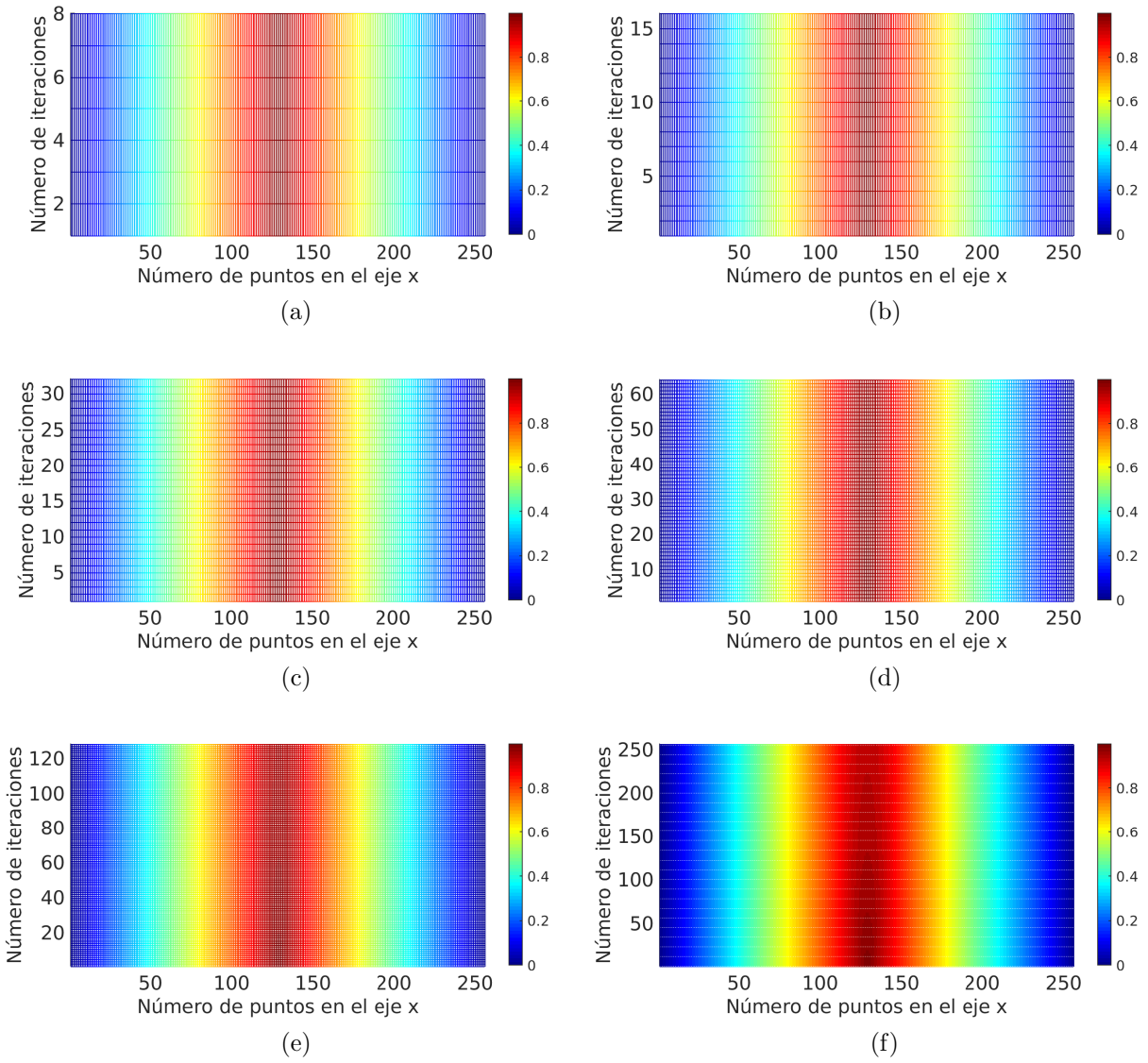


Figura 3-8.: Respuestas obtenidas con el sistema implementado en la ZedBoard para mallas de 256 puntos en el eje x y 8, 16, 32, 64, 128, y 256 iteraciones.

El error absoluto calculado con relación a los resultados numéricos obtenidos en CPU se muestra en la Figura **3-9**. Se observa que para una malla de 256×256 el error no supera el valor de 6×10^{-6}.

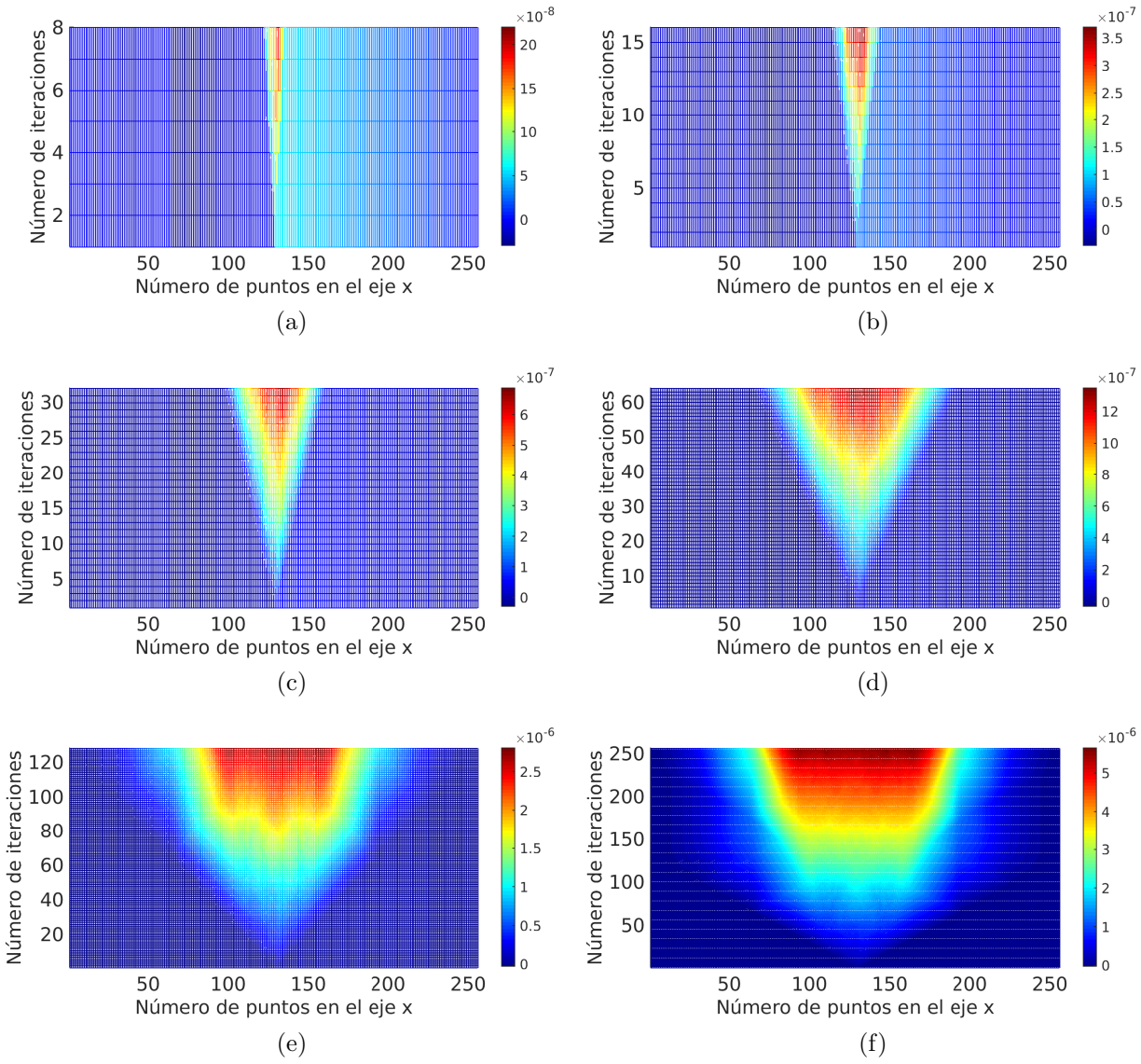


Figura 3-9.: Vista superior del error absoluto obtenido con el sistema implementado en la ZedBoard para mallas de 256 puntos en el eje x y (a) 8, (b) 16, (c) 32, (d) 64, (e) 128, y (f) 256 iteraciones.

El error relativo calculado con relación a los resultados numéricos obtenidos en CPU se muestra en la Figura **3-10**. Se observa que para una malla de 256×256 el error no supera el valor de 9×10^{-6}.

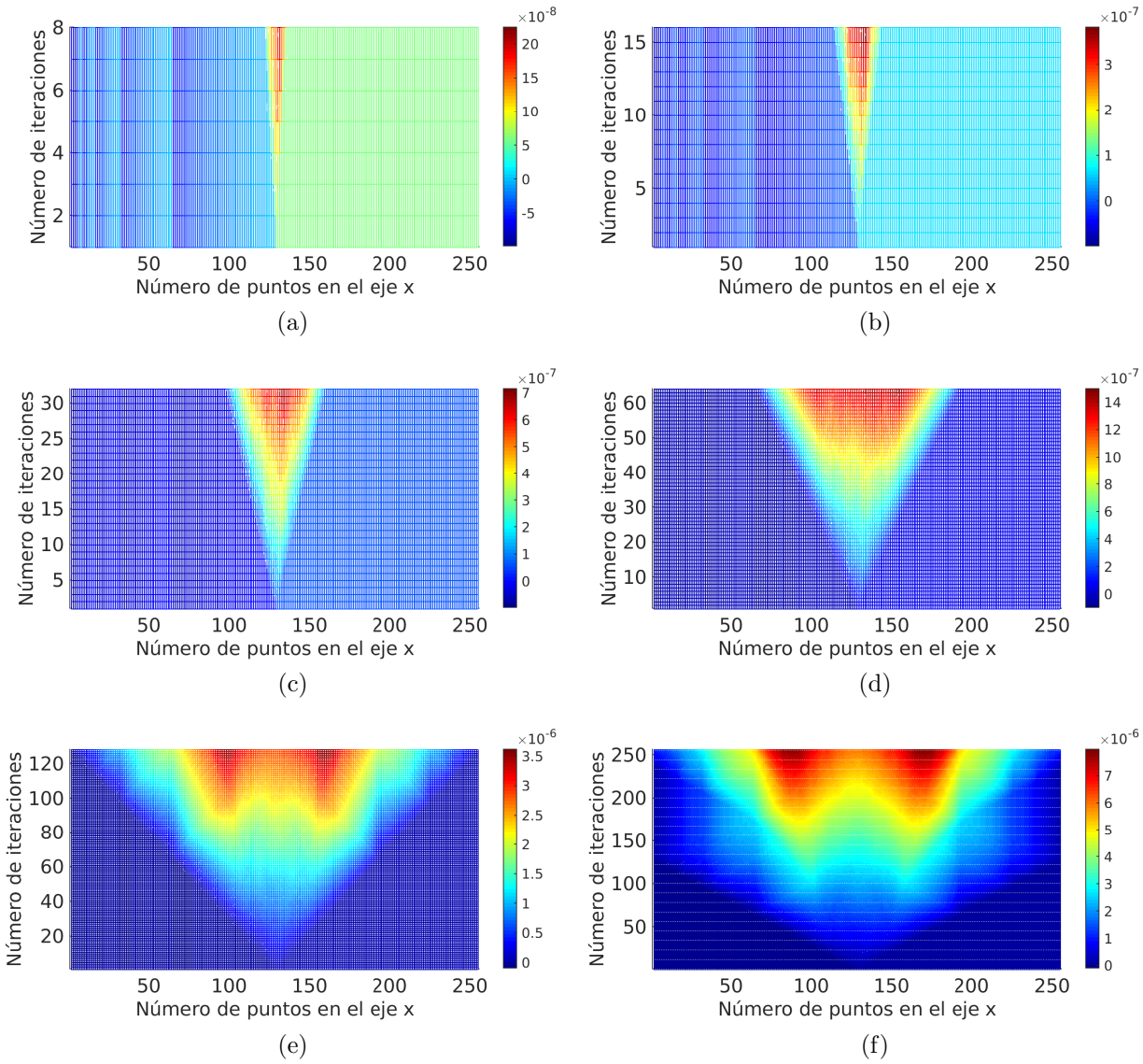
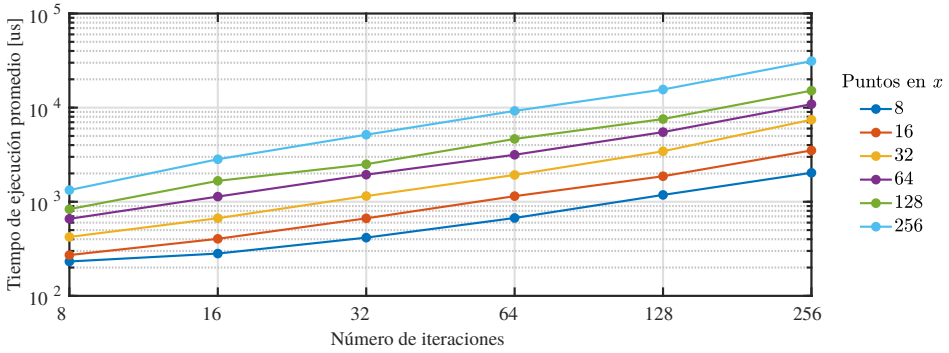


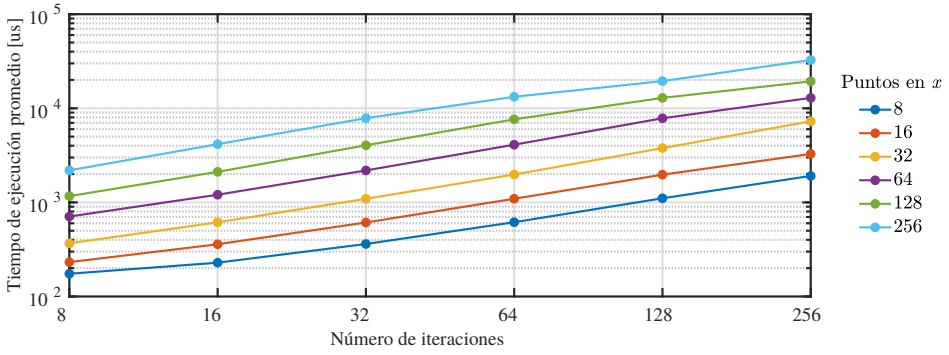
Figura 3-10.: Vista superior del error relativo obtenido con el sistema implementado en la ZedBoard para mallas de 256 puntos en el eje x y (a) 8, (b) 16, (c) 32, (d) 64, (e) 128, y (f) 256 iteraciones.

3.1.2. Evaluación en términos del tiempo de ejecución

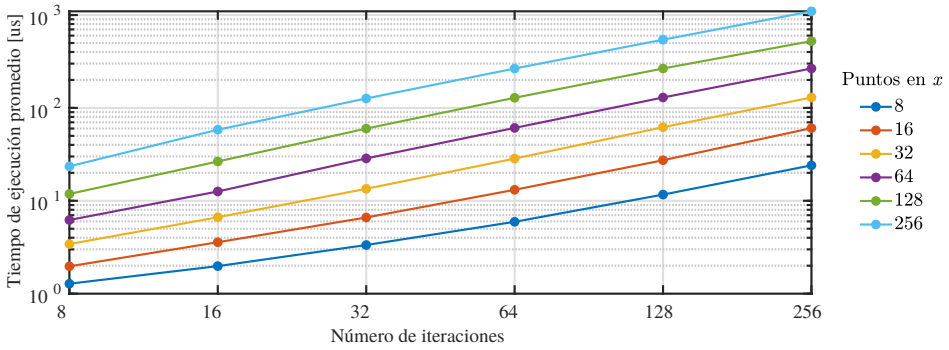
La evaluación del rendimiento en términos del tiempo de ejecución para el sistema implementado en la ZedBoard se realiza para diferentes tamaños de malla. Para tener una referencia de rendimiento, el algoritmo es implementado en C para su ejecución sobre un procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM. La Figura **3-11** muestra la variación del tiempo de ejecución en función del número de iteraciones según la cantidad de puntos en el eje x.



(a)



(b)



(c)

Figura 3-11.: Variación del tiempo de ejecución en microsegundos en el procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM en función del número de iteraciones según la cantidad de puntos en el eje x para (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento en segundos, y (c) procesamiento.

En la Tabla **3-1** se muestra el tiempo total en microsegundos para la ejecución en CPU, en función del número de iteraciones según la cantidad de puntos en el eje x.

Tabla 3-1.: Tiempo de ejecución total en microsegundos empleando el procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	231.97	281.45	415.62	672.13	1181.78	2030.86
	16	271.42	403.73	666.29	1146.81	1866.23	3509.87
	32	422.59	668.21	1149.18	1925.94	3438.70	7446.39
	64	658.33	1134.58	1938.74	3154.53	5494.16	10882.17
	128	833.73	1667.06	2503.48	4650.17	7562.89	15150.75
	256	1330.57	2827.56	5154.28	9229.99	15577.18	31209.65

En la Tabla **3-2** se muestra el tiempo en microsegundos para procesamiento y almacenamiento en CPU, en función del número de iteraciones según los puntos en el eje x.

Tabla 3-2.: Tiempo de procesamiento y almacenamiento en microsegundos empleando el procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	174.54	228.72	361.32	615.67	1106	1911.2
	16	231.74	358.94	611.49	1093.5	1970.4	3272.8
	32	367.2	614.49	1092.4	1979.2	3774.1	7283.2
	64	707.5	1206.1	2184.7	4104.2	7829.6	12900
	128	1165.3	2111.5	4043.2	7637.4	12897	19324
	256	2189.5	4154.8	7850.4	13245	19448	32575

En la Tabla **3-3** se muestra el tiempo en microsegundos para procesamiento en CPU, en función del número de iteraciones según la cantidad de puntos en el eje x.

Tabla 3-3.: Tiempo de procesamiento en microsegundos empleando el procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.28	1.98	3.34	5.94	11.66	24.07
	16	1.97	3.58	6.61	13.13	27.33	60.58
	32	3.42	6.65	13.45	28.47	61.97	129.3
	64	6.21	12.6	28.59	60.9	129.46	265.61
	128	11.85	26.5	59.9	128.53	265.64	523.89
	256	23.49	58.27	126.37	265.42	542.58	1096.3

Adicionalmente, el algoritmo es implementado en C para su ejecución sobre uno de los núcleos de ARM Cortex-A9. La Figura 3-12 muestra la variación del tiempo de ejecución en función del número de iteraciones según la cantidad de puntos en el eje x.

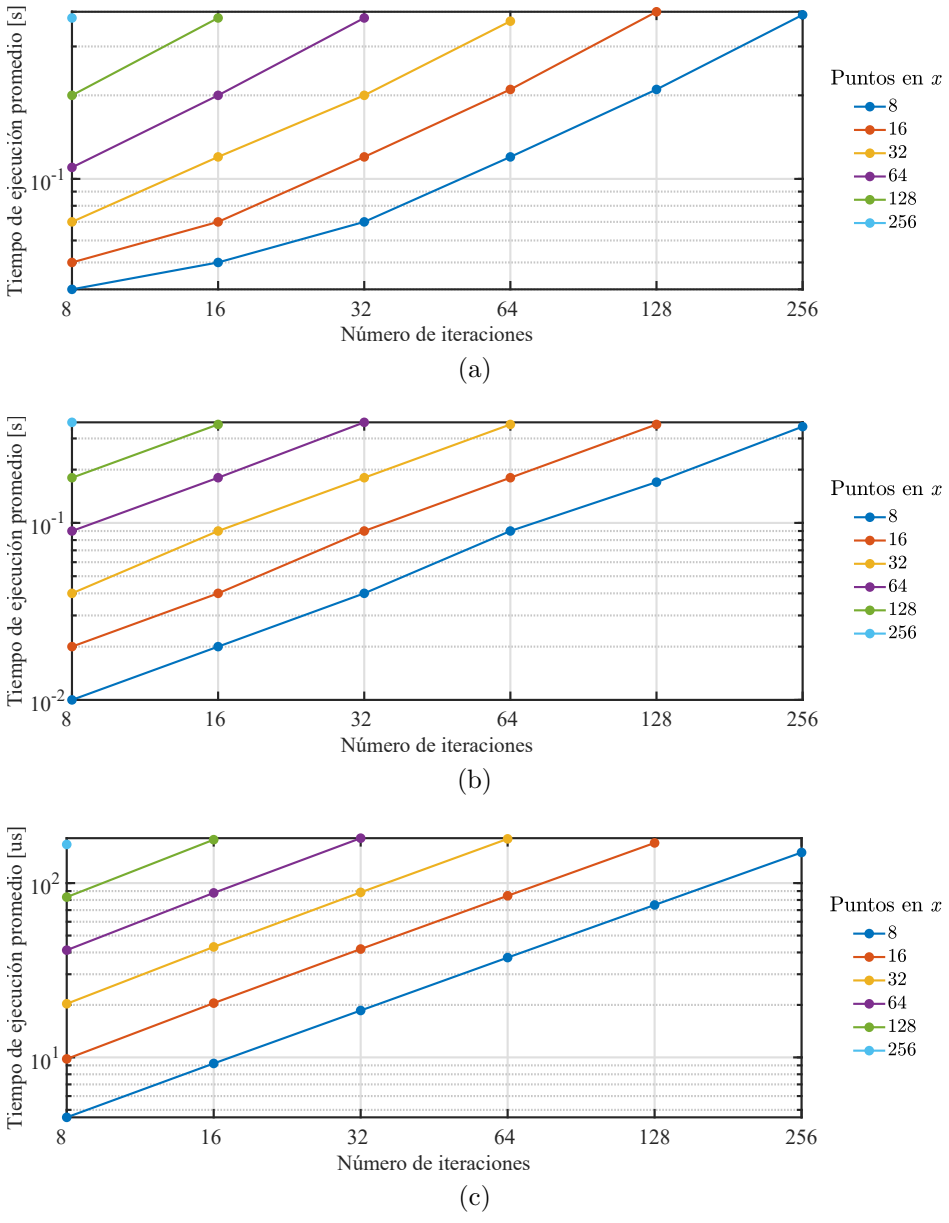


Figura 3-12.: Variación del tiempo de ejecución en el procesador ARM, en función del número de iteraciones según la cantidad de puntos en el eje x para (a) configuración, procesamiento y almacenamiento en segundos (b) procesamiento y almacenamiento en segundos, y (c) procesamiento en microsegundos.

En la Tabla 3-4 se muestra el tiempo de ejecución en segundos en el procesador ARM, en función del número de iteraciones según la cantidad de puntos en el eje x. Debido al tamaño

de la memoria, no es posible ejecutar el programa para todos los tamaños de malla de la tabla.

Tabla 3-4.: Tiempo de ejecución total en segundos empleando el procesador ARM a 667 MHz, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.04	0.05	0.07	0.12	0.21	0.39
	16	0.05	0.07	0.12	0.21	0.40	
	32	0.07	0.12	0.20	0.37		
	64	0.11	0.20	0.38			
	128	0.20	0.38				
	256	0.38					

En la Tabla **3-5** se muestra el tiempo en segundos para procesamiento y almacenamiento en el procesador ARM, en función del número de iteraciones según los puntos en el eje x.

Tabla 3-5.: Tiempo de procesamiento y almacenamiento en segundos empleando el procesador ARM, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.01	0.02	0.04	0.09	0.17	0.35
	16	0.02	0.04	0.09	0.18	0.36	
	32	0.04	0.09	0.18	0.36		
	64	0.09	0.18	0.37			
	128	0.18	0.36				
	256	0.37					

En la Tabla **3-6** se muestra el tiempo en microsegundos para procesamiento en el procesador ARM, en función del número de iteraciones según la cantidad de puntos en el eje x.

Tabla 3-6.: Tiempo de procesamiento en microsegundos empleando el procesador ARM, en función del número de iteraciones según la cantidad de puntos en el eje x.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	4.53	9.23	18.58	37.38	74.89	149.84
	16	9.80	20.48	41.84	84.57	170.01	
	32	20.31	43.03	88.47	179.35		
	64	41.23	87.84	180.99			
	128	83.1	177.41				
	256	166.64					

La evaluación de desempeño de la arquitectura A_1 en términos del tiempo de ejecución se realiza para diferentes combinaciones de tamaño de malla en función del número de iteraciones según la cantidad de puntos en el eje x, como se muestra en la Figura **3-13**.

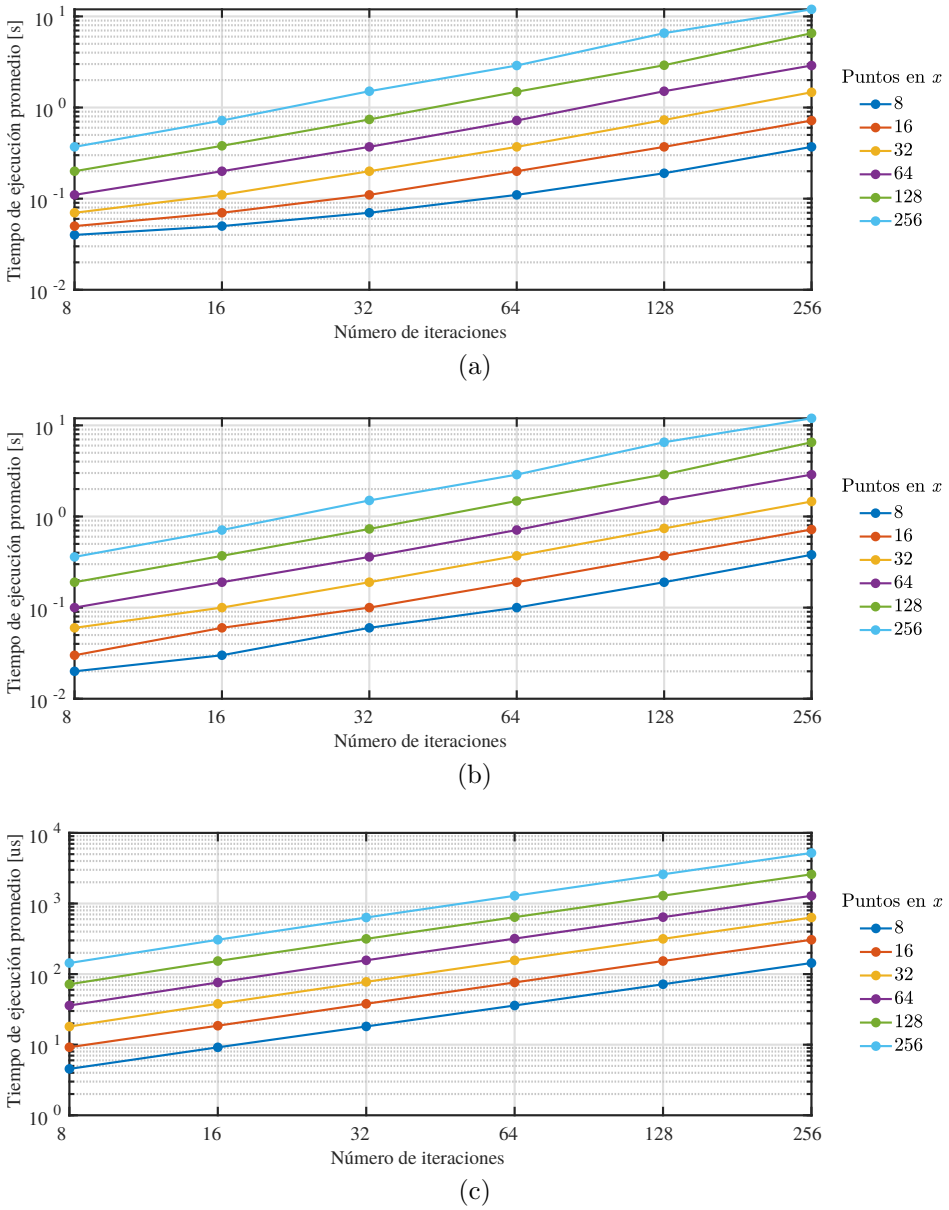


Figura 3-13.: Variación del tiempo en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento en segundos (b) procesamiento y almacenamiento en segundos, y (c) procesamiento en microsegundos.

En la Tabla **3-7** se muestra el tiempo total promedio en segundos empleando la arquitectura A_1 para diferentes combinaciones de $J \times N$.

Tabla 3-7.: Tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.04	0.05	0.07	0.11	0.19	0.37
	16	0.05	0.07	0.11	0.20	0.37	0.72
	32	0.07	0.11	0.20	0.37	0.73	1.47
	64	0.11	0.20	0.37	0.72	1.51	2.89
	128	0.20	0.38	0.74	1.49	2.91	6.54
	256	0.37	0.72	1.51	2.89	6.55	11.95

En la Tabla **3-8** se muestra el tiempo promedio en segundos medido desde que se envía la señal de inicio de procesamiento hasta que se realiza el almacenamiento de los resultados en archivo.

Tabla 3-8.: Tiempo promedio en segundos medido desde la señal de inicio de procesamiento hasta el almacenamiento de los resultados en archivo para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1.

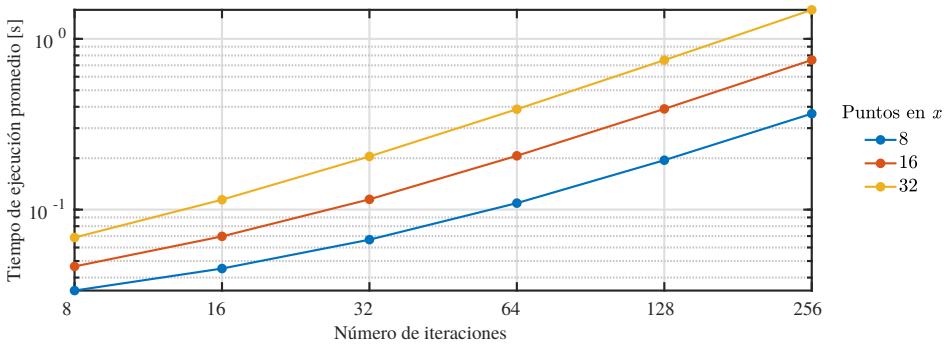
		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.02	0.03	0.06	0.10	0.19	0.38
	16	0.03	0.06	0.10	0.19	0.37	0.72
	32	0.06	0.10	0.19	0.37	0.74	1.46
	64	0.10	0.19	0.36	0.71	1.50	2.88
	128	0.19	0.37	0.73	1.48	2.89	6.52
	256	0.36	0.71	1.50	2.88	6.53	11.95

En la Tabla **3-9** se muestra el tiempo promedio en microsegundos medido desde el envío de la instrucción de inicio hasta la recepción de la indicación de finalización.

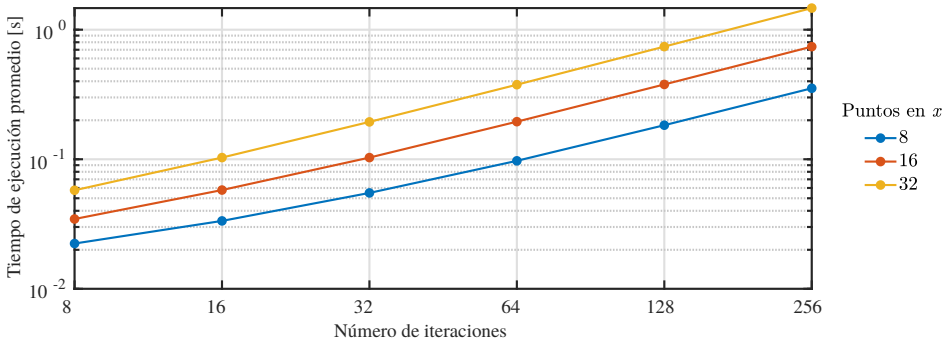
Tabla 3-9.: Tiempo promedio en microsegundos medido de la ejecución del algoritmo basado en esténcil para diferentes combinaciones de $J \times N$ empleando A_1.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	4.54	9.17	18.13	35.89	71.78	143.53
	16	9.20	18.57	37.96	76.16	153.17	306.60
	32	18.07	37.93	77.61	156.76	315.60	633.06
	64	35.90	76.12	156.78	318.09	640.68	1285.74
	128	71.78	153.16	315.59	640.62	1291.05	2591.44
	256	143.47	306.61	633.05	1285.74	2591.44	5202.57

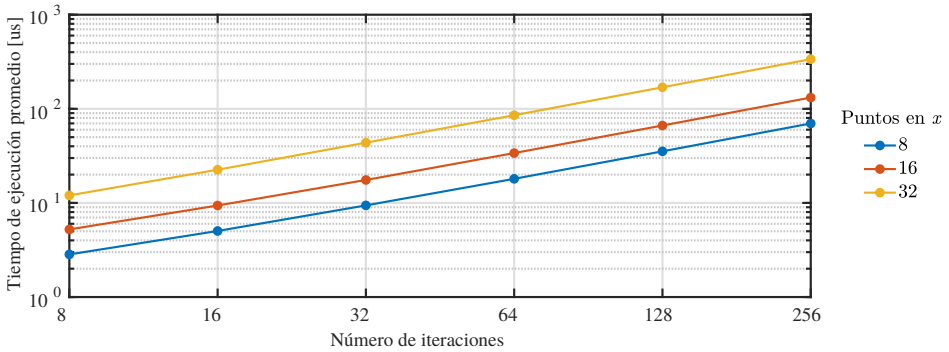
Para A_2 la variación del tiempo de ejecución en función del número de iteraciones según la cantidad de puntos en el eje x se muestra en la Figura **3-14**. Debido a la cantidad de elementos de procesamiento el valor máximo evaluado para la cantidad de puntos en x es 32.



(a)



(b)



(c)

Figura 3-14.: Variación del tiempo en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento en segundos (b) procesamiento y almacenamiento en segundos, y (c) procesamiento en microsegundos.

En la Tabla **3-10** se muestra el tiempo total promedio en segundos empleando la arquitectura A_2 para diferentes combinaciones de $J \times N$.

Tabla 3-10.: Tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.03	0.05	0.07	0.11	0.19	0.36
	16	0.05	0.07	0.11	0.21	0.39	0.75
	32	0.07	0.11	0.20	0.39	0.75	1.48

En la Tabla **3-11** se muestra el tiempo promedio en segundos medido desde que se envía la señal de inicio de procesamiento hasta que se realiza el almacenamiento de los resultados en archivo.

Tabla 3-11.: Tiempo promedio en segundos medido desde la señal de inicio de procesamiento hasta el almacenamiento de los resultados en archivo para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.02	0.03	0.05	0.10	0.18	0.35
	16	0.03	0.06	0.10	0.20	0.38	0.74
	32	0.06	0.10	0.19	0.38	0.74	1.47

En la Tabla **3-12** se muestra el tiempo promedio en microsegundos medido desde que se envía la instrucción de inicio hasta que se recibe la indicación de finalización.

Tabla 3-12.: Tiempo promedio en microsegundos medido de la ejecución del algoritmo basado en esténcil para diferentes combinaciones de $J \times N$ empleando A_2.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	2.84	5.03	9.41	18.03	35.34	69.89
	16	5.23	9.39	17.54	33.88	66.47	131.68
	32	12.02	22.53	43.67	85.49	169.40	337.10

Para A_3 la variación del tiempo de ejecución en función del número de iteraciones según la cantidad de puntos en el eje x se muestra en la Figura **3-15**. Debido a la cantidad de elementos de procesamiento el valor máximo evaluado para la cantidad de puntos en x es 32.

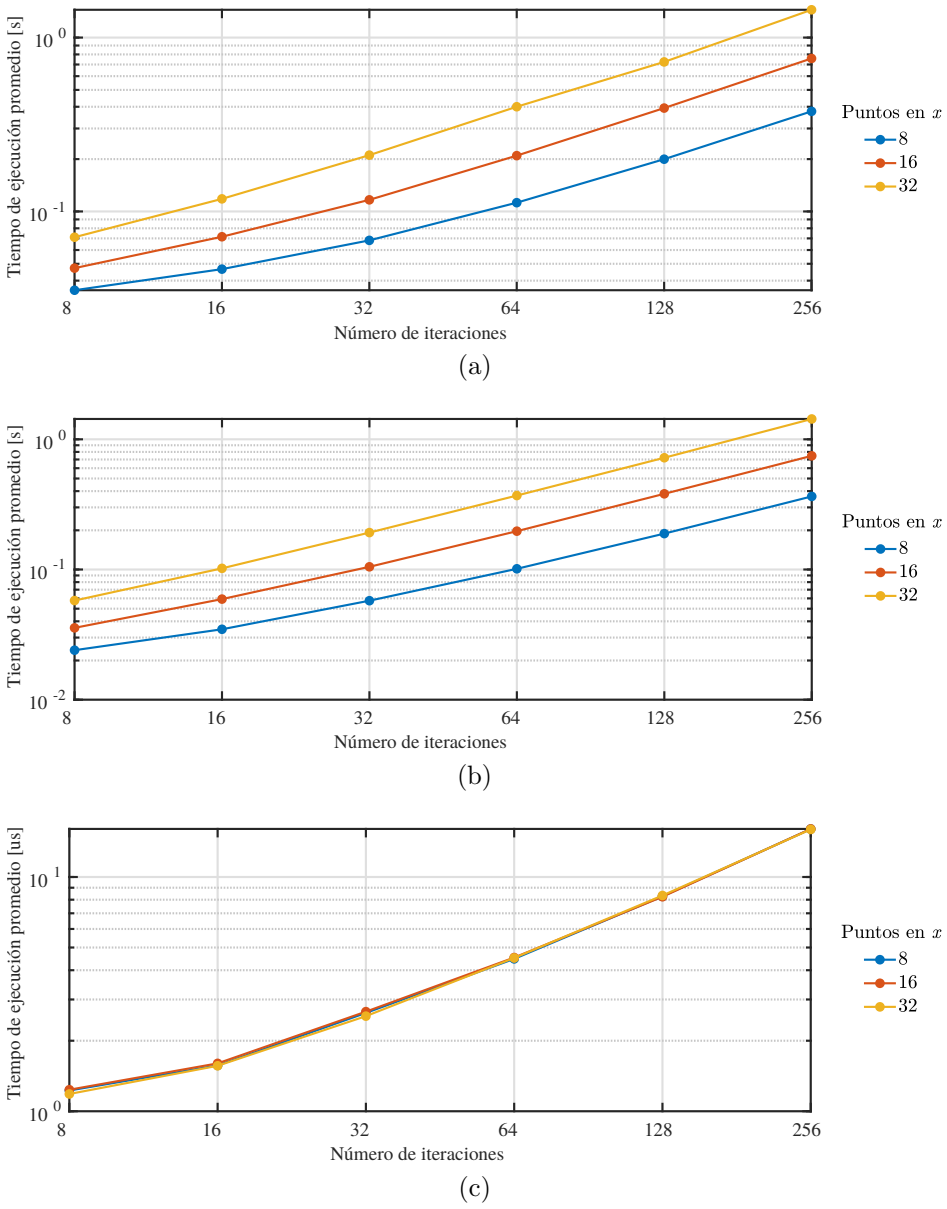


Figura 3-15.: Variación del tiempo en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento en segundos (b) procesamiento y almacenamiento en segundos, y (c) procesamiento en microsegundos.

En la Tabla **3-13** se muestra el tiempo total promedio en segundos empleando la arquitectura A_3 para diferentes combinaciones de $J \times N$.

Tabla 3-13.: Tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.04	0.05	0.07	0.11	0.20	0.38
	16	0.05	0.07	0.12	0.21	0.39	0.76
	32	0.07	0.12	0.21	0.40	0.72	1.44

En la Tabla **3-14** se muestra el tiempo promedio en segundos medido desde que se envía la señal de inicio de procesamiento hasta que se realiza el almacenamiento de los resultados en archivo.

Tabla 3-14.: Tiempo promedio en segundos medido desde la señal de inicio de procesamiento hasta el almacenamiento de los resultados en archivo para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.02	0.03	0.06	0.10	0.19	0.36
	16	0.04	0.06	0.10	0.20	0.38	0.75
	32	0.06	0.10	0.19	0.37	0.72	1.43

En la Tabla **3-15** se muestra el tiempo promedio en microsegundos medido desde que se envía la instrucción de inicio hasta que se recibe la indicación de finalización.

Tabla 3-15.: Tiempo promedio en microsegundos medido de la ejecución del algoritmo basado en esténcil para diferentes combinaciones de $J \times N$ empleando A_3.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.23	1.57	2.63	4.47	8.27	16.02
	16	1.24	1.60	2.66	4.53	8.24	16.04
	32	1.18	1.56	2.55	4.51	8.34	15.96

La variación de la aceleración para la arquitectura A_1 con relación al procesador Intel en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-16**.

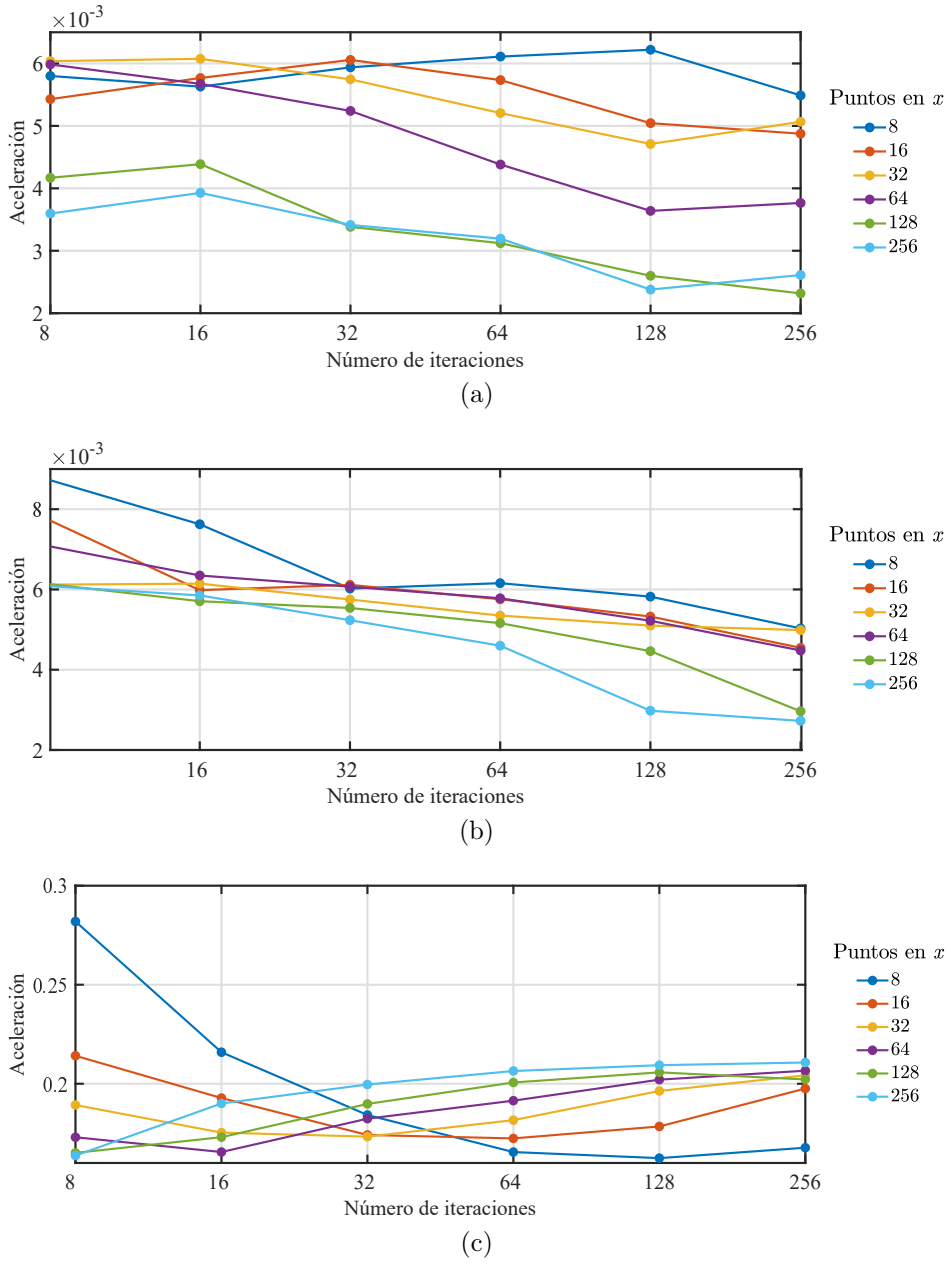


Figura 3-16.: Variación de la aceleración para la arquitectura A_1 con relación al procesador Intel en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-16** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

Tabla 3-16.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0058	0.0056	0.0059	0.0061	0.0062	0.0055
	16	0.0054	0.0058	0.0061	0.0057	0.0050	0.0049
	32	0.0060	0.0061	0.0057	0.0052	0.0047	0.0051
	64	0.0060	0.0057	0.0052	0.0044	0.0036	0.0038
	128	0.0042	0.0044	0.0034	0.0031	0.0026	0.0023
	256	0.0036	0.0039	0.0034	0.0032	0.0024	0.0026

En la Tabla **3-17** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

Tabla 3-17.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0087	0.0076	0.0060	0.0062	0.0058	0.0050
	16	0.0077	0.0060	0.0061	0.0058	0.0053	0.0045
	32	0.0061	0.0061	0.0057	0.0053	0.0051	0.0050
	64	0.0071	0.0063	0.0061	0.0058	0.0052	0.0045
	128	0.0061	0.0057	0.0055	0.0052	0.0045	0.0030
	256	0.0061	0.0059	0.0052	0.0046	0.0030	0.0027

En la Tabla **3-18** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

Tabla 3-18.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.28	0.22	0.18	0.17	0.16	0.17
	16	0.21	0.19	0.17	0.17	0.18	0.20
	32	0.19	0.18	0.17	0.18	0.20	0.20
	64	0.17	0.17	0.18	0.19	0.20	0.21
	128	0.17	0.17	0.19	0.20	0.21	0.20
	256	0.16	0.19	0.20	0.21	0.21	0.21

La variación de la aceleración para la arquitectura A_2 con relación al procesador Intel en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-17**.

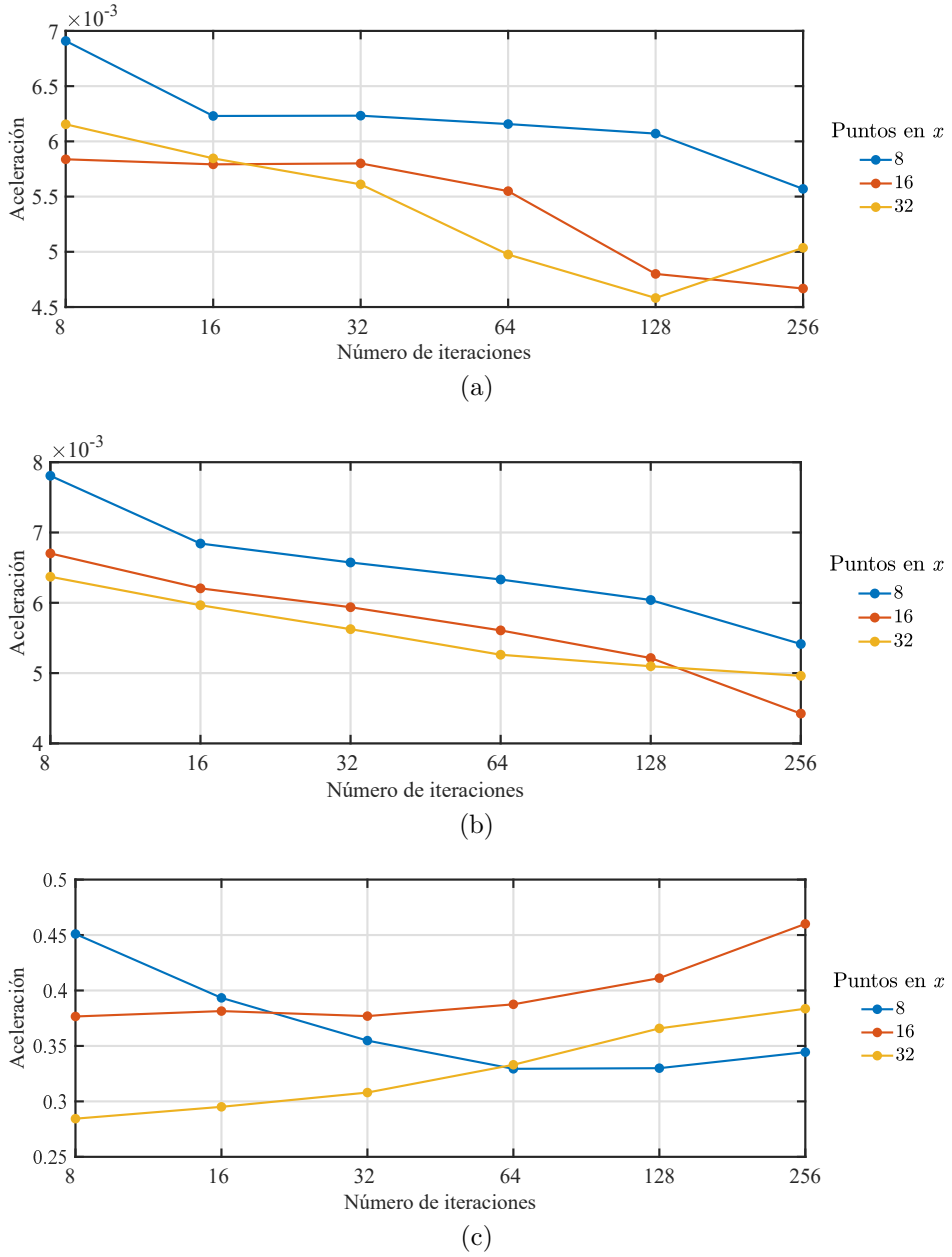


Figura 3-17.: Variación de la aceleración en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-19** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

Tabla 3-19.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0069	0.0062	0.0062	0.0062	0.0061	0.0056
	16	0.0058	0.0058	0.0058	0.0055	0.0048	0.0047
	32	0.0062	0.0058	0.0056	0.0050	0.0046	0.0050

En la Tabla **3-20** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

Tabla 3-20.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0078	0.0068	0.0066	0.0063	0.0060	0.0054
	16	0.0067	0.0062	0.0059	0.0056	0.0052	0.0044
	32	0.0064	0.0060	0.0056	0.0053	0.0051	0.0050

En la Tabla **3-21** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

Tabla 3-21.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.45	0.39	0.35	0.33	0.33	0.34
	16	0.38	0.38	0.38	0.39	0.41	0.46
	32	0.28	0.29	0.31	0.33	0.37	0.38

La variación de la aceleración para la arquitectura A_3 con relación al procesador Intel en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-18**.

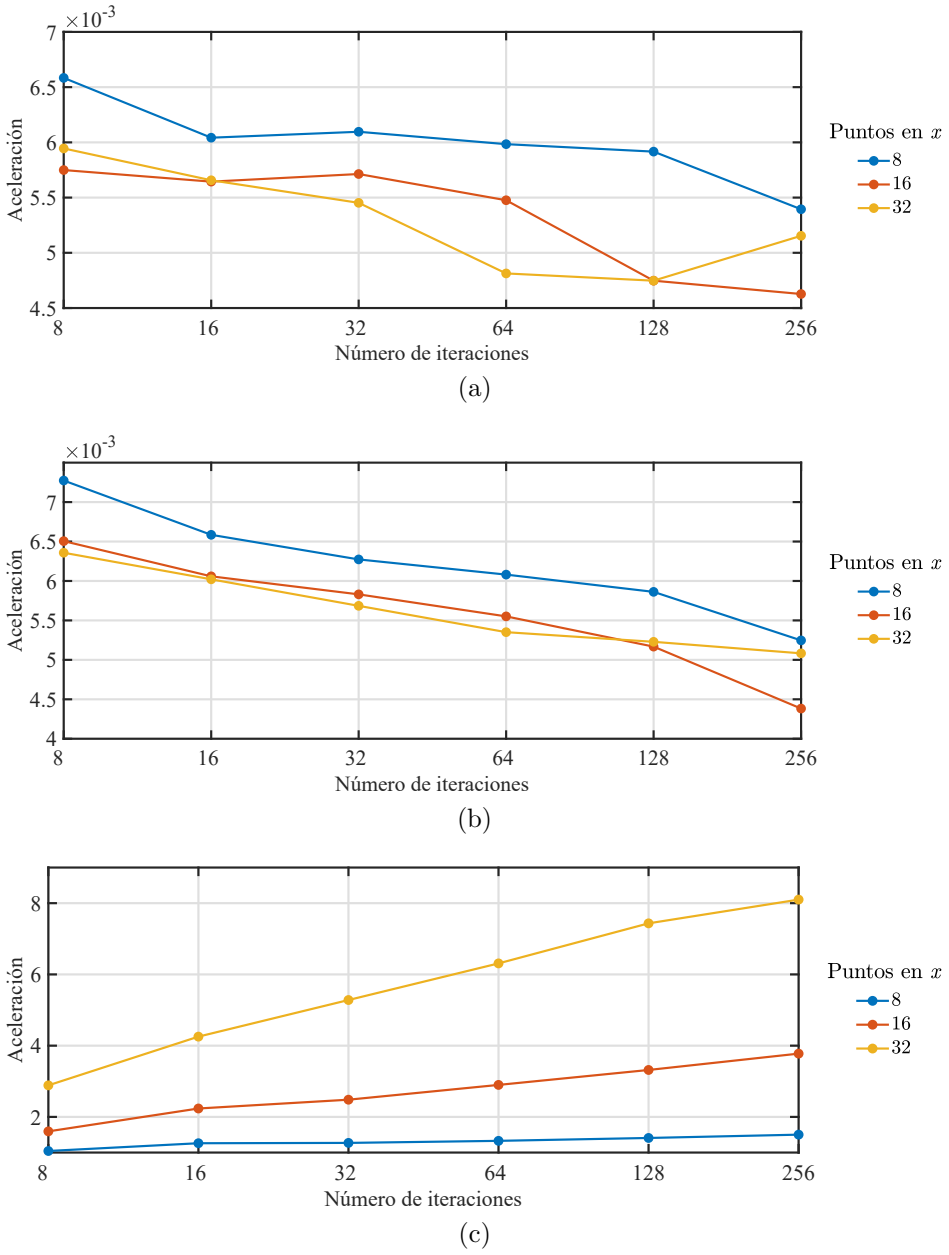


Figura 3-18.: Variación de la aceleración en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-22** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

Tabla 3-22.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0066	0.0060	0.0061	0.0060	0.0059	0.0054
	16	0.0057	0.0056	0.0057	0.0055	0.0047	0.0046
	32	0.0059	0.0057	0.0055	0.0048	0.0047	0.0052

En la Tabla **3-23** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

Tabla 3-23.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.0073	0.0066	0.0063	0.0061	0.0059	0.0052
	16	0.0065	0.0061	0.0058	0.0056	0.0052	0.0044
	32	0.0064	0.0060	0.0057	0.0054	0.0052	0.0051

En la Tabla **3-24** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

Tabla 3-24.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador Intel.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.04	1.26	1.27	1.33	1.41	1.50
	16	1.59	2.24	2.48	2.90	3.32	3.78
	32	2.89	4.25	5.28	6.31	7.43	8.10

La variación de la aceleración para la arquitectura A_1 con relación al procesador ARM en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-19**.

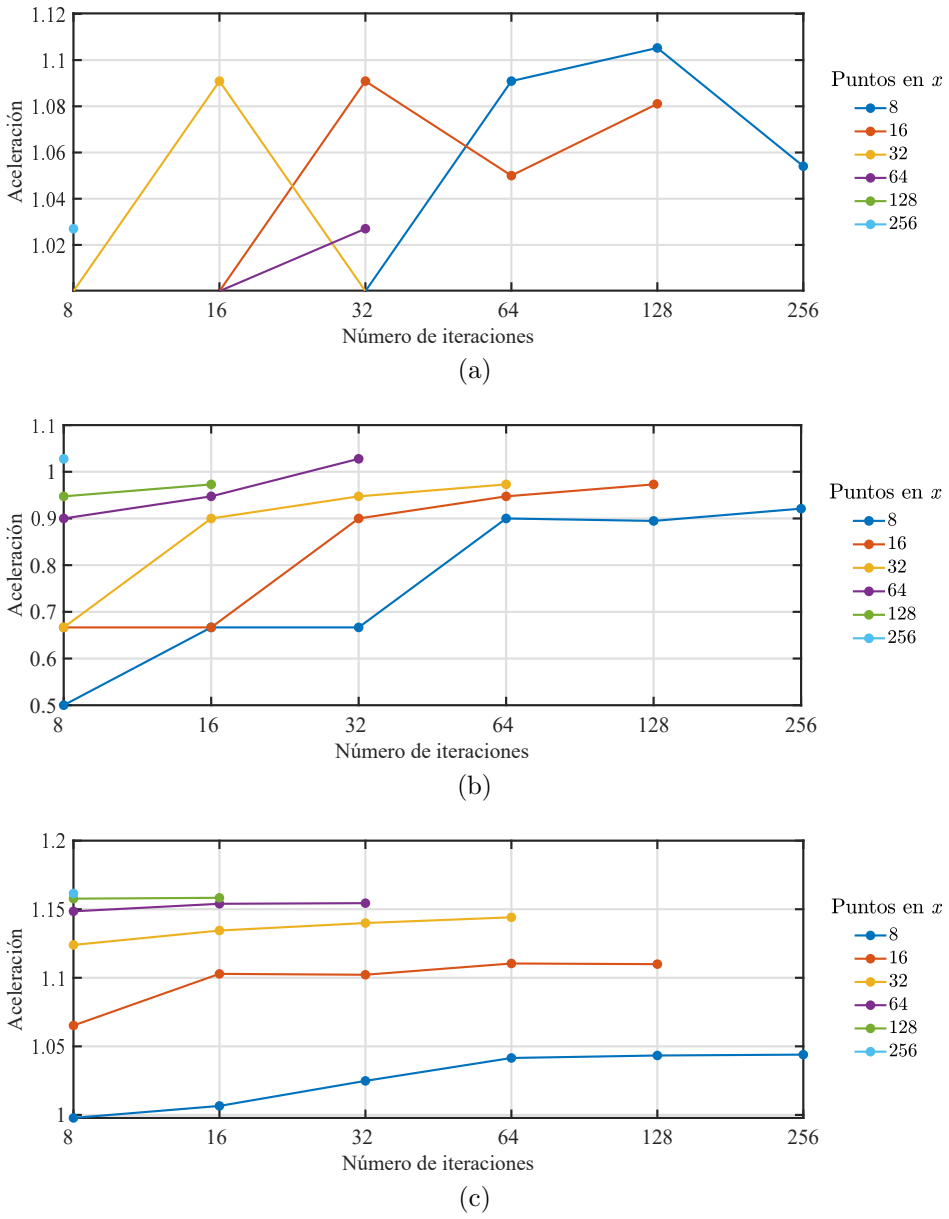


Figura 3-19.: Variación de la aceleración en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-25** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

Tabla 3-25.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.00	1.00	1.00	1.10	1.11	1.05
	16	1.00	1.00	1.10	1.05	1.08	
	32	1.00	1.10	1.00	1.00		
	64	1.00	1.00	1.03			
	128	1.00	1.00				
	256	1.03					

En la Tabla **3-26** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

Tabla 3-26.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.50	0.67	0.67	0.90	0.89	0.92
	16	0.67	0.67	0.90	0.95	0.97	
	32	0.67	0.90	0.95	0.97		
	64	0.90	0.95	1.03			
	128	0.95	0.97				
	256	1.03					

En la Tabla **3-27** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

Tabla 3-27.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.28	0.22	0.18	0.17	0.16	0.17
	16	0.21	0.19	0.17	0.17	0.18	0.20
	32	0.19	0.18	0.17	0.18	0.20	0.20
	64	0.17	0.17	0.18	0.19	0.20	0.21
	128	0.17	0.17	0.19	0.20	0.21	0.20
	256	0.16	0.19	0.20	0.21	0.21	0.21

La variación de la aceleración para la arquitectura A_2 con relación al procesador ARM en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-20**.

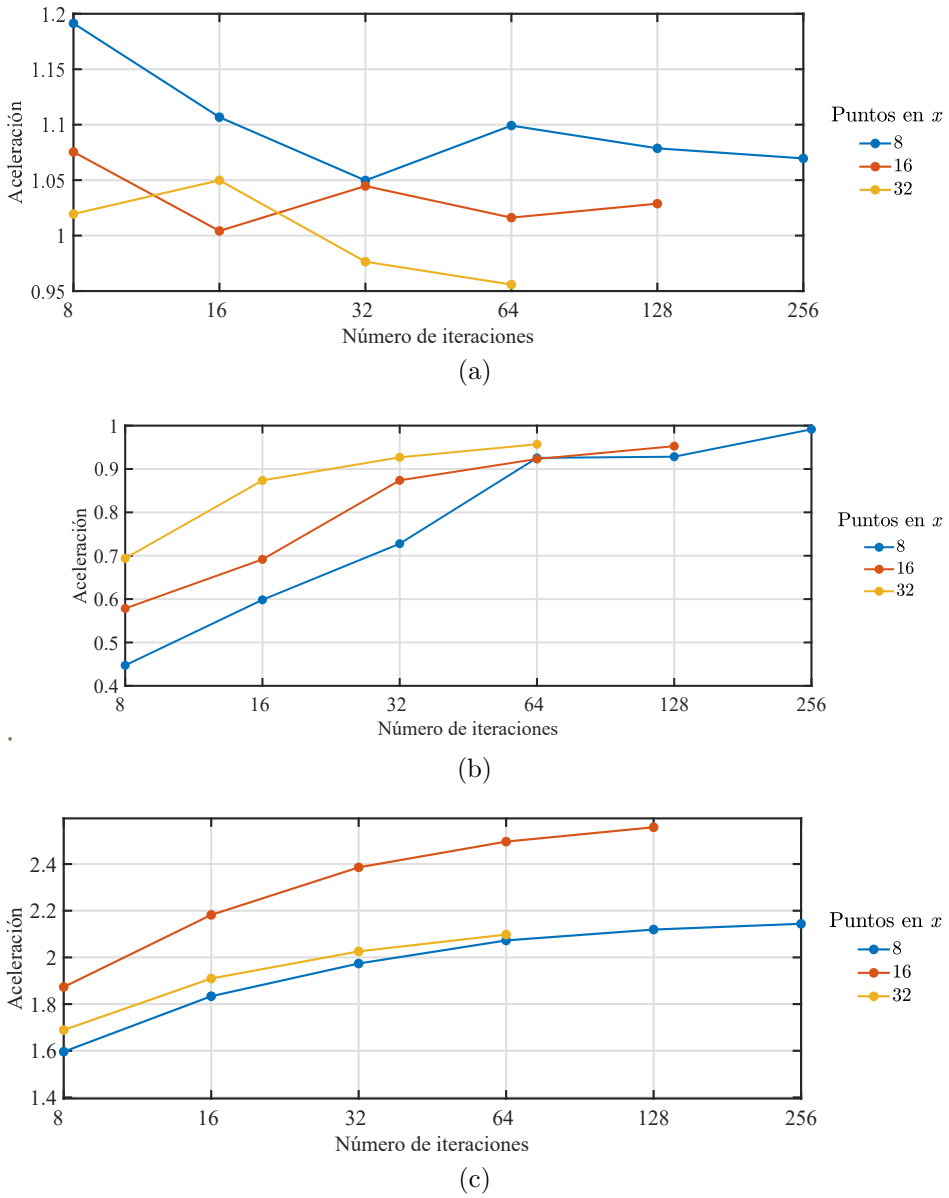


Figura 3-20.: Variación de la aceleración en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-28** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

Tabla 3-28.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.19	1.11	1.05	1.10	1.08	1.07
	16	1.08	1.00	1.04	1.02	1.03	
	32	1.02	1.05	0.98	0.96		

En la Tabla **3-29** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

Tabla 3-29.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.45	0.60	0.73	0.93	0.93	0.99
	16	0.58	0.70	0.87	0.92	0.95	
	32	0.70	0.87	0.93	0.96		

En la Tabla **3-30** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

Tabla 3-30.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_2 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.60	1.83	1.97	2.07	2.12	2.14
	16	1.87	2.18	2.39	2.50	2.56	
	32	1.69	1.91	2.03	2.10		

La variación de la aceleración para la arquitectura A_3 con relación al procesador ARM en función de la cantidad de puntos en x según el número de iteraciones se muestra en la Figura **3-21**.

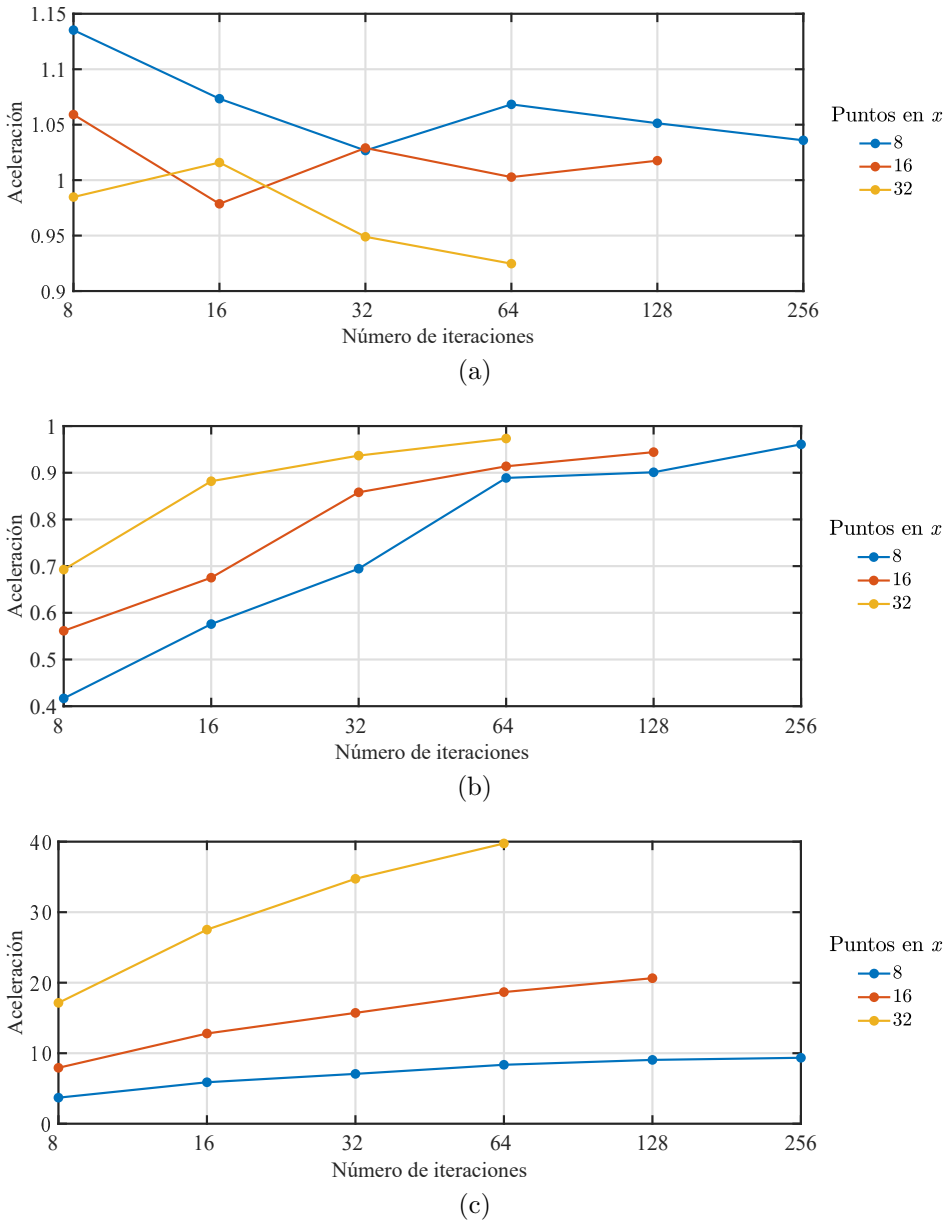


Figura 3-21.: Variación de la aceleración en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

En la Tabla **3-31** se muestra la aceleración obtenida de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

Tabla 3-31.: Aceleración de acuerdo con el tiempo total promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	1.14	1.07	1.03	1.07	1.05	1.04
	16	1.06	0.98	1.03	1.00	1.02	
	32	0.98	1.02	0.95	0.92		

En la Tabla **3-32** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

Tabla 3-32.: Aceleración de acuerdo con el tiempo de procesamiento y almacenamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	0.42	0.58	0.69	0.89	0.90	0.96
	16	0.56	0.68	0.86	0.91	0.94	
	32	0.69	0.88	0.94	0.97		

En la Tabla **3-33** se muestra la aceleración obtenida de acuerdo con el tiempo de procesamiento promedio en segundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

Tabla 3-33.: Aceleración de acuerdo con el tiempo de procesamiento promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_3 con relación al procesador ARM.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	3.69	5.88	7.07	8.36	9.05	9.35
	16	7.93	12.79	15.72	18.67	20.64	
	32	17.14	27.52	34.74	39.75		

Los recursos de hardware utilizados en síntesis se muestran en la Tabla **3-34**.

Tabla 3-34.: Recursos de hardware utilizados en síntesis.

Recurso	Disponibles	A_1	$A_{2_{8\times}}$	$A_{2_{16\times}}$	$A_{2_{32\times}}$	$A_{3_{8\times}}$	$A_{3_{16\times}}$	$A_{3_{32\times}}$
LUT	53200	2691	8077	20189	42054	11758	23128	45454
Flip Flop	106400	1761	1393	1653	2184	1890	1954	3063
BRAM	140	64	64	64	64	0	3	11
DSP48	220	4	24	56	120	24	56	120

Los recursos de hardware utilizados en implementación se muestran en la Tabla **3-35**.

Tabla 3-35.: Recursos de hardware utilizados en implementación. Todo el sistema incluye módulos de procesamiento.

Recurso	Disponibles	A_1	$A_{2_{8\times}}$	$A_{2_{16\times}}$	$A_{2_{32\times}}$	$A_{3_{8\times}}$	$A_{3_{16\times}}$	$A_{3_{32\times}}$
LUT	53200	2551	7828	19984	41670	11556	22869	45004
Flip Flop	106400	1618	1311	1649	2439	1885	2010	3403
BRAM	140	64	64	64	64	0	3	11
DSP48	220	4	24	56	120	24	56	120

El consumo de potencia para las arquitecturas implementadas en la Zedboard se muestra en la Tabla **3-36**.

Tabla 3-36.: Consumo de potencia para las arquitecturas implementadas en la Zedboard.

Recurso	A_1	$A_{2_{8\times}}$	$A_{2_{16\times}}$	$A_{2_{32\times}}$	$A_{3_{8\times}}$	$A_{3_{16\times}}$	$A_{3_{32\times}}$
Potencia Estática	0.152	0.158	0.171	0.198	0.149	0.163	0.203
Relojes	0.007	0.008	0.010	0.015	0.012	0.019	0.023
Señales	0.038	0.122	0.329	0.644	0.188	0.385	0.833
Lógica	0.019	0.077	0.193	0.428	0.123	0.275	0.588
BRAM	0.112	0.115	0.115	0.114	-	0.013	0.048
DSP	0.003	0.022	0.050	0.109	0.023	0.055	0.117
PS	1.524	1.530	1.530	1.530	1.530	1.530	1.530
Potencia Dinámica	1.704	1.874	2.228	2.839	1.877	2.276	3.140
Potencia Total	1.856	2.032	2.399	3.037	2.026	2.439	3.343

3.2. Análisis de resultados

La comparación de las arquitecturas implementadas permite observar cómo las variaciones propuestas en el camino de datos influyen en la reducción del tiempo de ejecución del algoritmo. Se muestra que la estructura de la memoria es uno de los aspectos del camino de datos que más influye en el rendimiento. Además, se muestra que el uso del arreglo de registros

introducido en el camino de datos permite tomar ventaja de la localidad espacial y temporal de los datos, reduciendo la cantidad de recursos utilizados y de operaciones de transferencia de memoria.

La evaluación de rendimiento se realiza en términos de los resultados numéricos y el tiempo de ejecución para la arquitectura base y las dos arquitecturas paralelas. En los resultados numéricos se observa que el error es mayor en la medida que aumenta la cantidad de iteraciones. Esto se debe a la acumulación del error por representación numérica y el tipo de aproximación utilizado. En el caso de 256 iteraciones el máximo del error relativo no supera 8×10^{-6}.

La evaluación en términos de tiempo de ejecución se realiza con A_1, A_2 y A_3 para mallas de 8, 16 y 32 puntos en el eje x, variando la cantidad de iteraciones en potencias de 2 hasta 256. La aceleración obtenida con las arquitecturas paralelas es calculada con relación a un procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM, a un núcleo de ARM de la SoC FPGA y a la arquitectura A_1.

La comparación de la arquitectura A_1 con la ejecución secuencial sobre el procesador Intel muestra una aceleración inferior a 1 para la mayoría de los casos evaluados. La arquitectura A_2 no muestra una mejora significa del rendimiento, presentando valores de aceleración similares o cercanos a los de la arquitectura A_1. Con la arquitectura A_3 se logra superar el desempeño del procesador tomando en cuenta únicamente el tiempo de procesamiento del algoritmo, obteniendo una aceleración de 8.1× para una malla de 32×256.

La comparación de la arquitectura A_1 con la ejecución secuencial sobre el procesador ARM muestra una aceleración cercana a 1 para todos los casos evaluados teniendo en cuenta el tiempo de ejecución total. Considerando que la ejecución del algoritmo con A_1 es secuencial se identifica que la mejora del rendimiento se logra a nivel del camino de datos debido al diseño de los elementos de proceso.

La arquitectura A_3 es la que presenta mejor desempeño, alcanzando una aceleración de 82.59× con relación al microprocesador ARM, 64.03× relación a A_1 y 8.04× con relación a la CPU. El rendimiento para la arquitectura A_2 puede ser mejorado si en la memoria RAM se almacenan únicamente los resultados de la última iteración, alcanzando una aceleración de 1.48 con relación a A_3.

4. Aceleración del algoritmo basado en esténcil para la ecuación de Laplace bidimensional

La aceleración de la ejecución del Algoritmo 5 se logra utilizando directivas de optimización. Para la aplicación y el análisis de las directivas de optimización se definen las etiquetas de los ciclos como se muestra en Algoritmo 6.

Algoritmo de descripción 6: Etiquetado del los ciclos de la operación con esténcil para la aplicación y análisis de las directivas de optimización.

Input: valores iniciales, condiciones de frontera, tamaño de malla, número de iteraciones
Output: temperatura en la iteración N

```
1  dominio_t: for n ← 0 to N − 1 do
2  |  dominio_y: for j ← 1 to J − 1 do
3  |  |  dominio_x: for i ← 1 to I − 1 do
4  |  |  |  s1 ← u[j ∗ X_I + i + 1]
5  |  |  |  s2 ← u[j ∗ X_I + i − 1]
6  |  |  |  s3 ← u[(j + 1) ∗ X_I + i]
7  |  |  |  s4 ← u[(j − 1) ∗ X_I + i]
8  |  |  |  v[j ∗ X_I + i] ← 0.25 (s1 + s2 + s3 + s4)
9  |  |  end
10 |  end
11 |  copia_y: for j ← 1 to J − 1 do
12 |  |  copia_x: for i ← 1 to I − 1 do
13 |  |  |  u[j ∗ X_I + i] ← v[j ∗ X_I + i]
14 |  |  end
15 |  end
16 end
```

Las directivas de optimización utilizadas son *HLS PIPELINE, HLS UNROLL y ARRAY PARTITION*, las cuales definen principalmente la forma en que es realizada la paralelización del algoritmo. Se definen vectores locales para que el procesamiento se realice con los datos

almacenados en memoria *on-chip*. Inicialmente, se analiza la aplicación de las directivas en el bucle interno definido como ***dominio_x***. En Algoritmo 7 se muestra la utilización de la directiva *HLS PIPELINE*.

Algoritmo de descripción 7: Aplicación de la directiva de optimización *HLS PIPELINE*.

Input: valores iniciales, condiciones de frontera, tamaño de malla, número de iteraciones
Output: temperatura en la iteración N

```
1  memcpy(local_u, u, sizeof(float) * I * J)
2  dominio_t: for n ← 0 to N − 1 do
3      dominio_y: for j ← 1 to J − 1 do
4          dominio_x: for i ← 1 to I − 1 do
5              #pragma HLS PIPELINE
6              s1 ← local_u[j * X_I + i + 1]
7              s2 ← local_u[j * X_I + i − 1]
8              s3 ← local_u[(j + 1) * X_I + i]
9              s4 ← local_u[(j − 1) * X_I + i]
10             local_v[j * X_I + i] ← 0.25 (s1 + s2 + s3 + s4)
11         end
12     end
13     copia_y: for j ← 1 to J − 1 do
14         copia_x: for i ← 1 to I − 1 do
15             local_u[j * X_I + i] ← local_v[j * X_I + i]
16         end
17     end
18 end
19 memcpy(v, local_v, sizeof(float) * I * J)
```

La utilización de la directiva *HLS UNROLL* se realiza de forma similar, colocando en el ciclo etiquetado ***dominio_x*** la expresión *#pragma HLS UNROLL*. La aplicación de la directiva *ARRAY PARTITION* se realiza sobre los vectores locales mediante las siguientes líneas de código. El tipo de partición puede ser definido como *complete, block o cyclic*. El factor de partición de memoria es un valor entero, el cual es utilizado como parámetro para la evaluación del espacio de diseño.

```
#pragma HLS ARRAY_PARTITION variable=local_U cyclic factor=16 dim=1
#pragma HLS ARRAY_PARTITION variable=local_V cyclic factor=16 dim=1
```

Un análisis similar es realizado para el bucle intermedio definido como ***dominio_y***. Finalmente, debido a que la aplicación de directivas por si sola no garantiza en todos los casos una implementación óptima, se proponen dos tipos de transformaciones en el código para reducir la latencia mediante la distribución de la forma en que son procesados los datos. Una

de las transformaciones está relacionada con el ***dominio_t***, la cual consiste en aprovechar la transferencia que se realiza desde el vector *local_v* al vector *local_u* después de aplicar el cálculo con esténcil para toda la malla. De esta forma se reduce el límite superior del ciclo externo a la mitad, como se muestra en Algoritmo 8.

Algoritmo de descripción 8: Transformación del código asociada a *dominio_t*

Input: valores iniciales, condiciones de frontera, tamaño de malla, número de iteraciones
Output: temperatura en la iteración N

```
1  memcpy(local_u, u, sizeof(float) * I * J)
2  dominio_t: for n ← 0 to N/2 − 1 do
3      dominio_yv: for j ← 1 to J − 1 do
4          dominio_xv: for i ← 1 to I − 1 do
5              s1 ← local_u[j * X_I + i + 1]
6              s2 ← local_u[j * X_I + i − 1]
7              s3 ← local_u[(j + 1) * X_I + i]
8              s4 ← local_u[(j − 1) * X_I + i]
9              local_v[j * X_I + i] ← 0.25 (s1 + s2 + s3 + s4)
10         end
11     end
12     dominio_yu: for j ← 1 to J − 1 do
13         dominio_xu: for i ← 1 to I − 1 do
14             s1 ← local_v[j * X_I + i + 1]
15             s2 ← local_v[j * X_I + i − 1]
16             s3 ← local_v[(j + 1) * X_I + i]
17             s4 ← local_v[(j − 1) * X_I + i]
18             local_u[j * X_I + i] ← 0.25 (s1 + s2 + s3 + s4)
19         end
20     end
21 end
22 memcpy(v, local_v, sizeof(float) * I * J)
```

La otra transformación está relacionada con ***dominio_y*** y está desarrollada a partir de la división del dominio de la solución en la dimensión espacial y. Se realiza una división de la malla en tres bloques en el eje y, como se muestra en la Figura **4-1**.

La distribución se realiza de tal forma que la cantidad de filas del bloque B2 es múltiplo de una potencia de 2, considerando que los bloques B1 y B3 tienen una fila menos por incluir condiciones de contorno. El cálculo de los valores correspondientes al bloque B2 se realiza utilizando un nuevo ciclo interno definido como ***dominio_k***. Esta distribución permite la aplicación de las directivas de optimización de tal manera que el tiempo de síntesis, la cantidad de recursos y la latencia son reducidos. Por otra parte la utilización de directivas de

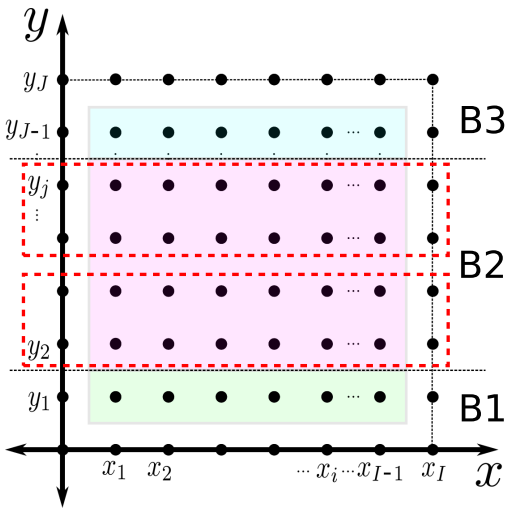


Figura 4-1.: Distribución de bloques para el procesamiento.

partición de memoria permite que los diferentes bloques accedan a los datos correspondientes de manera concurrente ya que son distribuidos en bloques definidos por el factor de partición. Un ejemplo de distribución para una malla de 256×256 se presenta en Algoritmo 9.

4.1. Evaluación de rendimiento del sistema implementado en SoC FPGA

El desempeño del sistema implementado en la ZedBoard es evaluado en términos de los resultados numéricos, el tiempo de ejecución del algoritmo basado en esténcil y el uso de recursos físicos en el FPGA.

4.1.1. Evaluación en términos del resultado numérico

Los resultados numéricos son obtenidos para diferentes tamaños de malla, a partir de condiciones de frontera y valores iniciales definidos como se muestra en (4-1).

$$\begin{cases} u_{xx} + u_{yy} = 0 \\ \quad u = 0, \qquad \forall (x,y) \in \Omega \\ \quad u = 1, \qquad \forall (x,y) \in \partial\Omega \end{cases} \tag{4-1}$$

Algoritmo de descripción 9: Pseudocódigo del algoritmo de computación con esténcil con arreglo bidimensional de 256×256 para N iteraciones.

Input: valores iniciales, condiciones de frontera, $I \times J$, N
Output: temperatura en la iteración N

```
1  memcpy(local_u, u, sizeof(float) * I * J)
2  dominio_t: for n ← 0 to N − 1 do
3      dominio_yB1B3: for j ← 1 to 4 do
4          dominio_xB1B3: for i ← 1 to I − 1 do
5              #pragma HLS PIPELINE
6              s1B1 ← local_u[j * X_I + i + 1] + local_u[j * X_I + i − 1]
7              s2B1 ← local_u[(j + 1) * X_I + i] + local_u[(j − 1) * X_I + i]
8              s1B3 ← local_u[251 + j * X_I + i + 1] + local_u[251 + j * X_I + i − 1]
9              s2B3 ← local_u[251 + (j + 1) * X_I + i] + local_u[251 + (j − 1) * X_I + i]
10             local_v[j * X_I + i] ← 0.25 (s1B1 + s2B1)
               local_v[251 + j * X_I + i] ← 0.25 (s1B3 + s2B3)
11         end
12     end
13     dominio_yB2: for j ← 1 to 4 do
14         dominio_xB2: for i ← 1 to I − 1 do
15             dominio_kB2: for k ← 1 to 63 do
16                 #pragma HLS PIPELINE
17                 s1 ← local_u[(j + 4 * k) * X_I + i + 1] + local_u[(j + 4 * k) * X_I + i − 1]
18                 s2 ← local_u[((j + 4 * k) + 1) * X_I + i] + local_u[((j + 4 * k) − 1) * X_I + i]
19                 local_v[(j + 4 * k) * X_I + i] ← 0.25 (s1 + s2)
20             end
21         end
22     end
23     copia_y: for j ← 1 to J − 1 do
24         copia_x: for i ← 1 to I − 1 do
25             local_u[j * X_I + i] ← local_v[j * X_I + i]
26         end
27     end
28 end
29 memcpy(v, local_v, sizeof(float) * I * J)
```

Los resultados de la aproximación a la solución numérica de la ecuación de Laplace, son leídos por el programa principal desde la memoria RAM y almacenados en un archivo de texto usando formato decimal con 15 cifras significativas. La visualización de los resultados es realizada empleando la función *mesh* en GNU Octave. En la Figura 4-2 se muestran las

respuestas obtenidas para mallas de 256×256 y 10^1, 10^2, 10^3, 10^4, 10^5, y 10^6 iteraciones.

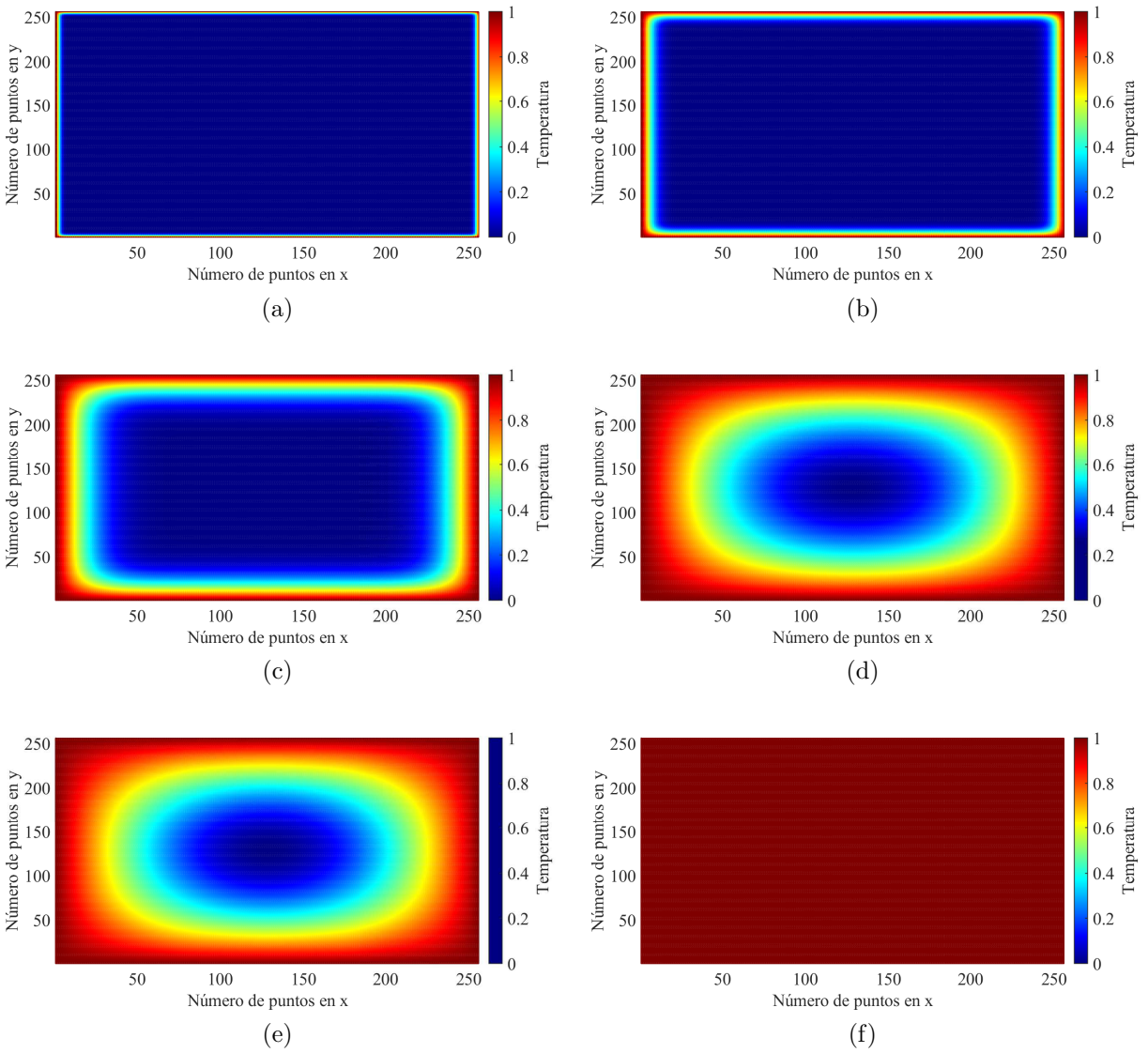


Figura 4-2.: Respuestas obtenidas con el sistema implementado en la ZedBoard para mallas de 256×256 y a) 10^1, b) 10^2, c) 10^3, d) 10^4, e) 10^5, y f) 10^6 iteraciones.

En la Figura 4-3 se muestran los errores absolutos obtenidos para mallas de 256×256 y 10^1, 10^2, 10^3, 10^4, 10^5, y 10^6 iteraciones.

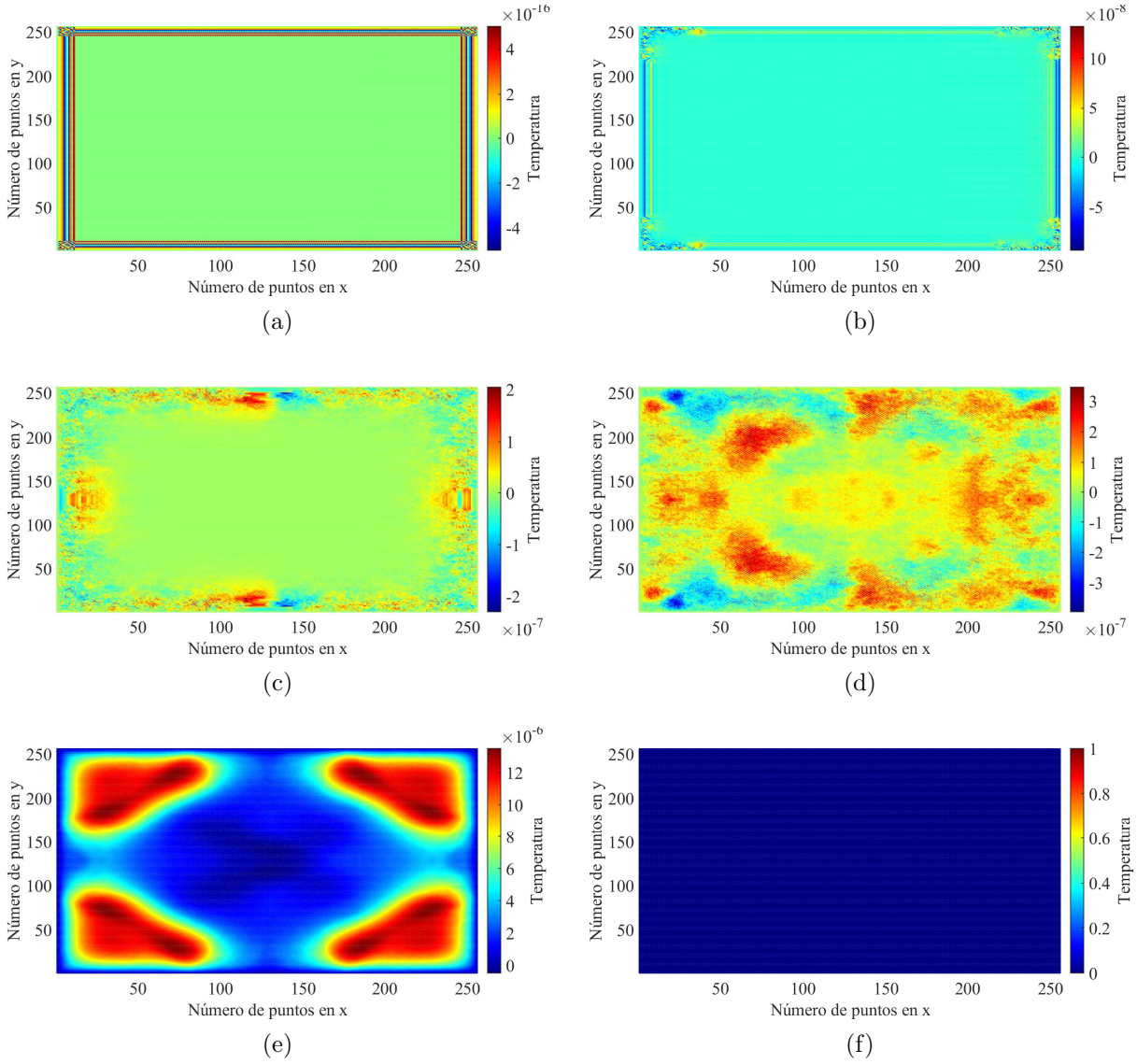


Figura 4-3.: Error absoluto para los resultados obtenidos con el sistema implementado en la ZedBoard con relación a la CPU, para mallas de 256×256 y a) 10^1, b) 10^2, c) 10^3, d) 10^4, e) 10^5, y f) 10^6 iteraciones.

4.1.2. Evaluación en términos del tiempo de ejecución

El desempeño evaluado en términos del tiempo de ejecución se presenta para la arquitectura base (A_1), la optimización utilizando directivas *pipeline* (A_2), y la arquitectura generalizada con combinación de partición de memoria y directivas pipeline (A_3). Dado que la arquitectura generalizada tiene variaciones, se realiza la exploración del espacio de diseño con base en la cantidad de subdivisiones de procesamiento de B2 y el factor utilizado para la partición de memoria. Para esto, se obtienen las latencias en términos de ciclos de reloj para diferentes combinaciones de ambos parámetros y cantidad de iteraciones. Las mediciones de latencia son realizadas para 4, 8, 16, 32, y 64 subdivisiones de B2, asignando al factor de partición de memoria los valores de 2, 4, 8, 16, 32, y 64. Para cada combinacón de estos parámetros se realiza la simulación para 10^1, 10^2, 10^3, 10^4, 10^5, y 10^6 iteraciones. A partir de las latencias obtenidas se calcula el tiempo de ejecución para una frecuencia de reloj de 100MHz. En la Table 4-1 se muestra el tiempo de ejecución en microsegundos para la arquitectura generalizada con 4 bloques de procesamiento en función de la cantidad de iteraciones y el factor de partición de memoria.

Tabla 4-1.: Tiempo de ejecución para la arquitectura generalizada con 4 bloques de procesamiento en función del número de iteraciones y el factor de partición de memoria

		Factor					
		2	**4**	**8**	**16**	**32**	**64**
N	10^1	7962,13	4914,03	3288,33	2475,93	2069,73	1866,03
	10^2	67916,08	37435,08	21178,08	13054,08	8992,08	6955,08
	10^3	667455,58	362645,58	200075,58	118835,58	78215,58	57845,58
	10^4	6662850,58	3614750,58	1989050,58	1176650,58	770450,58	566750,58
	10^5	66616800,58	36135800,58	19878800,58	11754800,58	7692800,58	5655800,58
	10^6	666156300,58	361346300,58	198776300,58	117536300,58	76916300,58	56546300,58

En la Figura 4-4 se muestra la latencia para 4 bloques de procesamiento en función de la cantidad de iteraciones y el factor de partición de memoria. Se observa que el rendimiento mejora con el incremento de este último parámetro.

La aceleración obtenida con la arquitectura generalizada con 4 bloques de procesamiento con relación a la arquitectura base en función de la cantidad de iteraciones y el factor de partición de memoria se muestra en la Tabla 4-2.

La aceleración obtenida con la arquitectura generalizada con 4 bloques de procesamiento con relación a la implementación secuencial en CPU en función de la cantidad de iteraciones y el factor de partición de memoria se muestra en la Tabla 4-3.

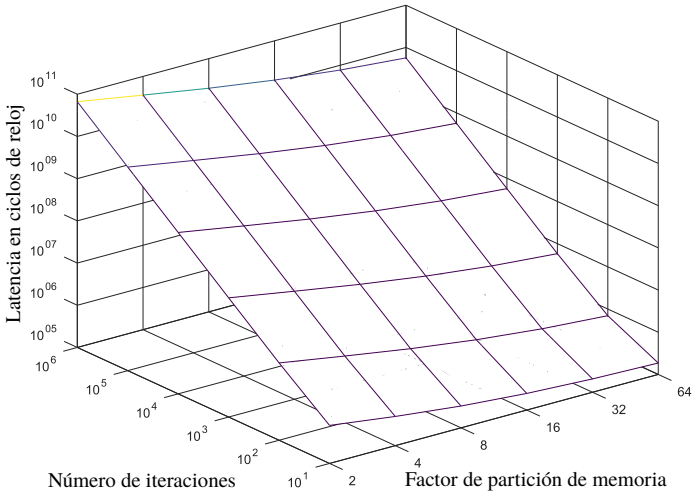


Figura 4-4.: Latencia para 4 bloques de procesamiento de la división intermedia en función de la cantidad de iteraciones y el factor de partición de memoria on-chip.

Tabla 4-2.: Aceleración con relación a la arquitectura base en función de la cantidad de iteraciones y el factor de partición de memoria.

		Factor					
		2	**4**	**8**	**16**	**32**	**64**
N	10^1	20,44	33,11	49,48	65,72	78,61	87,19
	10^2	23,78	43,15	76,27	123,73	179,63	232,24
	10^3	24,18	44,51	80,67	135,82	206,36	279,03
	10^4	24,22	44,65	81,14	137,16	209,48	284,77
	10^5	24,23	44,66	81,19	137,30	209,80	285,36
	10^6	24,23	44,66	81,19	137,31	209,83	285,42

Tabla 4-3.: Aceleración con relación a la ejecución secuencial en CPU en función de la cantidad de iteraciones y el factor de partición de memoria on-chip.

		Factor					
		2	**4**	**8**	**16**	**32**	**64**
N	10^1	0,87	1,41	2,10	2,79	3,34	3,77
	10^2	0,84	1,53	2,70	4,37	6,35	8,21
	10^3	0,79	1,46	2,64	4,45	6,76	9,14
	10^4	0,79	1,45	2,63	4,45	6,79	9,24
	10^5	0,78	1,44	2,61	4,41	6,74	9,17
	10^6	0,78	1,44	2,62	4,42	6,76	9,19

En ambos casos se observa que hay una cantidad de iteraciones a partir de las cuales la aceleración tiende a un valor constante, tal como muestra en la Figura **4-5**.

Para determinar la configuración que ofrece mejor rendimiento se realiza el gráfico de la

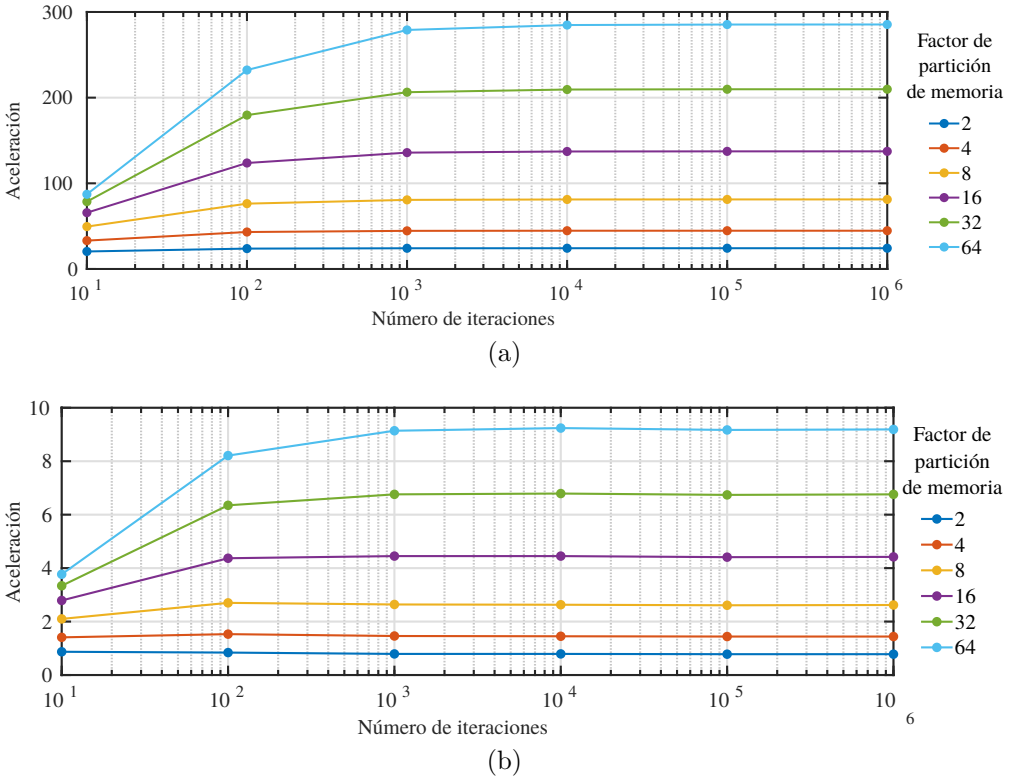


Figura 4-5.: Aceleración obtenida con la arquitectura generalizada con 4 bloques de procesamiento en función de la cantidad de iteraciones y el factor de partición de memoria: a) con relación a la arquitectura base, y b) con relación a la ejecución secuencial en CPU.

latencia en función de la cantidad de bloques de procesamiento de la división intermedia y el factor utilizado para la partición de memoria para 10^6 iteraciones, tal como se muestra en la Figura **4-6**. Se observa que la combinación de 4 bloques y factor 64 tiene la menor latencia. Esta configuración no se puede implementar en la ZedBoard por limitación de recursos físicos, por lo que la configuración implementada fue de 4 bloques y factor 32.

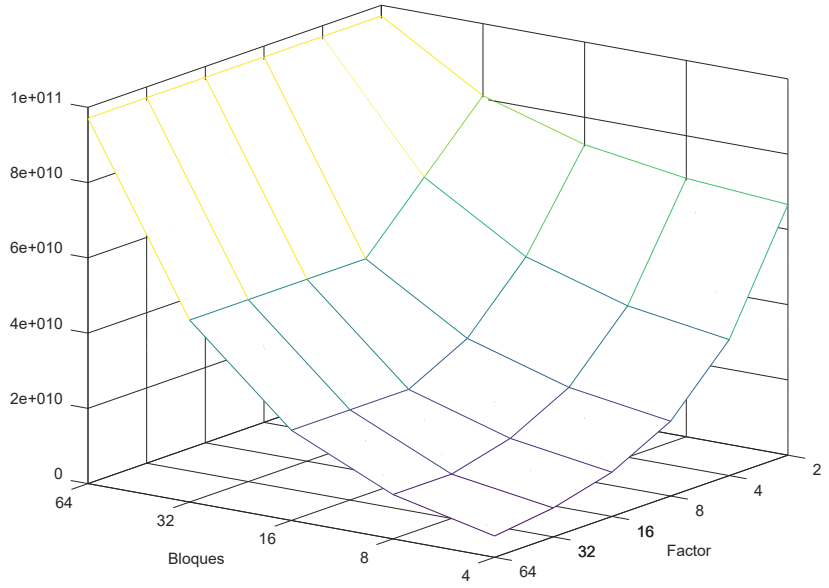


Figura 4-6.: Latencia para 10^6 de iteraciones en función de la cantidad de bloques de procesamiento y el factor de partición de memoria on-chip.

El tiempo de ejecución es medido para las arquitecturas implementadas en la ZedBoard. En la Tabla 4-4 se muestra el tiempo de ejecución según la cantidad de iteraciones para las arquitecturas A_1, A_2, la arquitectura generalizada con 16 bloques de procesamiento y factor 16 para partición de memoria (A_3), la arquitectura generalizada con 4 bloques de procesamiento y factor 32 para partición de memoria.

Tabla 4-4.: Tiempo de ejecución en microsegundos para diferentes valores de cantidad de iteraciones con las arquitecturas implementadas.

Iteraciones	A_1	A_2	$A_{3(16\times16)}$	$A_{3(4\times32)}$
10^1	158.212	16.199	3.710	2.086
10^2	1.552.677	150.163	25.259	9.008
10^3	15.497.303	1.489.794	240.753	78.231
10^4	154.943.576	14.886.115	2.395.667	770.466
10^5	1.549.406.305	148.849.317	23.944.818	7.692.819
10^6	15.494.033.580	1.488.481.343	239.436.321	76.916.319

La aceleración obtenida para las arquitecturas de procesamiento paralelo implementadas se calcula con relación a la arquitectura base, como se muestra en la Tabla **4-5**.

Tabla 4-5.: Aceleración obtenida con relación a la arquitectura base.

Iteraciones	t_{A_1}/t_{A_2}	$t_{A_1}/A_{3(16\times16)}$	$t_{A_1}/A_{3(4\times32)}$
10^1	9,77	42,65	75,85
10^2	10,34	61,47	172,37
10^3	10,40	64,37	198,10
10^4	10,41	64,68	201,10
10^5	10,41	64,71	201,41
10^6	10,41	64,71	201,44

Se observa que en cada caso hay una cantidad de iteraciones a partir de las cuales la aceleración tiende a un valor constante, tal como muestra en la Figura 4.1.2.

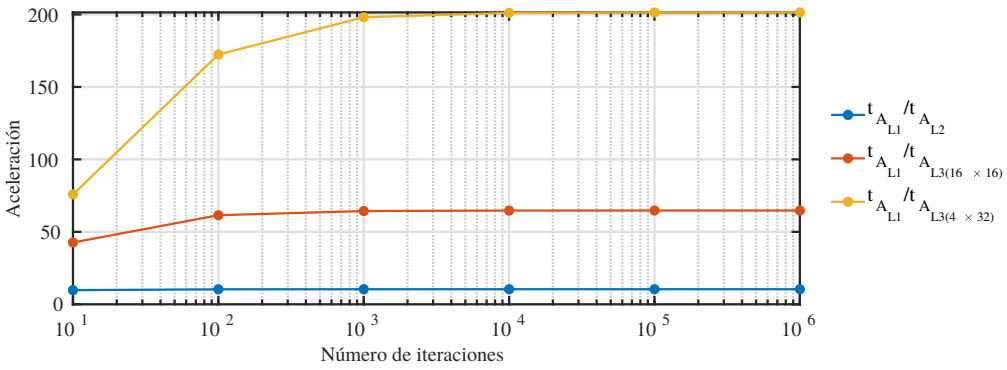


Figura 4-7.: Aceleración obtenida con relación a la arquitectura base.

La aceleración obtenida para las arquitecturas implementadas se calcula con relación a la implementación sobre CPU se muestra en la Tabla **4-6**.

Tabla 4-6.: Aceleración con relación a CPU para diferentes valores de cantidad de iteraciones con las arquitecturas implementadas.

Iteraciones	t_{CPU} $[\mu s]$	t_{CPU}/t_{A_1}	t_{CPU}/t_{A_2}	$t_{CPU}/t_{A_3(16\times16)}$	$t_{CPU}/t_{A_3(4\times32)}$
10^1	6913,55	0,04	0,43	1,86	3,31
10^2	57105,64	0,04	0,38	2,26	6,34
10^3	528741,06	0,03	0,35	2,20	6,76
10^4	5234150,50	0,03	0,35	2,18	6,79
10^5	51855436,00	0,03	0,35	2,17	6,74
10^6	519883296,00	0,03	0,35	2,17	6,76

Los datos de la Figura **4-8** muestran la variación de la aceleración con relación a la ejecución secuencial en CPU.

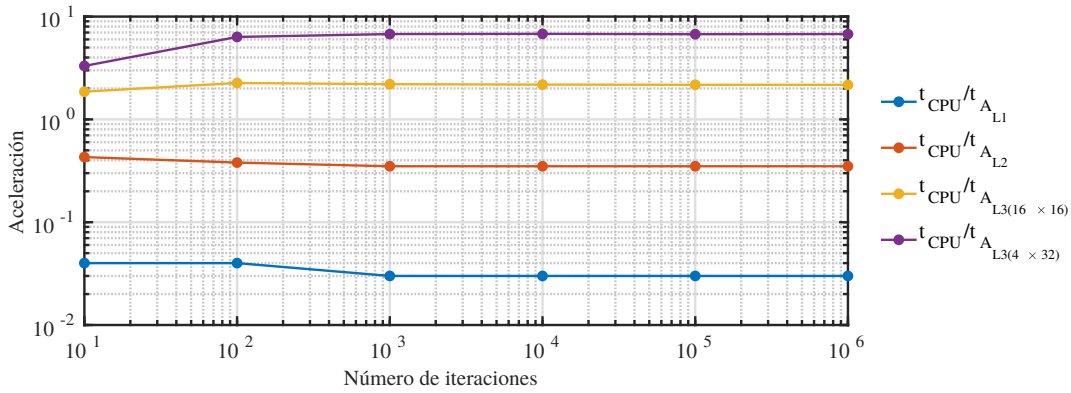


Figura 4-8.: Aceleración obtenida con relación a la ejecución secuencial en CPU.

La cantidad de recursos utilizados en cada arquitectura se muestra en la Tabla 4-7.

Tabla 4-7.: Recursos de hardware utilizados. Todo el sistema incluye módulos de procesamiento.

Resource	A_1	A_2	$A_{3(16\times16)}$	$A_{3(4\times32)}$
LUT	5.644	25.574	16.776	34.277
Flip Flop	6.599	22.836	17.689	39.204
Slices	2.267	9.100	6407	12.415
DSP48	5	17	36	97

El consumo de potencia para las arquitecturas implementadas en la Zedboard se muestra en la Tabla 4-8.

Tabla 4-8.: Consumo de potencia para las arquitecturas implementadas. Todo el sistema incluye módulos de procesamiento.

	A_1	A_2	$A_{3(16\times16)}$	$A_{3(4\times32)}$
Core Power (W)	0,297	1,005	1,037	1,535
TotalPower (W)	1,87	2,592	2,617	3,599

5. Conclusiones

En esta tesis se presenta el diseño, implementación y evaluación de desempeño de diferentes arquitecturas para la ejecución en FPGA de algoritmos basados en esténcil empleando el esquema de diferencias finitas. Se presentan como casos de estudio la aproximación a la solución numérica de la ecuación de calor en una dimensión y de la ecuación de Laplace en dos dimensiones.

Para la implementación de las diferentes arquitecturas se emplea un *ZedBoard Zynq Evaluation and Development Kit* bajo el entorno *Vivado Design Suite*. El esquema general de las arquitecturas propuestas involucra el uso de un microprocesador ARM Cortex-A9 que actúa como maestro, sobre el cual se ejecuta la aplicación principal. El código fuente de la aplicación principal es realizado en C y ejecutado bajo un sistema operativo *stand-alone*. El microprocesador interactúa con un IP personalizado descrito en VHDL o en C, el cual realiza la ejecución del algoritmo basado en esténcil. La comunicación entre el microprocesador y el IP se realiza a través de una interfaz de comunicación AXI.

En el primer caso de estudio, a partir de la implementación de una arquitectura base (A_1) se proponen dos arquitecturas paralelas para la aceleración del tiempo de ejecución del algoritmo. La primera arquitectura paralela (A_2) utiliza un banco de registros en el camino de datos para aprovechar la localidad espacial y temporal de los datos. La segunda arquitectura paralela (A_3) incorpora además del banco de registros, varios bloques de memoria RAM para mejorar el tiempo de ejecución realizando procesamiento y almacenamiento concurrente. Para todas las arquitecturas desarrolladas se presenta el diseño a nivel de transferencia entre registros basado en el modelo de Glushkov.

La comparación de las arquitecturas implementadas permite observar cómo las variaciones propuestas en el camino de datos influyen en la reducción del tiempo de ejecución del algoritmo. Se muestra que la estructura de la memoria es uno de los aspectos del camino de datos que más influye en el rendimiento. Además, se muestra que el uso del arreglo de registros introducido en el camino de datos permite tomar ventaja de la localidad espacial y temporal de los datos, reduciendo la cantidad de recursos utilizados y de operaciones de transferencia de memoria.

La evaluación de rendimiento se realiza en términos de los resultados numéricos y el tiempo de ejecución para la arquitectura base y las dos arquitecturas paralelas. En los resultados numéricos se observa que el error es mayor en la medida que aumenta la cantidad de iteraciones. Esto se debe a la acumulación del error por representación numérica y el tipo de aproximación utilizado. En el caso de 256 iteraciones el máximo del error relativo no supera 8×10^{-6}.

La evaluación en términos de tiempo de ejecución se realiza con A_1, A_2 y A_3 para mallas de 8, 16 y 32 puntos en el eje x, variando la cantidad de iteraciones en potencias de 2 hasta 256. La aceleración obtenida con las arquitecturas paralelas es calculada con relación a un procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM, a un núcleo de ARM de la SoC FPGA y a la arquitectura A_1.

La comparación de la arquitectura A_1 con la ejecución secuencial sobre el procesador Intel muestra una aceleración inferior a 1 para la mayoría de los casos evaluados. La arquitectura A_2 no muestra una mejora significa del rendimiento, presentando valores de aceleración similares o cercanos a los de la arquitectura A_1. Con la arquitectura A_3 se logra superar el desempeño del procesador tomando en cuenta únicamente el tiempo de procesamiento del algoritmo, obteniendo una aceleración de $8.1\times$ para una malla de 32×256.

La comparación de la arquitectura A_1 con la ejecución secuencial sobre el procesador ARM muestra una aceleración cercana a 1 para todos los casos evaluados teniendo en cuenta el tiempo de ejecución total. Considerando que la ejecución del algoritmo con A_1 es secuencial se identifica que la mejora del rendimiento se logra a nivel del camino de datos debido al diseño de los elementos de proceso.

La arquitectura A_3 es la que presenta mejor desempeño, alcanzando una aceleración de $82.59\times$ con relación al microprocesador ARM, $64.03\times$ relación a A_1 y $8.04\times$ con relación a la CPU. El rendimiento para la arquitectura A_2 puede ser mejorado si en la memoria RAM se almacenan únicamente los resultados de la última iteración, alcanzando una aceleración de 1.48 con relación a A_3.

En el segundo caso de estudio, se presenta una estrategia para la implementación de algoritmos basados en esténcil en SoC-FPGA utilizando la herramienta de síntesis de alto nivel Vivado HLS, abordando el problema de optimización en términos de gestión de memoria y paralelización de ciclos. La estrategia se basa en la combinación de directivas de optimización con transformaciones en el código para lograr un balance entre tiempo de respuesta y utilización de recursos lógicos.

El esquema general de las arquitecturas implementadas involucra el uso de un microprocesa-

dor ARM Cortex-A9 que actúa como maestro, sobre el cual se ejecuta la aplicación principal. El procesador interactúa a través de una interfaz AXI con un IP creado en *Vivado HLS*, el cual realiza la ejecución del algoritmo basado en esténcil. Las arquitecturas son implementadas sobre un sistema de desarrollo *ZedBoard Zynq Evaluation and Development Kit* bajo el entorno *Vivado Design Suite*. El código fuente de la aplicación principal es realizado en C y ejecutado bajo PetaLinux sobre el PS usando una consola de terminal. La comunicación es realizada empleando una interfaz AXI y acceso directo a memoria (DMA).

Se presenta una evaluación de desempeño segÃºn la aplicación de las directivas de optimización y se propone una estrategia para incrementar el desempeño a partir de la división del dominio de la solución en el espacio. Esta división se implementa con la definición de un nuevo ciclo interno, estableciendo un parámetro asociado al límite superior de este ciclo y otro parámetro correspondiente al factor de partición de memoria on-chip.

La nueva definición de los ciclos se establece a partir de la división del dominio de la solución en tres partes en el eje y. La distribución se realiza para que la cantidad de filas del bloque B2 sea múltiplo de una potencia de 2, considerando que los bloques B1 y B3 tienen una fila menos por incluir condiciones de contorno. Adicionalmente, la partición de la memoria on-chip se realiza de tal forma que cada subdivisión pueda acceder a los datos correspondientes de forma concurrente.

Esta estrategia permite obtener una implementación con un desempeño comparable al alcanzado con la metodología de Xilinx, con una utilización más eficiente de los recursos de hardware y mejor rendimiento en el tiempo. Se propone una transformación en dominio del tiempo, la cual combinada con la división en el dominio espacial y permite superar en más de 12 veces el rendimiento obtenido con esta estrategia utilizada en el dominio del espacio, utilizando menos recursos.

El desempeño es evaluado para la arquitectura base (A_1), la paralelización utilizando directivas *pipeline* (A_2), y una arquitectura generalizada con combinación de partición de memoria y directivas pipeline. Se realiza una exploración del espacio de diseño para la arquitectura generalizada, con base en la cantidad de subdivisiones de procesamiento de B2 y el factor utilizado para la partición de memoria. Para esto, se obtienen las latencias en función de ciclos de reloj para diferentes combinaciones de ambos parámetros y cantidad de iteraciones. Se observa que el rendimiento mejora con el incremento del factor de partición de memoria.

Se encuentra que la combinación que proporciona mejor rendimiento y que puede ser implementada en la *ZedBoard* es con 4 divisiones de B2 y 32 particiones de memoria. Para esta configuración se obtiene una aceleración de aproximadamente 209.83× con relación a la

arquitectura base y de 6.76× con relación a la CPU utilizada como referencia. El consumo de potencia con esta arquitectura es de aproximadamente 3.6 vatios.

Como trabajo futuro se propone evaluar la metodología propuesta para otro tipo de algoritmos de computación científica y para sistemas de desarrollo basados en otros dispositivos. Adicionalmente, es necesario evaluar la arquitectura en FPGA de mayores prestaciones para analizar el escalamiento de las estrategias propuestas a problemas con mayor cantidad de puntos en el dominio de la solución. Por otra parte, se requiere evaluar la aceleración que es posible alcanzar adicionando estrategias de optimización a nivel de transferencia de datos y gestión de memoria. Finalmente, se propone el desarrollo de una herramienta automática para la evaluación del espacio de diseño y la determinación de una implementación óptima dependiendo de la plataforma de desarrollo.

Appendices

A. Evaluación de rendimiento del sistema implementado con DE4

Las arquitecturas implementadas en el sistema *Altera (Intel) DE4 Development and Education Board* son desarrolladas bajo el entorno de desarrollo *Quartus+II*. En la Figura **A-1** se muestra el sistema de desarrollo utilizado.

Figura A-1.: Sistema de desarrollo *Altera DE4 Development and Education Board*.

El esquema general del sistema involucra el uso de un microprocesador que actúa como maestro, sobre el cual se ejecuta la aplicación principal. Este procesador interactúa con un IP personalizado descrito en VHDL, a través de una interfaz de comunicación determinada según el tipo de sistema de desarrollo. El IP se encarga de la ejecución del algoritmo basado en esténcil. La comunicación del sistema DE4 con el procesador es realizada mediante PCIe. El código fuente de la aplicación principal es realizado en C y ejecutado bajo una distribución del sistema operativo Linux. El diagrama de bloques del sistema implementado en la *DE4* se muestra la Figura **A-2**.

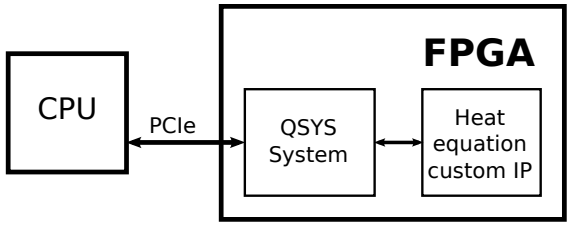


Figura A-2.: Diagrama de bloques del sistema implementado en una *Altera DE4 Development and Education Board* bajo el entorno *Quartus+II*.

La evaluación del rendimiento en términos de la aproximación a la solución numérica se realiza en comparación con los resultados obtenidos con CPU. Para este caso los valores iniciales y las condiciones de frontera están definidos como se muestra en (A-1).

$$\begin{cases} u(x,0) = & 2x, x & < 0.5 \\ u(x,0) = & 2*(1-x), 0.5 & < x < 1 \\ u(0,t) = & 0 & \\ u(L,t) = & 0 & \end{cases} \tag{A-1}$$

La vista superior de las respuestas obtenidas con la arquitectura base se muestra de la Figura **A-3**a a la Figura **A-3**f para mallas de 256 puntos en el eje x y 8, 16, 32, 64, 128, y 256 iteraciones respectivamente.

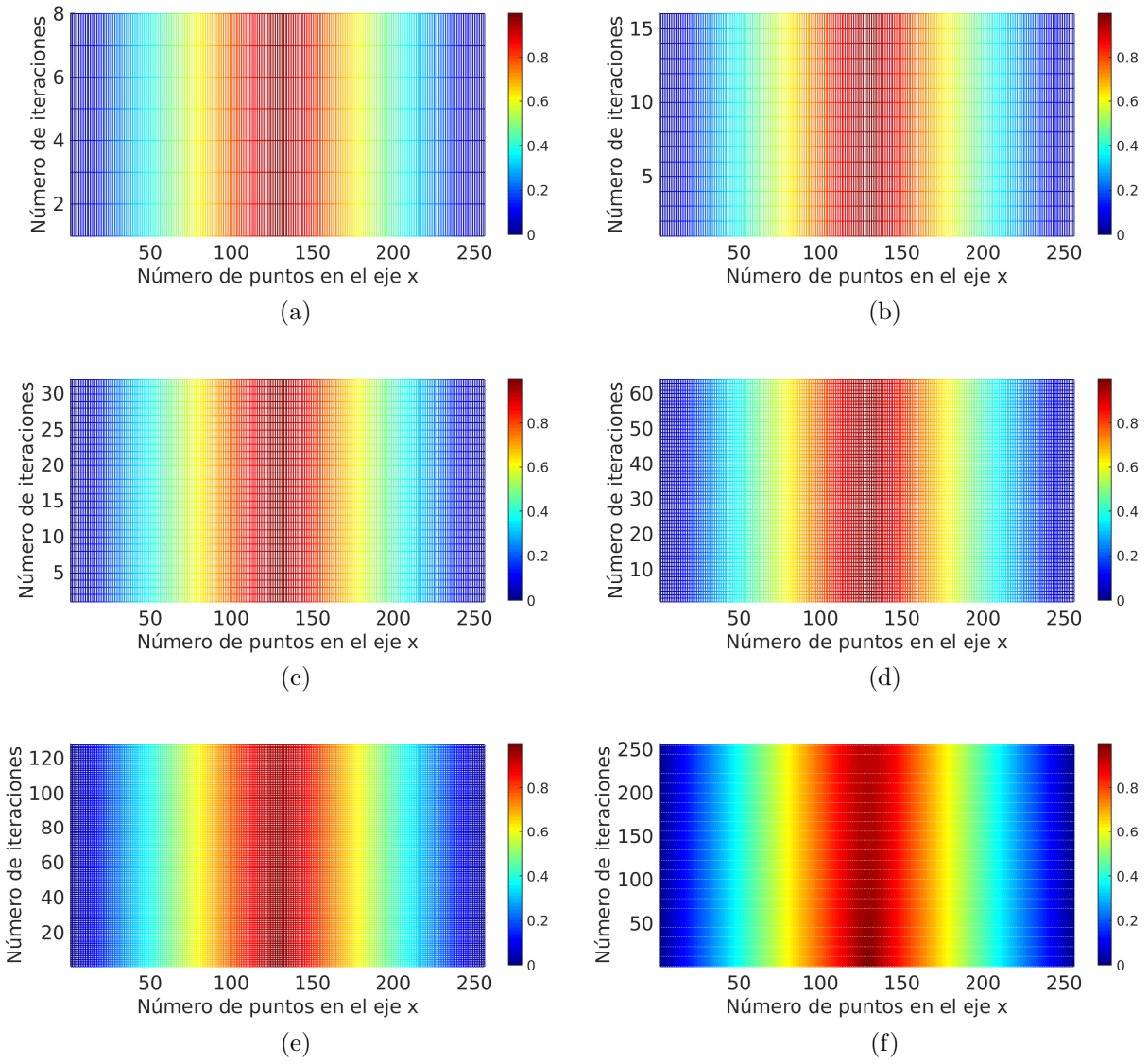


Figura A-3.: Vista superior de las respuestas obtenidas con el sistema implementado en la tarjeta DE4 para mallas de 256 puntos en el eje x y (a) 8, (b) 16, (c) 32, (d) 64, (e) 128, y (f) 256 iteraciones.

El error absoluto calculado con relación a los resultados numéricos obtenidos en CPU se muestra en la Figura **A-4**. Se observa que para una malla de 256×256 el error no supera el valor de 6×10^{-6}.

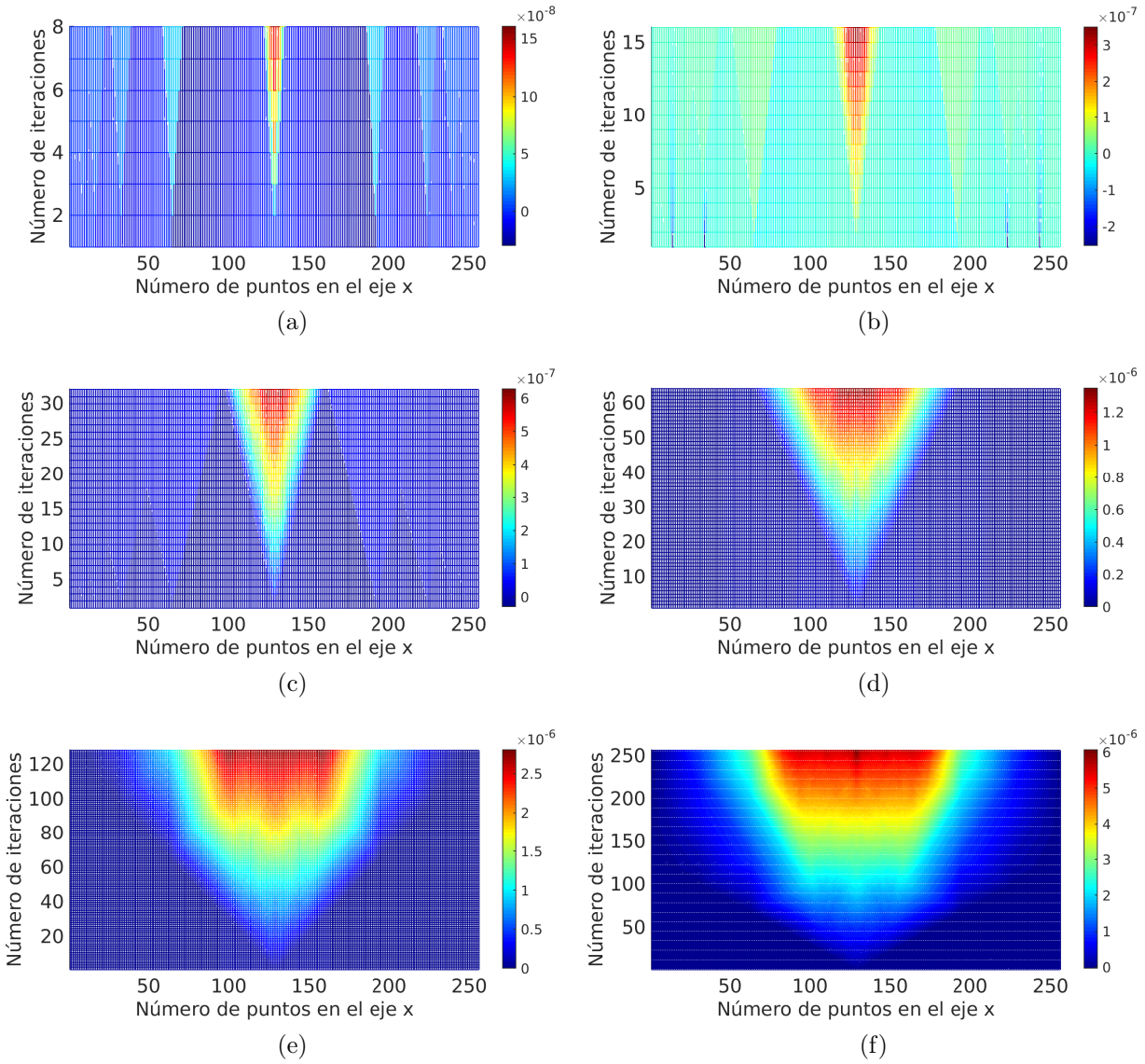


Figura A-4.: Vista superior del error absoluto obtenido con el sistema implementado en la tarjeta DE4 para mallas de 256 puntos en el eje x y (a) 8, (b) 16, (c) 32, (d) 64, (e) 128, y (f) 256 iteraciones.

El error relativo calculado con relación a los resultados numéricos obtenidos en CPU se muestra en la Figura **A-5**. Se observa que para una malla de 256×256 el error no supera el valor de 9×10^{-6}.

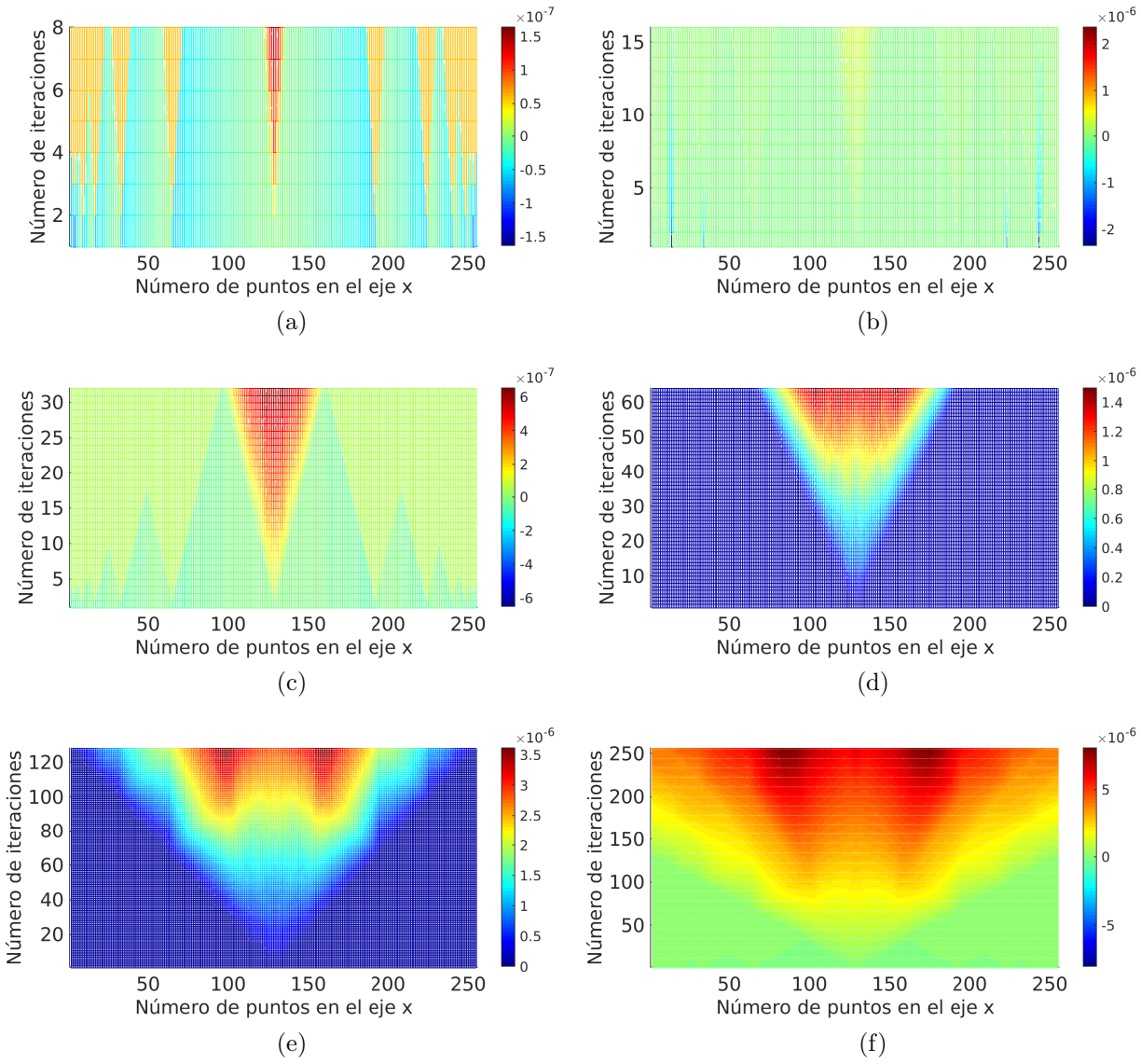


Figura A-5.: Vista superior del error relativo obtenido con el sistema implementado en la tarjeta DE4 para mallas de 256 puntos en el eje x y (a) 8, (b) 16, (c) 32, (d) 64, (e) 128, y (f) 256 iteraciones.

La evaluación del rendimiento en términos del tiempo de ejecución se realiza para diferentes tamaños de malla. Debido a que las medidas varían por factores relacionados con el sistema operativo, se toma el promedio de los valores de tiempo medidos en 1000 repeticiones para cada tamaño de malla evaluado. Las medidas son tomadas considerando la ejecución del código para la generación de los valores iniciales, la inicialización de la RAM, la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados en archivo de texto. En la Figura **A-6** se muestran las resultados correspondientes a los tiempos de ejecución sobre la arquitectura A_1 para mallas de 8 puntos en el eje x variando la cantidad de iteraciones.

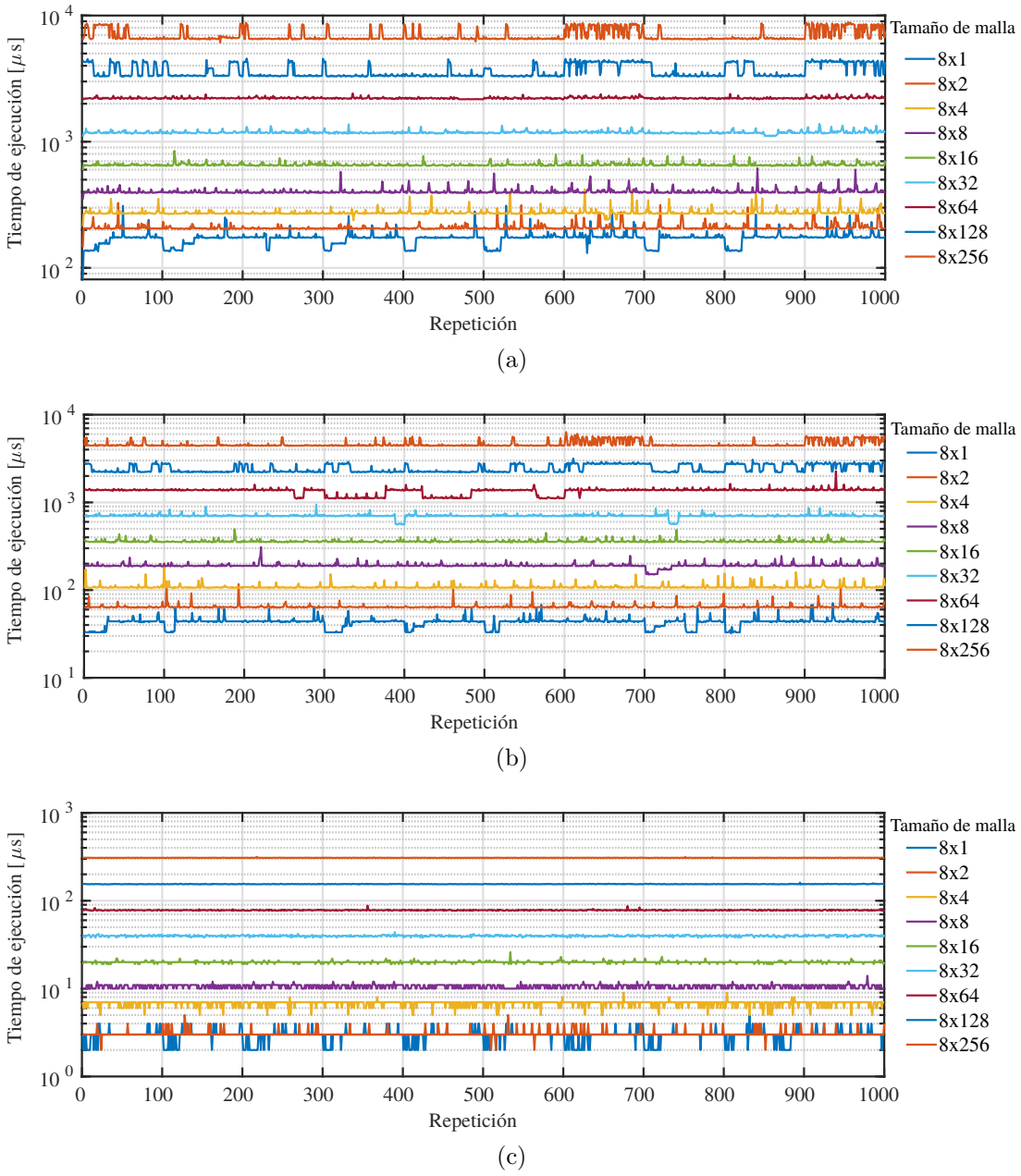


Figura A-6.: Tiempo transcurrido para mallas con 8 puntos en el eje x y diferentes valores del número de iteraciones correspondiente a (a) la ejecución de la tarea completa, (b) ejecución del algoritmo basado en esténcil con almacenamiento de los resultados en archivo de texto, y (c) ejecución del núcleo del algoritmo basado en esténcil.

En la Tabla **A-1** se muestra el tiempo total promedio empleando la arquitectura A_1 con una memoria de 4096 posiciones de 32 bits.

Tabla A-1.: Tiempo total promedio en microsegundos para diferentes combinaciones de $J \times N$ empleando la arquitectura A_1.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	401,85	660,40	1187,38	2210,81	3519,73	6619,24
	16	653,99	1164,42	2090,29	3576,69	6624,09	15129,79
	32	1106,12	2079,06	3513,70	6566,05	14634,49	
	64	2149,77	3488,39	6702,75	14749,89		
	128	3582,61	6561,87	14606,17			
	256	6660,77	14647,33				

En la Tabla **A-2** se muestra el tiempo promedio medido desde que se envía la señal de inicio de procesamiento hasta que se realiza el almacenamiento de los resultados en archivo.

Tabla A-2.: Tiempo promedio en microsegundos medido desde la señal de inicio de procesamiento hasta el almacenamiento de los resultados en archivo.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	190,02	355,82	704,46	1335,14	2348,57	4500,44
	16	359,06	703,28	1360,67	2367,50	4549,85	9213,12
	32	695,43	1344,12	2370,53	4592,15	9192,51	
	64	1383,31	2397,90	4600,93	9121,03		
	128	2372,78	4568,18	9131,11			
	256	4546,56	9091,77				

En la Tabla **A-3** se muestra el tiempo promedio medido desde que se envía la instrucción de inicio de procesamiento hasta que se recibe la indicación de finalización.

Tabla A-3.: Tiempo promedio en microsegundos medido para la ejecución del algoritmo basado en esténcil para diferentes combinaciones de $J \times N$ empleando A_1.

		N					
		8	**16**	**32**	**64**	**128**	**256**
J	**8**	10,80	20,76	40,48	78,61	154,79	308,16
	16	22,07	42,87	84,75	168,81	337,58	674,43
	32	41,40	85,85	173,63	349,76	703,36	
	64	81,83	171,69	351,55	712,93		
	128	162,16	344,37	708,02			
	256	323,07	689,35				

Con las medidas realizadas por cada tamaño de malla se obtiene el tiempo de ejecución promedio de los diferentes segmentos de código. Los datos de la Figura **A-7** muestran esta variación para cada uno de los segmentos de código definidos.

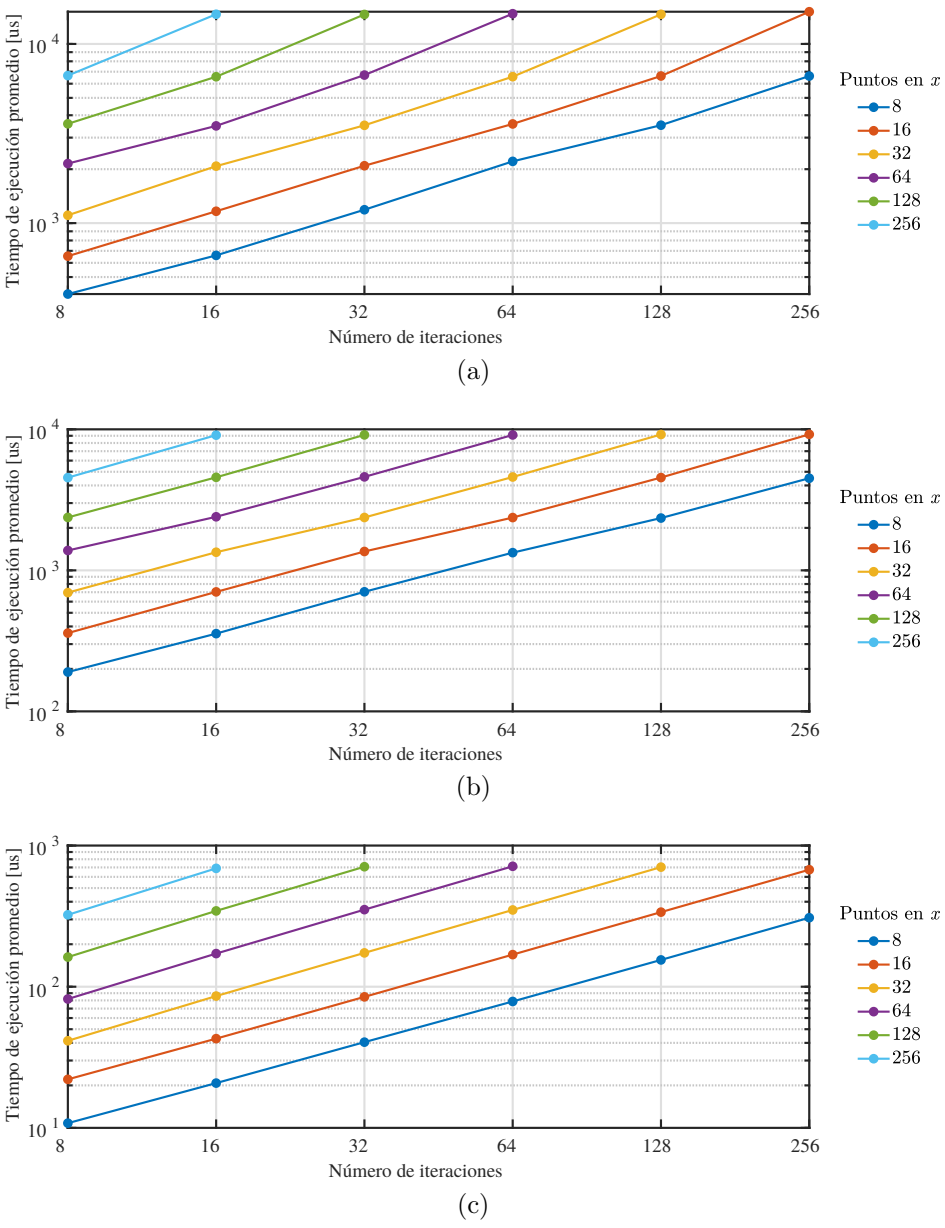


Figura A-7.: Variación del tiempo en función de la cantidad de puntos en x según el número de iteraciones para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

Para tener una referencia de rendimiento, el algoritmo es implementado en C para su ejecución sobre un procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM. La comparación de los tiempos de ejecución sobre las arquitecturas A_1, A_2, A_3, y el procesador se muestra en la Figura **A-8** para una malla de 8 puntos en x en función del número de iteraciones.

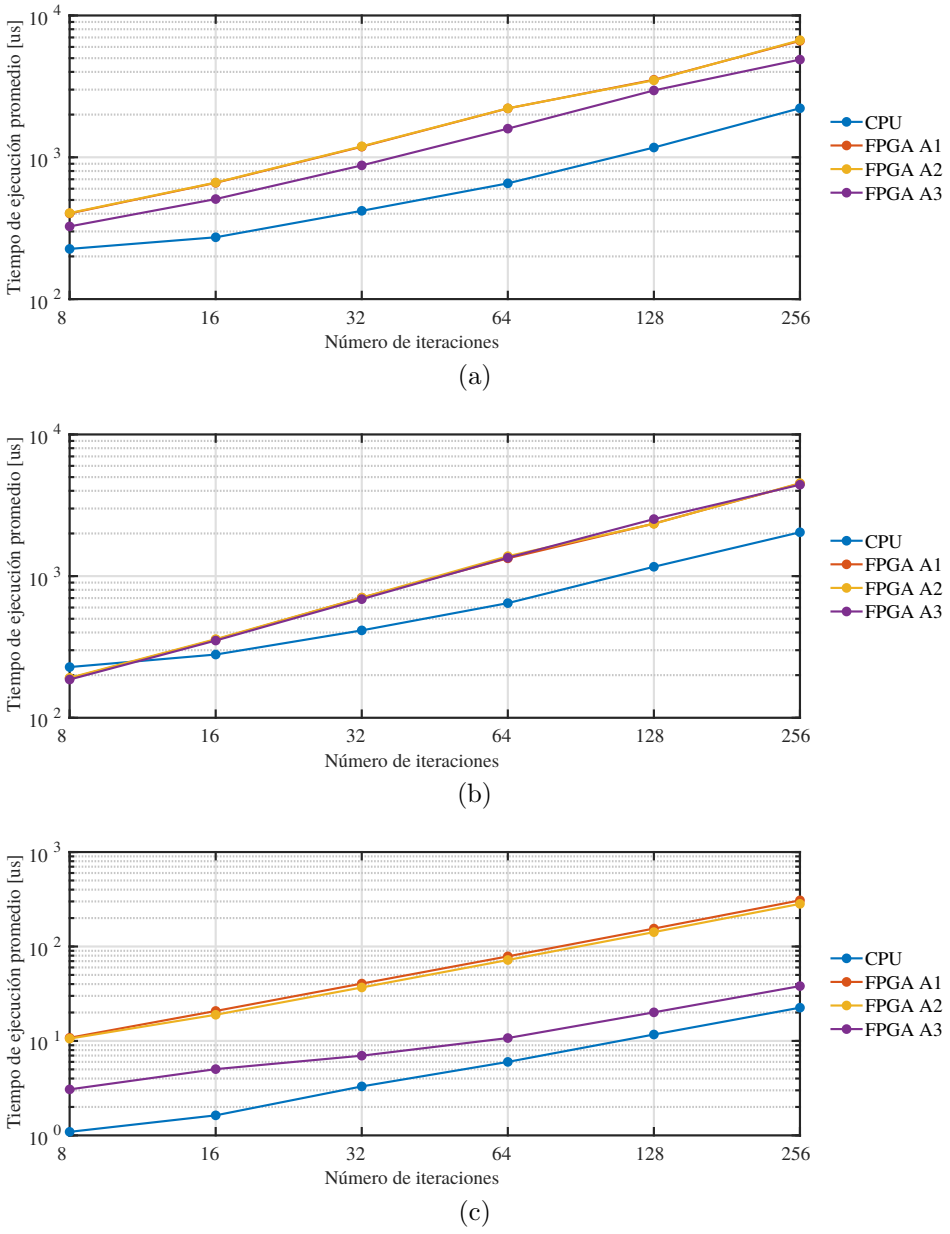


Figura A-8.: Comparación del tiempo de ejecución para una malla de 8 puntos en x según el número de iteraciones empleando las arquitecturas A_1, A_2, A_3, y un procesador Intel Xeon E5-2667 a 2.90GHz con 32 GB de RAM. El tiempo es medido para para los segmentos de código de (a) configuración, procesamiento y almacenamiento (b) procesamiento y almacenamiento, y (c) procesamiento.

La aceleración obtenida empleando las arquitecturas A_1, A_2, y A_3 para mallas de 8 puntos en el eje x, se calcula con relación a la ejecución secuencial sobre el microprocesador. En la Tabla **A-4** se muestra la aceleración para la ejecución de la tarea completa. Se observa que no se mejora el rendimiento con las arquitecturas propuestas.

Tabla A-4.: Aceleración obtenida con las arquitecturas A_1, A_2, y A_3 para la ejecución de la tarea completa con relación a la ejecución secuencial sobre CPU.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	0,56	0,41	0,35	0,30	0,33	0,33
$S_2 = t_{CPU}/t_{A_2}$	0,56	0,41	0,35	0,30	0,34	0,33
$S_3 = t_{CPU}/t_{A_3}$	0,69	0,54	0,48	0,41	0,40	0,45

En la Tabla **A-5** se muestra el cálculo de la aceleración obtenida con las arquitecturas A_1, A_2, y A_3 considerando la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados, con relación a la ejecución secuencial sobre la CPU. Los valores obtenidos son similares para las tres arquitecturas debido a que los datos son leídos de la memoria de la misma forma, independientemente de la arquitectura.

Tabla A-5.: Aceleración obtenida con las arquitecturas A_1, A_2, y A_3 considerando procesamiento y almacenamiento con relación a la ejecución secuencial sobre la CPU.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	1,20	0,79	0,59	0,48	0,50	0,45
$S_2 = t_{CPU}/t_{A_2}$	1,19	0,78	0,59	0,47	0,50	0,45
$S_3 = t_{CPU}/t_{A_3}$	1,22	0,80	0,60	0,48	0,46	0,46

En la Tabla **A-6** se muestra la aceleración obtenida con las arquitecturas A_1, A_2, y A_3 considerando la ejecución del algoritmo basado en esténcil con relación a la ejecución secuencial sobre la CPU. Se observa un mejor rendimiento de la arquitectura A_3 con relación a A_1 y A_2. Además, la aceleración se incrementa con el aumento de la cantidad de iteraciones.

Tabla A-6.: Aceleración obtenida con A_1, A_2, y A_3 para la ejecución del algoritmo basado en esténcil con relación a la ejecución secuencial sobre la CPU.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	0,10	0,08	0,08	0,08	0,08	0,07
$S_2 = t_{CPU}/t_{A_2}$	0,10	0,09	0,09	0,08	0,08	0,08
$S_3 = t_{CPU}/t_{A_3}$	0,36	0,32	0,47	0,56	0,58	0,59

Por otra parte, se calcula la aceleración obtenida con las arquitecturas A_2 y A_3 con relación a la arquitectura A_1. En la Tabla **A-7** se muestra la aceleración para la ejecución de la tarea completa. Por otra parte, se observa que no se logra mejor rendimiento con relación a la ejecución secuencial en CPU.

Tabla A-7.: Aceleración obtenida con A_2 y A_3 para la ejecución de la tarea completa con relación a A_1.

	N					
	8	16	32	64	128	256
t_{A_1}/t_{A_2}	0.99	1.00	0.99	1.00	1.01	0.99
t_{A_1}/t_{A_3}	1.23	1.30	1.35	1.39	1.19	1.35

En la Tabla **A-8** se muestra la aceleración para la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados. Se observa que con la arquitectura A_2 no se logra un incremento significativo en el rendimiento a pesar de contar con un mayor número de elementos de proceso, debido pricipalmente a la estructura de la memoria.

Tabla A-8.: Aceleración obtenida con A_2 y A_3 para la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados con relación a A_1.

	N					
	8	16	32	64	128	256
t_{A_1}/t_{A_2}	0.99	0.99	1.00	0.97	1.00	0.99
t_{A_1}/t_{A_3}	1.02	1.01	1.02	0.99	0.93	1.02

En la Tabla **A-9** se muestra la aceleración para la ejecución del algoritmo basado en esténcil con relación a A_1. Se observa que con la arquitectura A_3 se logra mejorar el rendimiento, llegando a superar los 8× para 256 y 512 iteraciones. Esto se debe principalmente a la reducción de la cantidad de instrucciones necesarias para el acceso a los datos, por la forma en que está organizada la memoria y el banco de registros.

Tabla A-9.: Aceleración obtenida con las arquitecturas A_2 y A_3 para la ejecución del algoritmo basado en esténcil, con relación a la arquitectura A_1

	N					
	8	16	32	64	128	256
t_{A_1}/t_{A_2}	1.02	1.10	1.10	1.09	1.09	1.09
t_{A_1}/t_{A_3}	3.52	4.13	5.80	7.33	7.71	8.09

Aunque se logra una aceleración superior a 8× en la ejecución del algoritmo basado en esténcil, el incremento del rendimiento para la ejecución de la tarea completa es bajo. Se observa que el almacenamiento de los resultados limita la mejora del rendimiento, ya que en la realización de esta tarea se emplea una mayor proporción del tiempo de ejecución que el algoritmo basado en esténcil.

Adicionalmente, se presenta la evaluación del rendimiento al aumentar la cantidad de EP a 14 y RAM a 16. La comparación del tiempo de ejecución según el número de iteraciones con las arquitecturas A_1, A_3 y el microprocesador, se muestra en la Figura **A-9** para una malla de 16 puntos en x.

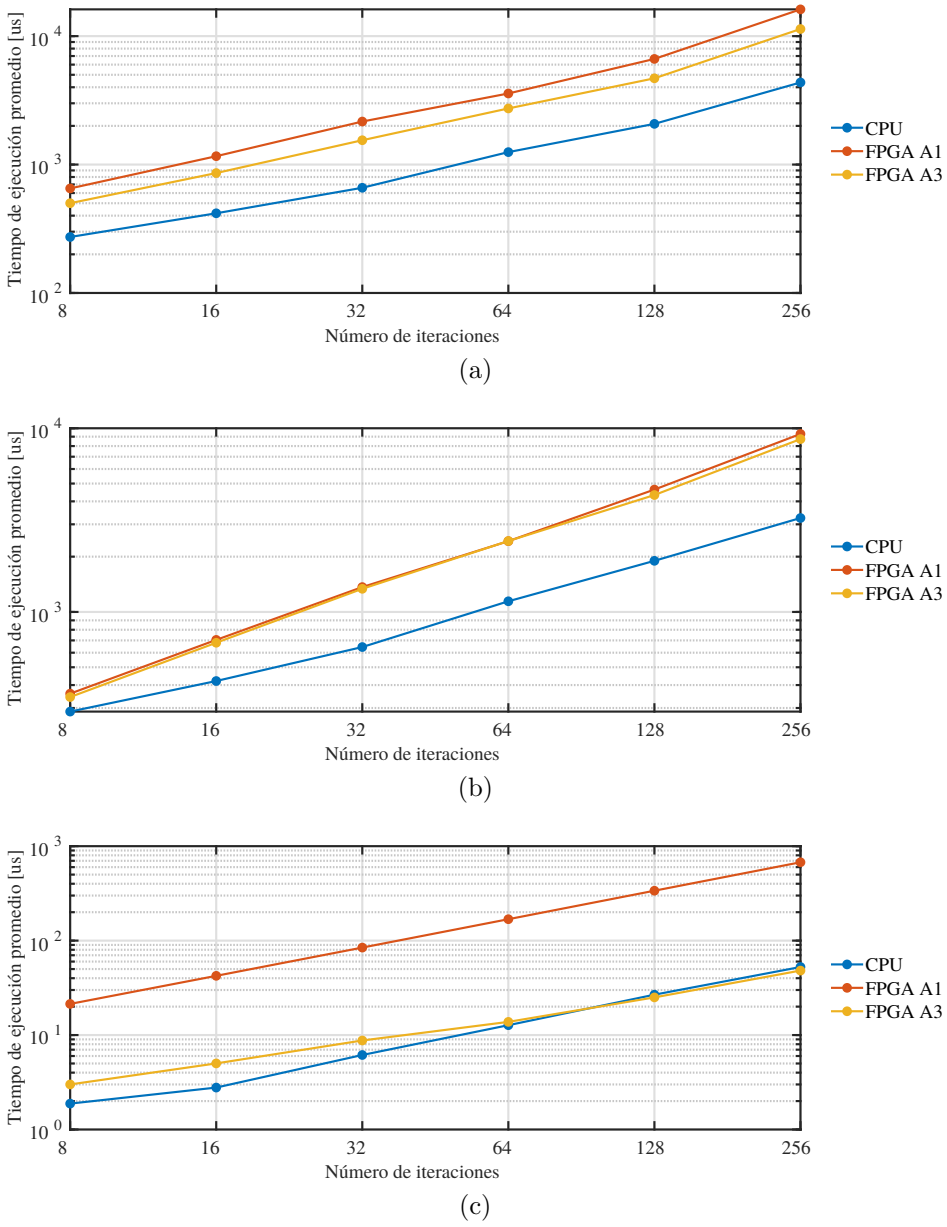


Figura A-9.: Comparación del tiempo de ejecución para una malla de 16 puntos en x según el número de iteraciones con A_1, A_3, y el procesador. El tiempo es medido para los segmentos de código de a) configuración, procesamiento y almacenamiento b) procesamiento y almacenamiento, y c) procesamiento.

En la Tabla **A-10** se muestra la aceleración para la ejecución de la tarea completa con mallas de 16 puntos en el eje x. Se presenta la aceleración obtenida con A_1 y A_3 con relación a la ejecución secuencial en microprocesador y la obtenida con A_3 con relación a A_1.

Tabla A-10.: Aceleración obtenida para la ejecución de la tarea completa con A_1 y A_3.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	0.42	0.36	0.30	0.35	0.31	0.27
$S_2 = t_{CPU}/t_{A_3}$	0.55	0.49	0.43	0.46	0.44	0.38
$S_3 = t_{A_1}/t_{A_3}$	1.30	1.35	1.40	1.31	1.42	1.42

En la Tabla **A-11** se muestra el cálculo de la aceleración obtenida con A_1 y A_3 para mallas de 16 puntos en el eje x considerando la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados. Los valores obtenidos utilizando A_3 con relación a A_1, son similares a los obtenidos con 8 elementos de proceso debido a que el volumen de datos que son leídos de la memoria está definido por el tamaño de la malla.

Tabla A-11.: Aceleración obtenida con A_1 y A_3 considerando la ejecución del algoritmo basado en esténcil y el almacenamiento de los resultados.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	0.80	0.60	0.47	0.47	0.41	0.35
$S_2 = t_{CPU}/t_{A_3}$	0.83	0.62	0.48	0.47	0.44	0.37
$S_3 = t_{A_1}/t_{A_3}$	1.04	1.04	1.02	1.00	1.07	1.07

En la Tabla **A-12** se muestra la aceleración obtenida con A_1 y A_3 para mallas de 16 puntos en el eje x considerando solamente la ejecución del algoritmo basado en esténcil. Se observa que en este caso A_3 logra mejorar el rendimiento de la CPU a partir de 128 iteraciones. Por otro lado, se observa un mejor desempeño con relación a la arquitectura con 8 elementos de proceso.

Tabla A-12.: Aceleración obtenida con A_1, A_2, y A_3 considerando la ejecución del algoritmo basado en esténcil.

	N					
	8	16	32	64	128	256
$S_1 = t_{CPU}/t_{A_1}$	0.09	0.07	0.07	0.08	0.08	0.08
$S_2 = t_{CPU}/t_{A_3}$	0.63	0.55	0.70	0.92	1.07	1.09
$S_3 = t_{A_1}/t_{A_3}$	7.12	8.46	9.63	12.22	13.48	14.02

B. Publicaciones

Artículos en eventos

Montoya, C. A., & Sánchez, R. D., & Castaño, L. F. (2016, Agosto). Approach to the numerical solution of Lorenz system on SoC FPGA. En Signal Processing, Images and Artificial Vision (STSIVA), 2016 XXI Symposium on (pp. 1-4). IEEE.

Castano, L., & Osorio, G. (2017, Octubre). *Stencil computation for the approach to the numerical solution of heat transfer problems on SoC FPGA*. En Automatic Control (CLCA), 2016 XVII Latin American Conference of (pp. 350–355).

Artículos en revista

Castaño, L., & Osorio, G. (2017). An approach to the numerical solution of one-dimensional heat equation on SoC FPGA. Revista Científica de Ingeniería Electrónica, Automática y Comunicaciones, 38(2), 83-93.

Castano-Londono L., Alzate Anzola C., Marquez-Viloria D., Gallo G., Osorio G. (2019) Evaluation of Stencil Based Algorithm Parallelization over System-on-Chip FPGA Using a High Level Synthesis Tool. In: Figueroa-Garcá J., Duarte-González M., Jaramillo-Isaza S., Orjuela-Cañon A., Díaz-Gutierrez Y. (eds) Applied Computer Sciences in Engineering. WEA 2019. Communications in Computer and Information Science, vol 1052. Springer, Cham.

C. Código fuente

C.1. Arquitectura base ecuación de calor 1D

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity c_up_heat_streaming_65536 is
6  Port (  data_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
7                  clk              : IN STD_LOGIC;
8                  control : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
9                  K1, K2  : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
10                 N                : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
11                 L                : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
12                 RESULTADOS       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
13                 estado   : INOUT STD_LOGIC;
14                 n_estados        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
15                 C1               : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
16                 C2               : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
17 );
18 end c_up_heat_streaming_65536;
19
20 architecture estructural of c_up_heat_streaming_65536 is
21
22 component ep_heat is
23     Port ( X_0N  : in STD_LOGIC_VECTOR (31 downto 0);
24            X_1N  : in STD_LOGIC_VECTOR (31 downto 0);
25            X_2N  : in STD_LOGIC_VECTOR (31 downto 0);
26            K_1   : in STD_LOGIC_VECTOR (31 downto 0);
27            K_2   : in STD_LOGIC_VECTOR (31 downto 0);
28            X_1Np : out STD_LOGIC_VECTOR (31 downto 0));
29 end component;
30
31 component mux2a1_nb is
32 generic(N: integer range 1 to 64 := 32 );
33 port(   D1, D0 : in  std_logic_vector(N-1 downto 0);
34         s    : in  std_logic;
35         Y    : out std_logic_vector(N-1 downto 0));
36 end component;
37
38 component sp_bram_65536x32 is
39   port (        addra : in STD_LOGIC_VECTOR ( 15 downto 0 );
40                 clka : in STD_LOGIC;
41                 dina : in STD_LOGIC_VECTOR ( 31 downto 0 );
42                 douta : out STD_LOGIC_VECTOR ( 31 downto 0 );
43                 wea : in STD_LOGIC_VECTOR ( 0 to 0 ));
44 end component sp_bram_65536x32;
45
```

```
46 component registro is
47 generic(N: integer range 1 to 32 := 32 );
48 port(   D:   in  std_logic_vector(N-1 downto 0);
49         clk: in  std_logic;
50         en, clr: in  std_logic;
51         Q:   out std_logic_vector(N-1 downto 0));
52 end component;
53
54 component unidad_control_streaming_65536 is
55 port(   control:        in  std_logic_vector(31 downto 0);
56         N:              in  std_logic_vector(15 downto 0);
57         L:              in  std_logic_vector(15 downto 0);
58         clk:            in  std_logic;
59         data_src:       out std_logic;
60         r_wr_en:                out std_logic;
61         r_clr:          out std_logic;
62         ram_address:    out std_logic_vector(15 downto 0);
63         ram_wr_en:      out std_logic_vector(0 to 0);
64         clr:                    out std_logic;
65         n_estados:      out std_logic_vector(3 downto 0);
66         e_control:      out std_logic );
67 end component;
68
69 component performance_counter is
70 port(   EN1, EN2: in  std_logic;
71         clk, clr: in  std_logic;
72         C1      : out std_logic_vector(63 DOWNTO 0);
73         C2      : out std_logic_vector(31 downto 0));
74 end component;
75
76 signal ram_data_in:  std_logic_vector(31 downto 0);
77 signal ram_data_out: std_logic_vector(31 downto 0);
78 signal data_back:    std_logic_vector(31 downto 0);
79 signal X0, X1, X2:   std_logic_vector(31 downto 0);
80 signal data_src:     std_logic;
81 signal r_wr_en:      std_logic;
82 signal r_clr:        std_logic;
83 signal ram_address:  std_logic_vector(15 downto 0);
84 signal ram_wr_en:    std_logic_vector(0 to 0);
85 signal clk_n, clr:   std_logic;
86
87 begin
88
89 clk_n <= not(clk);
90
91 u_uc:  unidad_control_streaming_65536
92 port map (control, N, L, clk, data_src, r_wr_en, r_clr,
93 ram_address, ram_wr_en, clr, n_estados, estado);
94
95 u_mux: mux2a1_nb port map
96 (data_back, data_in, data_src, ram_data_in);
97
98 u_ram: sp_bram_65536x32
99 port map (ram_address, clk_n, ram_data_in,
100 ram_data_out, ram_wr_en);
101
102 u_r0: registro port map ( ram_data_out, clk, r_wr_en, clr, X0);
103
104 u_r1: registro port map ( X0, clk, r_wr_en, clr, X1);
105
106 u_r2: registro port map ( X1, clk, r_wr_en, clr, X2);
```

```
107
108 u_ep: ep_heat port map (X0, X1, X2, K1, K2, data_back);
109
110 u_k:  performance_counter port map ( control(29), estado, clk, control(30), C1, C2 );
111
112 RESULTADOS <= ram_data_out;
113
114 end estructural;
```

C.1.1. Sumador de punto flotante de 32 bits

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity sumador_pf32 is
6      Port ( X : in std_logic_vector (31 downto 0);
7             Y : in std_logic_vector (31 downto 0);
8             S : out std_logic_vector (31 downto 0));
9  end sumador_pf32;
10
11 architecture estructural of sumador_pf32 is
12
13 component comparador is
14 generic( N: integer range 1 to 64 := 32 );
15 port(   X, Y: in  unsigned( N-1 downto 0 );
16        c:    out std_logic );
17 end component;
18
19 component corrimiento is
20 generic( M, N: integer range 1 to 64 := 32 );
21 port(   X:     in  unsigned( N-1 downto 0 );
22        shamt: in  unsigned( M-1 downto 0 );
23        dir:   in  std_logic;
24        Y:     out unsigned( N-1 downto 0 ) );
25 end component;
26
27 component detector_zero is
28 generic( N: integer range 1 to 64 := 32 );
29 port(   X: in  unsigned( N-1 downto 0 );
30        z: out std_logic );
31 end component;
32
33 component mux2a1_nb is
34 generic(N: integer range 1 to 64 := 32 );
35 port(   D1, D0: in  std_logic_vector(N-1 downto 0);
36        s:      in  std_logic;
37        Y:      out std_logic_vector(N-1 downto 0));
38 end component;
39
40 component normalizar is
41 port(   ExpX: in  unsigned( 7 downto 0 );
42        X: in  unsigned( 39 downto 0 );
43        Y: out unsigned( 30 downto 0 ) );
44 end component;
45
46 component redimensionar is
47 port(   X: in  unsigned( 23 downto 0 );
```

```
48          Y: out unsigned( 39 downto 0 ) );
49  end component;
50
51  component restador is
52  generic( N: integer range 1 to 64 := 32 );
53  port(    X, Y: in  unsigned( N-1 downto 0 );
54          R:    out unsigned( N-1 downto 0 ) );
55  end component;
56
57  component sumador_restador is
58  generic( N: integer range 1 to 64 := 32 );
59  port(    X, Y: in  unsigned( N-1 downto 0 );
60          s: in std_logic;
61          F:    out unsigned( N-1 downto 0 ) );
62  end component;
63
64  signal signo_X, signo_Y: std_logic;
65  signal AS: std_logic;
66  signal exp_X, exp_Y: unsigned(7 downto 0);
67  signal exp_A, exp_B: std_logic_vector(7 downto 0);
68  signal exp_comp: std_logic;
69  signal mant_comp, mant_comp_n: std_logic;
70  signal shamt: unsigned(7 downto 0);
71  signal mant_X, mant_Y: unsigned(23 downto 0);
72  signal mant_A, mant_B: std_logic_vector(23 downto 0);
73  signal mant_Ar, mant_Br: unsigned(39 downto 0);
74  signal mant_Ad, mant_Bd: std_logic_vector(39 downto 0);
75  signal mant_Brc, mant_S: unsigned(39 downto 0);
76  signal mant_sel, mant_sel_n: std_logic;
77  signal zX, zY: std_logic;
78  signal St: unsigned(30 downto 0);
79
80  begin
81
82  signo_X <= X(31);
83  signo_Y <= Y(31);
84
85  exp_X <= unsigned(X(30 downto 23));
86  exp_Y <= unsigned(Y(30 downto 23));
87
88  mant_sel <= signo_X xor signo_Y;
89  mant_sel_n <= not(mant_sel);
90
91  mant_X <= unsigned(not(zX) & X(22 downto 0));
92  mant_Y <= unsigned(not(zY) & Y(22 downto 0));
93
94  exp_comp_n <= not(exp_comp);
95  mant_comp_n <= not(mant_comp);
96
97  u_comparador_exp: comparador generic map (8)
98                              port map (exp_X, exp_Y, exp_comp);
99
100
101 u_mux_exp_A: mux2a1_nb  generic map (8)
102                         port map (  std_logic_vector(exp_X),
103                                     std_logic_vector(exp_Y),
104                                     exp_comp, exp_A);
105
106 u_mux_exp_B: mux2a1_nb  generic map (8)
107                         port map (  std_logic_vector(exp_X),
108                                     std_logic_vector(exp_Y),
```

```
109                                     exp_comp_n, exp_B);
110
111 u_restador: restador generic map (8)
112                       port map ( unsigned(exp_A),
113                                  unsigned(exp_B),
114                                  shamt);
115
116 u_dz1: detector_zero port map (unsigned(X),zX);
117
118 u_dz2: detector_zero port map (unsigned(Y),zY);
119
120 u_mux_mant_A: mux2a1_nb generic map (24)
121                         port map (  std_logic_vector(mant_Y),
122                                     std_logic_vector(mant_X),
123                                     exp_comp_n, mant_A);
124
125 u_mux_mant_B: mux2a1_nb generic map (24)
126                         port map (  std_logic_vector(mant_Y),
127                                     std_logic_vector(mant_X),
128                                     exp_comp, mant_B);
129
130 u_redim_A: redimensionar port map ( unsigned(mant_A),
131                                     mant_Ar );
132
133 u_redim_B: redimensionar port map ( unsigned(mant_B),
134                                     mant_Br );
135
136 u_corrimiento: corrimiento  generic map (8,40)
137                             port map (  mant_Br,
138                                         shamt,
139                                         '0',
140                                         mant_Brc);
141
142 u_comparador_mant: comparador   generic map (40)
143                                 port map (  mant_Ar,
144                                             mant_Brc,
145                                             mant_comp);
146
147 u_mux_mant_Ad: mux2a1_nb    generic map (40)
148                             port map (  std_logic_vector(mant_Ar),
149                                         std_logic_vector(mant_Brc),
150                                         mant_comp, mant_Ad);
151
152 u_mux_mant_Bd: mux2a1_nb    generic map (40)
153                             port map (  std_logic_vector(mant_Ar),
154                                         std_logic_vector(mant_Brc),
155                                         mant_comp_n, mant_Bd);
156
157 u_sum_rest: sumador_restador    generic map (40)
158                                 port map (  unsigned(mant_Ad),
159                                             unsigned(mant_Bd),
160                                             mant_sel_n, mant_S );
161
162 u_norm: normalizar port map ( unsigned(exp_A), mant_S, St);
163
164 process(exp_X, exp_Y, signo_X, signo_Y, exp_comp, exp_comp_n, mant_comp, mant_comp_n)
165 begin
166     if (exp_X = exp_Y) then
167        S(31) <= ( signo_X and mant_comp ) or ( signo_Y and mant_comp_n);
168     else
169        S(31) <= ( signo_X and exp_comp ) or ( signo_Y and exp_comp_n);
```

```
170     end if;
171 end process;
172 
173 S(30 downto 0) <= std_logic_vector(St);
174 
175 end estructural;
```

C.1.2. Multiplicador de punto flotante de 32 bits

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4  
5  entity multiplicador_pf32 is
6  Port (  X : in STD_LOGIC_VECTOR (31 downto 0);
7                  Y : in STD_LOGIC_VECTOR (31 downto 0);
8                  S : out STD_LOGIC_VECTOR (31 downto 0));
9  end multiplicador_pf32;
10 
11 architecture estructural of multiplicador_pf32 is
12 
13 component comparador is
14 generic( N: integer range 1 to 64 := 32 );
15 port(   X, Y: in  unsigned( N-1 downto 0 );
16         c:    out std_logic );
17 end component;
18 
19 component detector_zero is
20 generic( N: integer range 1 to 64 := 32 );
21 port(   X: in  unsigned( N-1 downto 0 );
22         z: out std_logic );
23 end component;
24 
25 component multiplicador is
26 generic( N: integer range 1 to 64 := 32 );
27 port(   X, Y: in  unsigned( N-1 downto 0 );
28         M:    out unsigned( 2*N-1 downto 0 ) );
29 end component;
30 
31 component mux2a1_nb is
32 generic(N: integer range 1 to 64 := 32 );
33 port(   D1, D0: in  std_logic_vector(N-1 downto 0);
34         s:      in  std_logic;
35         Y:      out std_logic_vector(N-1 downto 0));
36 end component;
37 
38 component normalizar_mult is
39 port(   sEa, sEb, sE: in std_logic;
40         Ea, Eb, Ee:    in  unsigned( 7 downto 0 );
41         X:                 in  unsigned( 47 downto 0 );
42         Y:             out unsigned( 30 downto 0 ) );
43 end component;
44 
45 component restador is
46 generic( N: integer range 1 to 64 := 32 );
47 port(   X, Y: in  unsigned( N-1 downto 0 );
48         R:    out unsigned( N-1 downto 0 ) );
49 end component;
```

```
50
51 component sumador_restador is
52 generic( N: integer range 1 to 64 := 32 );
53 port(   X, Y: in  unsigned( N-1 downto 0 );
54         s: in std_logic;
55         F:    out unsigned( N-1 downto 0 ) );
56 end component;
57
58 signal signo_X, signo_Y: std_logic;
59 signal exp_X, exp_Y: unsigned(7 downto 0);
60 signal expX_comp, expY_comp: std_logic;
61 signal expX_comp_n, expY_comp_n: std_logic;
62 signal exp_comp, exp_comp_n: std_logic;
63 signal exp_A0, exp_A1: std_logic_vector(7 downto 0);
64 signal exp_B0, exp_B1: std_logic_vector(7 downto 0);
65 signal exp_A, exp_B, exp_S: unsigned(7 downto 0);
66 signal exp_Ad, exp_Bd: std_logic_vector(7 downto 0);
67 signal op_sel: std_logic;
68 signal mant_X, mant_Y: unsigned(23 downto 0);
69 signal mant_S: unsigned(47 downto 0);
70 signal zX, zY: std_logic;
71 signal St: unsigned(30 downto 0);
72
73 constant k: unsigned(7 downto 0) := "01111111";
74
75 begin
76
77 signo_X <= X(31);
78 signo_Y <= Y(31);
79
80 exp_X <= unsigned(X(30 downto 23));
81 exp_Y <= unsigned(Y(30 downto 23));
82
83 mant_X <= unsigned(not(zX) & X(22 downto 0));
84 mant_Y <= unsigned(not(zY) & Y(22 downto 0));
85
86 expX_comp_n <= not(expX_comp);
87 expY_comp_n <= not(expY_comp);
88 exp_comp_n <= not(exp_comp);
89
90 op_sel <= expX_comp_n xnor expY_comp_n;
91
92 u_comp_expX: comparador generic map (8)
93 port map (exp_X, k, expX_comp);
94
95 u_comp_expY: comparador generic map (8)
96 port map (exp_Y, k, expY_comp);
97
98 u_mux0_exp_A: mux2a1_nb generic map (8)
99 port map (std_logic_vector(exp_X), std_logic_vector(k), expX_comp, exp_A0);
100
101 u_mux1_exp_A: mux2a1_nb generic map (8)
102 port map (std_logic_vector(exp_X), std_logic_vector(k), expX_comp_n, exp_A1);
103
104 u_mux0_exp_B: mux2a1_nb generic map (8)
105 port map (std_logic_vector(exp_Y), std_logic_vector(k), expY_comp, exp_B0);
106
107 u_mux1_exp_B: mux2a1_nb generic map (8)
108 port map (std_logic_vector(exp_Y), std_logic_vector(k), expY_comp_n, exp_B1);
109
110 u_restadorA: restador generic map (8)
```

```
port map (unsigned(exp_A0), unsigned(exp_A1), exp_A);

u_restadorB: restador generic map (8)
port map (unsigned(exp_B0), unsigned(exp_B1), exp_B);

u_comp_exp: comparador generic map (8)
port map (exp_A, exp_B, exp_comp);

u_mux_exp_A: mux2a1_nb generic map (8)
port map (std_logic_vector(exp_A), std_logic_vector(exp_B), exp_comp, exp_Ad);

u_mux_exp_B: mux2a1_nb generic map (8)
port map (std_logic_vector(exp_A), std_logic_vector(exp_B), exp_comp_n, exp_Bd);

u_sum_rest: sumador_restador generic map (8)
port map (unsigned(exp_Ad), unsigned(exp_Bd), op_sel, exp_S );

u_dz1: detector_zero port map (unsigned(X),zX);

u_dz2: detector_zero port map (unsigned(Y),zY);

u_mult: multiplicador generic map (24)
port map (mant_X, mant_Y, mant_S);

u_norm: normalizar_mult port map ( expX_comp, expY_comp, exp_comp,
exp_X, exp_Y, exp_S, mant_S, St );

S(31) <= signo_X xor signo_Y;
S(30 downto 0) <= std_logic_vector(St);

end estructural;
```

C.1.3. Elemento de proceso esténcil ecuación de calor 1D

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ep_heat is
    Port ( X_0N : in STD_LOGIC_VECTOR (31 downto 0);
           X_1N : in STD_LOGIC_VECTOR (31 downto 0);
           X_2N : in STD_LOGIC_VECTOR (31 downto 0);
           K_1 : in STD_LOGIC_VECTOR (31 downto 0);
           K_2 : in STD_LOGIC_VECTOR (31 downto 0);
           X_1Np : out STD_LOGIC_VECTOR (31 downto 0));
end ep_heat;

architecture estructural of ep_heat is

component multiplicador_pf32 is
    Port ( X : in STD_LOGIC_VECTOR (31 downto 0);
           Y : in STD_LOGIC_VECTOR (31 downto 0);
           S : out STD_LOGIC_VECTOR (31 downto 0));
end component multiplicador_pf32;

component sumador_pf32 is
    Port ( X : in STD_LOGIC_VECTOR (31 downto 0);
           Y : in STD_LOGIC_VECTOR (31 downto 0);
           S : out STD_LOGIC_VECTOR (31 downto 0));
```

```
25 end component sumador_pf32;
26
27 signal Suma, Mult0, Mult1: STD_LOGIC_VECTOR (31 downto 0);
28
29 begin
30
31 u_sumpf_in: sumador_pf32 port map (X_0N, X_2N, Suma);
32 u_multpf_0: multiplicador_pf32 port map (Suma, K_1, Mult0);
33 u_multpf_1: multiplicador_pf32 port map (X_1N, K_2, Mult1);
34 u_sumpf_out: sumador_pf32 port map (Mult0, Mult1, X_1Np);
35
36 end estructural;
```

C.1.4. Unidad de control arquitectura base ecuación de calor 1D

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity unidad_control_streaming_65536 is
6  port(   control:        in  std_logic_vector(31 downto 0);
7          N:              in  std_logic_vector(15 downto 0);
8          L:              in  std_logic_vector(15 downto 0);
9          clk:            in  std_logic;
10         data_src:       out std_logic;
11         r_wr_en:                out std_logic;
12         r_clr:          out std_logic;
13         ram_address:    out std_logic_vector(15 downto 0);
14         ram_wr_en:      out std_logic_vector(0 to 0);
15         clr:                    out std_logic;
16         n_estados:      out std_logic_vector(3 downto 0);
17         e_control:      out std_logic );
18 end entity;
19
20 architecture funcional of unidad_control_streaming_65536 is
21
22 signal kj: unsigned(15 downto 0);
23 signal kn: unsigned(15 downto 0);
24 signal ka: unsigned(15 downto 0);
25 type estados is (e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, e10);
26 signal estado: estados;
27
28 begin
29
30 clr <= control(30);
31 r_clr <= control(30);
32
33 process(clk, control, estado, L, N, ka, kj, kn)
34 begin
35         if ( control(30) = '1' ) then
36                 ka       <= ( others => '0' );
37                 kj       <= ( others => '0' );
38                 kn     <= ( others => '0' );
39                 estado <= e0;
40         elsif( rising_edge(clk) ) then
41
42                 case estado is
43
```

```
44                          when e0 =>
45                               ka <= ( others => '0' );
46                               kn <= ( others => '0' );
47                               kj <= ( others => '0' );
48                               if ( control(31) = '1' ) then
49                                      estado <= e1;
50                               else
51                                      estado <= e0;
52                               end if;
53
54                          when e1 =>
55                               if (kn < unsigned(N)-1) then
56                                      estado <= e2;
57                               else
58                                      kn     <= (others => '0');
59                                      estado <= e10;
60                               end if;
61
62                          when e2 =>
63                               if (kj    < unsigned(L)) then
64                                      estado <= e3;
65                               else
66                                      kj     <= (others => '0');
67                                      kn         <= kn + 1;
68                                      estado <= e1;
69                               end if;
70
71                          when e3 =>
72                               estado <= e4;
73
74                          when e4 =>
75                               if (kj < 2) then
76                                  ka     <= ka + 1;
77                                  kj     <= kj + 1;
78                                  estado <= e2;
79                               else
80                                  ka         <= ka + unsigned(L);
81                                  estado <= e5;
82                               end if;
83
84                          when e5 =>
85                               ka     <= ka - 1;
86                               estado <= e6;
87
88                          when e6 =>
89                               estado <= e7;
90
91                          when e7 =>
92                               estado <= e8;
93
94                          when e8 =>
95                               ka <= ka - unsigned(L);
96                               estado <= e9;
97
98                          when e9 =>
99                               ka     <= ka + 2;
100                              kj     <= kj + 1;
101                              estado <= e2;
102
103                         when e10 =>
104                              if ( control(31) = '1' ) then
```

```
105                                         estado <= e10;
106                                 else
107                                         estado <= e0;
108                                 end if;
109
110                 end case;
111
112         end if;
113
114 end process;
115
116 with estado select data_src <=  '0' when e0,
117                                                 '1' when others;
118
119 with estado select ram_wr_en(0) <=      control(16) when e0,
120                                         '1' when e7,
121                                         '0' when others;
122
123 with estado select ram_address <= control(15 downto 0) when e0,
124                         std_logic_vector(ka(15 downto 0)) when others;
125
126 with estado select r_wr_en <=   '1' when e3,
127                                                 '0' when others;
128
129 with estado select n_estados <=  "0000" when e0,
130                                  "0001" when e1,
131                                  "0010" when e2,
132                                  "0011" when e3,
133                                  "0100" when e4,
134                                  "0101" when e5,
135                                  "0110" when e6,
136                                  "0111" when e7,
137                                  "1000" when e8,
138                                  "1001" when e9,
139                                  "1010" when e10;
140
141 with estado select e_control <= '0' when e0,
142                                 '1' when others;
143
144 end architecture;
```

C.1.5. Contador de ciclos de reloj (latencia)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity performance_counter is
6  port(   EN1, EN2: in  std_logic;
7          clk, clr: in  std_logic;
8          C1: out std_logic_vector(63 downto 0);
9          C2:   out std_logic_vector(31 downto 0));
10 end entity;
11
12 architecture funcional of performance_counter is
13
14 signal k1: unsigned(63 downto 0) := ( others => '0' );
15 signal k2: unsigned(31 downto 0) := ( others => '0' );
```

```
16
17 begin
18
19 process(clr, clk, EN1)
20 begin
21         if ( clr = '1' ) then
22                 k1 <= ( others => '0' );
23         elsif ( EN1 = '1' ) then
24                 if ( rising_edge(clk) ) then
25                   k1 <= k1 + 1;
26                 end if;
27         end if;
28 end process;
29
30 process(clr, clk, EN2)
31 begin
32         if ( clr = '1' ) then
33                 k2 <= ( others => '0' );
34         elsif ( EN2 = '1' ) then
35                 if ( rising_edge(clk) ) then
36                   k2 <= k2 + 1;
37                 end if;
38         end if;
39 end process;
40
41 C1 <= std_logic_vector(k1);
42 C2 <= std_logic_vector(k2);
43
44 end architecture;
```

C.1.6. Codigo C para el host

```
1  #include "platform.h"
2  #include "xbasic_types.h"
3  #include "xparameters.h"
4  #include "xgpio.h"
5  #include "ZedboardOLED.h"
6  #include "xscugic.h"
7  #include "xil_exception.h"
8  #include "xsdps.h"
9  #include "xil_printf.h"
10 #include "ff.h"
11 #include "xil_cache.h"
12
13 static FATFS FS_instance; // File System instance
14 static FIL file1;    // File instance
15 FRESULT result;      // FRESULT variable
16 static char FileName[32] = "RECORDS.txt"; // name of the log
17 static char *Log_File; // pointer to the log
18 char *Path = "0:/";  //  string pointer to the logical drive number
19 unsigned int BytesWr, BytesRd;
20 u8 Buffer_logger[32] __attribute__ ((aligned(32))); // Buffer should be word aligned (
       multiple of 4)
21 u32 Buffer_size = 32, line_size;
22
23 Xuint32 *baseaddr_p = (Xuint32 *)XPAR_HEAT_EQ_0_S00_AXI_BASEADDR;
24
25 int main (void)
```

```
26 {
27   float x=0, dx=0, ic;
28   float k1 = 0.25, k2;
29   float l = 32;
30   unsigned int N = 512, L = 32;
31   unsigned int k, kn, kj;
32   unsigned int C1m, C1l, C2;
33   unsigned int estado = 0x00000001;
34
35   //xil_printf("Heat Equation Test\n\r");
36   printf("Heat Equation Test\n");
37
38   *(baseaddr_p+1) = 0x40000000;
39   *(baseaddr_p+1) = 0x20000000;
40
41   k2 = 1-2*k1;
42   dx = 1/(l-1);
43
44   init_platform();
45
46   // Mount SD Card and initialize device
47   result = f_mount(&FS_instance,Path, 0);
48
49   // Open log for writing
50   Log_File = (char *)FileName;
51   result = f_open(&file1, Log_File, FA_WRITE | FA_CREATE_ALWAYS);
52
53   *(baseaddr_p+2) = *((unsigned int*)&k1);
54   *(baseaddr_p+3) = *((unsigned int*)&k2);
55   *(baseaddr_p+4) = N;
56   *(baseaddr_p+5) = L;
57
58   *(baseaddr_p+0) = 0x00000000;
59
60   for ( kn = 0; kn < N; kn++ )
61   {
62     *(baseaddr_p+1) = 0x20010000 + kn*L;
63     *(baseaddr_p+1) = 0x2000FFFF + (kn+1)*L;
64   }
65
66   for ( kj = 0; kj < L; kj++ )
67   {
68     if ( x < 0.5 )
69       ic = 2*x;
70     else
71       ic = 2*(1-x);
72
73     *(baseaddr_p+1) = 0x20010000 + kj;
74     *(baseaddr_p+0) = *((unsigned int*)&ic);
75
76     x = x + dx;
77   }
78
79   *(baseaddr_p+1) = 0xB0000000;
80   *(baseaddr_p+1) = 0x30000000;
81
82   while ( estado == 0x00000001 )
83   {
84     *(baseaddr_p+1) = 0x20000000;
85     estado = *(baseaddr_p+7) && 0x0000000F;
86     //printf("%X\n",estado);
```

```
87    }
88
89    *(baseaddr_p+1) = 0x20000000;
90
91    for ( kn = 0; kn < N; kn++ )
92    {
93      for ( kj = 0; kj < L; kj++ )
94      {
95        k = L*kn + kj;
96        *(baseaddr_p+1) = 0x20000000 + k;
97        if ( kj < L-1 )
98        {
99          sprintf(Buffer_logger, "%1.15f                ", *((float*)&(*(baseaddr_p+6))) );
100         result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
101       }
102       else
103       {
104         sprintf(Buffer_logger, "%1.15f              \n", *((float*)&(*(baseaddr_p+6))) );
105         result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
106       }
107     }
108   }
109
110   //Close file.
111   result = f_close(&file1);
112
113   result = f_mount(NULL, Path, 0);
114
115   *(baseaddr_p+1) = 0x00000000;
116
117   printf("Data written to log Successfully\n");
118
119   C1m = *(baseaddr_p+8);
120   C1l = *(baseaddr_p+9);
121   C2  = *(baseaddr_p+10);
122
123   printf("Medidas de rendimiento\n");
124   printf("Numero de ciclos total: %8X%8X\n\r",C1m,C1l);
125   printf("MSB: %8X\n\r",C1m);
126   printf("LSB: %8X\n\r",C1l);
127   printf("Tiempo total: %u ns\n",10*C1l);
128   printf("Numero de ciclos de proceso: %X (%d ns)\n\r",C2,10*C2);
129
130   printf("Heat Equation Test Finished\n\r");
131   return 0;
132 }
```

C.2. Arquitectura paralela ecuación de calor: 14 EP

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity c_up_heat_concurrent_ram65536x1_rx16 is
6  Port (  data_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
7                  clk              : IN STD_LOGIC;
8                  control : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
9                  K1, K2  : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
10                 N                : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
11                 L                : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
12                 RESULTADOS       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
13                 estado   : INOUT STD_LOGIC;
14                 n_estados        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
15                 C1               : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
16                 C2               : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
17 end c_up_heat_concurrent_ram65536x1_rx16;
18
19 architecture estructural of c_up_heat_concurrent_ram65536x1_rx16 is
20
21 component banco_registros_x16r is
22 port (  data_in    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
23                 DB0,  DB1   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
24                 DB2,  DB3   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
25                 DB4,  DB5   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
26                 DB6,  DB7   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
27                 DB8,  DB9   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
28                 DB10, DB11 : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
29                 DB12, DB13 : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
30                 DB14, DB15       : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
31                 clk              : IN  STD_LOGIC;
32                 datai_src   : IN  STD_LOGIC;
33         r_wr_en    : IN  STD_LOGIC;
34                 r_clr       : IN  STD_LOGIC;
35                 datao_src   : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
36                 Q           : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
37                 Q0,  Q1     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
38                 Q2,  Q3     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
39                 Q4,  Q5     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
40                 Q6,  Q7     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
41                 Q8,  Q9     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
42                 Q10, Q11    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
43                 Q12, Q13    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
44                 Q14, Q15    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
45 end component banco_registros_x16r;
46
47 component ep_heat is
48 port ( X_0N  : in STD_LOGIC_VECTOR (31 downto 0);
49        X_1N  : in STD_LOGIC_VECTOR (31 downto 0);
50        X_2N  : in STD_LOGIC_VECTOR (31 downto 0);
51        K_1   : in STD_LOGIC_VECTOR (31 downto 0);
52        K_2   : in STD_LOGIC_VECTOR (31 downto 0);
53        X_1Np : out STD_LOGIC_VECTOR (31 downto 0));
54 end component;
55
56 component mux2a1_nb is
57 generic(N: integer range 1 to 64 := 32 );
58 port(   D1, D0   : in  std_logic_vector(N-1 downto 0);
59        s         : in  std_logic;
```

```
60          Y         : out std_logic_vector(N-1 downto 0));
61  end component;
62
63  component sp_bram_65536x32 is
64  port (  addra : in STD_LOGIC_VECTOR ( 15 downto 0 );
65                  clka : in STD_LOGIC;
66                  dina : in STD_LOGIC_VECTOR ( 31 downto 0 );
67                  douta : out STD_LOGIC_VECTOR ( 31 downto 0 );
68                  wea : in STD_LOGIC_VECTOR ( 0 to 0 ));
69  end component sp_bram_65536x32;
70
71  component unidad_control_concurrente is
72  generic(K: integer range 0 to 8 := 4);
73  port(   control:      in  std_logic_vector(31 downto 0);
74          N:            in  std_logic_vector(15 downto 0);
75          L:            in  std_logic_vector(15 downto 0);
76          clk:             in  std_logic;
77          ram_data_src: out std_logic;
78          ram_address:  out std_logic_vector(15 downto 0);
79          ram_wr_en:    out std_logic;
80          datai_src:    out std_logic;
81          r_wr_en:         out std_logic;
82          r_clr:           out std_logic;
83          datao_src:    out std_logic_vector(K-1 downto 0);
84          n_estados:    out std_logic_vector(3 downto 0);
85          e_control:    out std_logic );
86  end component unidad_control_concurrente;
87
88  component performance_counter is
89  port(   EN1, EN2: in  std_logic;
90          clk, clr: in  std_logic;
91          C1:       out std_logic_vector(63 downto 0);
92          C2:       out std_logic_vector(31 downto 0));
93  end component;
94
95  signal ram_data_in  :   std_logic_vector(31 downto 0);
96  signal ram_data_out :   std_logic_vector(31 downto 0);
97  signal data_back    :   std_logic_vector(31 downto 0);
98  signal   X0,   X1   :   std_logic_vector(31 downto 0);
99  signal   X2,   X3   :   std_logic_vector(31 downto 0);
100 signal   X4,   X5   :   std_logic_vector(31 downto 0);
101 signal   X6,   X7   :   std_logic_vector(31 downto 0);
102 signal   X8,   X9   :   std_logic_vector(31 downto 0);
103 signal  X10,  X11   :   std_logic_vector(31 downto 0);
104 signal  X12,  X13   :   std_logic_vector(31 downto 0);
105 signal  X14,  X15   :   std_logic_vector(31 downto 0);
106 signal  DB0,  DB1   :   std_logic_vector(31 downto 0);
107 signal  DB2,  DB3   :   std_logic_vector(31 downto 0);
108 signal  DB4,  DB5   :   std_logic_vector(31 downto 0);
109 signal  DB6,  DB7   :   std_logic_vector(31 downto 0);
110 signal  DB8,  DB9   :   std_logic_vector(31 downto 0);
111 signal DB10, DB11   :   std_logic_vector(31 downto 0);
112 signal DB12, DB13   :   std_logic_vector(31 downto 0);
113 signal DB14, DB15   :   std_logic_vector(31 downto 0);
114 signal ram_data_src :   std_logic;
115 signal datai_src    :   std_logic;
116 signal datao_src    :   std_logic_vector(3 downto 0);
117 signal r_wr_en      :   std_logic;
118 signal r_clr        :   std_logic;
119 signal ram_address  :   std_logic_vector(15 downto 0);
120 signal ram_wr_en    :   std_logic;
```

```
121 signal ram_wren     :   std_logic_vector(0 to 0);
122 signal clk_n, clr   :   std_logic;
123
124 begin
125
126 clk_n <= not(clk);
127 ram_wren(0) <= ram_wr_en;
128
129 u_uc: unidad_control_concurrente port map (control,
130                 N, L, clk, ram_data_src,
131                 ram_address, ram_wr_en, datai_src,
132                 r_wr_en, r_clr,
133                 datao_src, n_estados, estado);
134
135 u_mux: mux2a1_nb port map (data_back, data_in,
136                                 ram_data_src, ram_data_in);
137
138 u_ram: sp_bram_65536x32 port map (ram_address, clk_n,
139                         ram_data_in, ram_data_out, ram_wren);
140
141 u_br: banco_registros_x16r port map ( ram_data_out,
142                                       DB0,
143                                       DB1,
144                                       DB2,
145                                       DB3,
146                                       DB4,
147                                       DB5,
148                                       DB6,
149                                       DB7,
150                                       DB8,
151                                       DB9,
152                                       DB10,
153                                       DB11,
154                                       DB12,
155                                       DB13,
156                                       DB14,
157                                       DB15,
158                                       clk,
159                                       datai_src,
160                                       r_wr_en,
161                                       r_clr,
162                                       datao_src,
163                                       data_back,
164                                       X0,
165                                       X1,
166                                       X2,
167                                       X3,
168                                       X4,
169                                       X5,
170                                       X6,
171                                       X7,
172                                       X8,
173                                       X9,
174                                       X10,
175                                       X11,
176                                       X12,
177                                       X13,
178                                       X14,
179                                       X15 );
180
181 u_ep1:  ep_heat port map ( X0,  X1,  X2, K1, K2,  DB1);
```

```
182 u_ep2:  ep_heat port map ( X1,  X2,  X3, K1, K2,  DB2);
183 u_ep3:  ep_heat port map ( X2,  X3,  X4, K1, K2,  DB3);
184 u_ep4:  ep_heat port map ( X3,  X4,  X5, K1, K2,  DB4);
185 u_ep5:  ep_heat port map ( X4,  X5,  X6, K1, K2,  DB5);
186 u_ep6:  ep_heat port map ( X5,  X6,  X7, K1, K2,  DB6);
187 u_ep7:  ep_heat port map ( X6,  X7,  X8, K1, K2,  DB7);
188 u_ep8:  ep_heat port map ( X7,  X8,  X9, K1, K2,  DB8);
189 u_ep9:  ep_heat port map ( X8,  X9, X10, K1, K2,  DB9);
190 u_ep10: ep_heat port map ( X9, X10, X11, K1, K2, DB10);
191 u_ep11: ep_heat port map (X10, X11, X12, K1, K2, DB11);
192 u_ep12: ep_heat port map (X11, X12, X13, K1, K2, DB12);
193 u_ep13: ep_heat port map (X12, X13, X14, K1, K2, DB13);
194 u_ep14: ep_heat port map (X13, X14, X15, K1, K2, DB14);
195 u_k:   performance_counter port map ( estado, estado, clk, control(30), C1, C2 );
196
197 DB0  <=  X0;
198 DB15 <= X15;
199
200 RESULTADOS <= ram_data_out;
201
202 end estructural;
```

C.2.1. Banco de registros x16

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity banco_registros_x16r is
6  port (  data_in    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
7          DB0,  DB1  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
8                  DB2,  DB3  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
9                  DB4,  DB5  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
10                 DB6,  DB7  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
11                 DB8,  DB9  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
12                 DB10, DB11 : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
13                 DB12, DB13 : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
14                 DB14, DB15 : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
15                 clk                : IN  STD_LOGIC;
16                 datai_src  : IN  STD_LOGIC;
17                 r_wr_en    : IN  STD_LOGIC;
18                 r_clr      : IN  STD_LOGIC;
19                 datao_src  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
20                 Q          : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
21                 Q0,  Q1    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
22                 Q2,  Q3    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
23                 Q4,  Q5    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
24                 Q6,  Q7    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
25                 Q8,  Q9    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
26                 Q10, Q11   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
27                 Q12, Q13   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
28                 Q14, Q15   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
29 end banco_registros_x16r;
30
31 architecture estructural of banco_registros_x16r is
32
33 component mux2a1_nb is
34 generic(N: integer range 1 to 64 := 32 );
```

```
35 port(   D1, D0: in  std_logic_vector(N-1 downto 0);
36         s:      in  std_logic;
37         Y:      out std_logic_vector(N-1 downto 0));
38 end component;
39
40 component mux16a1_nb is
41 port(   I0,  I1,  I2,  I3:      in  std_logic_vector(31 downto 0);
42         I4,  I5,  I6,  I7:      in  std_logic_vector(31 downto 0);
43         I8,  I9,  I10, I11:     in  std_logic_vector(31 downto 0);
44      I12, I13, I14, I15:        in  std_logic_vector(31 downto 0);
45         S:                          in  std_logic_vector(3 downto 0);
46         Y:                          out std_logic_vector(31 downto 0) );
47 end component;
48
49 component registro is
50 generic(N: integer range 1 to 32 := 32 );
51 port(   D:   in  std_logic_vector(N-1 downto 0);
52         clk: in  std_logic;
53         en, clr: in  std_logic;
54         Q:   out std_logic_vector(N-1 downto 0));
55 end component;
56
57 signal D0,  D1:  std_logic_vector(31 downto 0);
58 signal D2,  D3:  std_logic_vector(31 downto 0);
59 signal D4,  D5:  std_logic_vector(31 downto 0);
60 signal D6,  D7:  std_logic_vector(31 downto 0);
61 signal D8,  D9:  std_logic_vector(31 downto 0);
62 signal D10, D11: std_logic_vector(31 downto 0);
63 signal D12, D13: std_logic_vector(31 downto 0);
64 signal D14, D15: std_logic_vector(31 downto 0);
65 signal X0,  X1: std_logic_vector(31 downto 0);
66 signal X2,  X3: std_logic_vector(31 downto 0);
67 signal X4,  X5: std_logic_vector(31 downto 0);
68 signal X6,  X7: std_logic_vector(31 downto 0);
69 signal X8,  X9: std_logic_vector(31 downto 0);
70 signal X10, X11: std_logic_vector(31 downto 0);
71 signal X12, X13: std_logic_vector(31 downto 0);
72 signal X14, X15: std_logic_vector(31 downto 0);
73
74 begin
75
76 u_mux_r0:  mux2a1_nb port map ( DB0,  X1, datai_src, D0);
77 u_mux_r1:  mux2a1_nb port map ( DB1,  X2, datai_src, D1);
78 u_mux_r2:  mux2a1_nb port map ( DB2,  X3, datai_src, D2);
79 u_mux_r3:  mux2a1_nb port map ( DB3,  X4, datai_src, D3);
80 u_mux_r4:  mux2a1_nb port map ( DB4,  X5, datai_src, D4);
81 u_mux_r5:  mux2a1_nb port map ( DB5,  X6, datai_src, D5);
82 u_mux_r6:  mux2a1_nb port map ( DB6,  X7, datai_src, D6);
83 u_mux_r7:  mux2a1_nb port map ( DB7,  X8, datai_src, D7);
84 u_mux_r8:  mux2a1_nb port map ( DB8,  X9, datai_src, D8);
85 u_mux_r9:  mux2a1_nb port map ( DB9, X10, datai_src, D9);
86 u_mux_r10: mux2a1_nb port map (DB10, X11, datai_src, D10);
87 u_mux_r11: mux2a1_nb port map (DB11, X12, datai_src, D11);
88 u_mux_r12: mux2a1_nb port map (DB12, X13, datai_src, D12);
89 u_mux_r13: mux2a1_nb port map (DB13, X14, datai_src, D13);
90 u_mux_r14: mux2a1_nb port map (DB14, X15, datai_src, D14);
91 u_mux_r15: mux2a1_nb port map (DB15, data_in, datai_src, D15);
92
93 u_r0:  registro port map (  D0, clk, r_wr_en, r_clr,  X0);
94 u_r1:  registro port map (  D1, clk, r_wr_en, r_clr,  X1);
95 u_r2:  registro port map (  D2, clk, r_wr_en, r_clr,  X2);
```

```
96  u_r3:  registro port map (  D3, clk, r_wr_en, r_clr,  X3);
97  u_r4:  registro port map (  D4, clk, r_wr_en, r_clr,  X4);
98  u_r5:  registro port map (  D5, clk, r_wr_en, r_clr,  X5);
99  u_r6:  registro port map (  D6, clk, r_wr_en, r_clr,  X6);
100 u_r7:  registro port map (  D7, clk, r_wr_en, r_clr,  X7);
101 u_r8:  registro port map (  D8, clk, r_wr_en, r_clr,  X8);
102 u_r9:  registro port map (  D9, clk, r_wr_en, r_clr,  X9);
103 u_r10: registro port map ( D10, clk, r_wr_en, r_clr, X10);
104 u_r11: registro port map ( D11, clk, r_wr_en, r_clr, X11);
105 u_r12: registro port map ( D12, clk, r_wr_en, r_clr, X12);
106 u_r13: registro port map ( D13, clk, r_wr_en, r_clr, X13);
107 u_r14: registro port map ( D14, clk, r_wr_en, r_clr, X14);
108 u_r15: registro port map ( D15, clk, r_wr_en, r_clr, X15);
109
110 u_mux_o: mux16a1_nb port map( D0, D1, D2, D3, D4, D5, D6, D7,
111                 D8, D9, D10, D11, D12, D13, D14, D15,
112                 datao_src, Q);
113
114  Q0 <=  X0;
115  Q1 <=  X1;
116  Q2 <=  X2;
117  Q3 <=  X3;
118  Q4 <=  X4;
119  Q5 <=  X5;
120  Q6 <=  X6;
121  Q7 <=  X7;
122  Q8 <=  X8;
123  Q9 <=  X9;
124 Q10 <= X10;
125 Q11 <= X11;
126 Q12 <= X12;
127 Q13 <= X13;
128 Q14 <= X14;
129 Q15 <= X15;
130
131 end estructural;
```

C.2.2. Unidad de control arquitectura paralela 14EP, ecuación de calor 1D

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity unidad_control_concurrente is
6  generic(K: integer range 0 to 8 := 4);
7  port(   control:        in  std_logic_vector(31 downto 0);
8          N:              in  std_logic_vector(15 downto 0);
9          L:              in  std_logic_vector(15 downto 0);
10         clk:            in  std_logic;
11         ram_data_src:   out std_logic;
12         ram_address:    out std_logic_vector(15 downto 0);
13      ram_wr_en:         out std_logic;
14         datai_src:      out std_logic;
15         r_wr_en:                out std_logic;
16         r_clr:          out std_logic;
17         datao_src:      out std_logic_vector(K-1 downto 0);
```

```
18          n_estados:      out std_logic_vector(3 downto 0);
19          e_control:      out std_logic );
20  end entity unidad_control_concurrente;
21
22  architecture funcional of unidad_control_concurrente is
23
24  signal kj: unsigned(K downto 0);
25  signal kn: unsigned(15 downto 0);
26  signal ka: unsigned(15 downto 0);
27  type estados is (e0, e1, e2, e3, e4, e5, e6, e7, e8, e9);
28  signal estado: estados;
29
30  signal u_N: unsigned(15 downto 0);
31  signal u_L: unsigned(15 downto 0);
32
33  begin
34
35  u_N <= unsigned(N);
36  u_L <= unsigned(L);
37
38  process(clk, control, estado, u_L, u_N, ka, kj, kn)
39  begin
40          if ( control(30) = '1' ) then
41                  ka       <= ( others => '0' );
42                  kj       <= ( others => '0' );
43                  kn     <= ( others => '0' );
44                  estado <= e0;
45          elsif( rising_edge(clk) ) then
46
47                  case estado is
48
49                          when e0 =>
50                                  ka <= ( others => '0' );
51                                  kn <= ( others => '0' );
52                                  kj <= ( others => '0' );
53                                  if ( control(31) = '1' ) then
54                                          estado <= e1;
55                                  else
56                                          estado <= e0;
57                                  end if;
58
59                          when e1 =>
60                   if (kj < u_L) then
61                       estado <= e2;
62                   else
63                       kj     <= (others => '0');
64                       estado <= e4;
65                   end if;
66
67                          when e2 =>
68                                  estado <= e3;
69
70                          when e3 =>
71                                  ka          <= ka + 1;
72                                  kj     <= kj + 1;
73                                  estado <= e1;
74
75              when e4 =>
76                   if (kn < u_N-2) then
77                      kj  <= (others => '0');
78                      if ( control(28) = '0' ) then
```

```
79                          estado <= e8;
80                      else
81                          estado <= e5;
82                      end if;
83                  else
84                      estado <= e5;
85                  end if;
86
87              when e5 =>
88                  if (kn = u_N) then
89                      kn     <= (others => '0');
90                      estado <= e9;
91                  else
92                      if (kj < u_L) then
93                          estado <= e6;
94                      else
95                          kj     <= (others => '0');
96                          estado <= e8;
97                      end if;
98                  end if;
99
100             when e6 =>
101                 estado <= e7;
102
103             when e7 =>
104                 kj     <= kj + 1;
105                           ka     <= ka + 1;
106                 estado <= e5;
107
108             when e8 =>
109                 kn     <= kn + 1;
110                 estado <= e4;
111
112                         when e9 =>
113                 if ( control(31) = '1' ) then
114                     estado <= e9;
115                 else
116                     estado <= e0;
117                 end if;
118
119                 end case;
120
121         end if;
122
123 end process;
124
125 with estado select ram_data_src <=      '0' when e0,
126                                                     '1' when others;
127
128 with estado select ram_wr_en <= control(16) when e0,
129                                                 '1' when e6,
130                                                 '0' when others;
131
132 with estado select ram_address <= control(15 downto 0) when e0,
133                                       std_logic_vector(ka(15 downto 0))       when others;
134
135 with estado select datai_src <= '0' when e0,
136                                  '0' when e1,
137                                  '0' when e2,
138                                  '0' when e3,
139                                  '1' when others;
```

```
140
141 with estado select r_wr_en <=   '1' when e2,
142                                                 '1' when e8,
143                                                 '0' when others;
144
145 r_clr <= control(30);
146
147 datao_src <= std_logic_vector(kj(K-1 downto 0));
148
149 with estado select n_estados <= "0000" when e0,
150                                                 "0001" when e1,
151                                                 "0010" when e2,
152                                                 "0011" when e3,
153                                                 "0100" when e4,
154                                                 "0101" when e5,
155                                                 "0110" when e6,
156                                                 "0111" when e7,
157                                                 "1000" when e8,
158                                                 "1000" when e9;
159
160 with estado select e_control <= '0' when e0,
161                                                 '1' when others;
162
163 end architecture;
```

C.2.3. Codigo C para el host

```
1  #include "platform.h"
2  #include "xbasic_types.h"
3  #include "xparameters.h"
4  #include "xscugic.h"
5  #include "xil_exception.h"
6  #include "xsdps.h"
7  #include "xil_printf.h"
8  #include "ff.h"
9  #include "xil_cache.h"
10
11 static FATFS FS_instance; // File System instance
12 static FIL file1;   // File instance
13 FRESULT result;     // FRESULT variable
14 static char FileName[32] = "SOL16.txt"; // name of the log
15 static char *Log_File; // pointer to the log
16 char *Path = "0:/";  //  string pointer to the logical drive number
17 unsigned int BytesWr, BytesRd;
18 u8 Buffer_logger[32] __attribute__ ((aligned(32))); // Buffer should be word aligned (
       multiple of 4)
19 u32 Buffer_size = 32, line_size;
20
21 Xuint32 *baseaddr_p = (Xuint32 *)XPAR_HEQ_CRX16_0_S00_AXI_BASEADDR;
22
23 int main (void)
24 {
25   float x=0, dx=0, ic;
26   float k1=0.25, k2;
27   float l = 16;
28   unsigned int N = 512, L = 16;
29   unsigned int k, kn, kj;
30   unsigned int C1m, C1l, C2;
```

```
31  unsigned int estado = 0x00000001, n_estado;
32
33  printf("Heat Equation Test\n");
34
35  init_platform();
36
37  *(baseaddr_p+1) = 0x60000000;
38  *(baseaddr_p+1) = 0x20000000;
39
40  k2 = 1-2*k1;
41  dx = 1/(l-1);
42
43  // Mount SD Card and initialize device
44  result = f_mount(&FS_instance,Path, 0);
45
46  // Open log for writing
47  Log_File = (char *)FileName;
48  result = f_open(&file1, Log_File, FA_WRITE | FA_CREATE_ALWAYS);
49
50  *(baseaddr_p+2) = *((unsigned int*)&k1);
51  *(baseaddr_p+3) = *((unsigned int*)&k2);
52  *(baseaddr_p+4) = N;
53  *(baseaddr_p+5) = L;
54
55  *(baseaddr_p+0) = 0x00000000;
56
57  for ( kn = 0; kn < N; kn++ )
58  {
59    *(baseaddr_p+1) = 0x20010000 + kn*L;
60    *(baseaddr_p+1) = 0x2000FFFF + (kn+1)*L;
61  }
62
63  for ( kj = 0; kj < L; kj++ )
64  {
65    if ( x < 0.5 )
66      ic = 2*x;
67    else
68      ic = 2*(1-x);
69
70    *(baseaddr_p+1) = 0x20010000 + kj;
71    *(baseaddr_p+0) = *((unsigned int*)&ic);
72
73    x = x + dx;
74  }
75
76  //*(baseaddr_p+1) = 0xA0000000;
77  *(baseaddr_p+1) = 0xB0000000;
78
79  while ( estado == 0x00000001 )
80  {
81    //*(baseaddr_p+1) = 0x20000000;
82    *(baseaddr_p+1) = 0x30000000;
83    estado = *(baseaddr_p+7) && 0x0000000F;
84    //printf("%X\n",estado);
85  }
86
87  //printf("%1.5f\n\n", *((float*)&(*(baseaddr_p+6))) );
88
89  for ( kn = 0; kn < N; kn++ )
90  {
91    for ( kj = 0; kj < L; kj++ )
```

```
92      {
93        k = L*kn + kj;
94        *(baseaddr_p+1) = 0x20000000 + k;
95        if ( kj < L-1 )
96        {
97          //printf("%1.5f ", *((float*)&(*(baseaddr_p+6))) );
98          sprintf(Buffer_logger, "%1.15f                ", *((float*)&(*(baseaddr_p+6))) );
99          result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
100       }
101       else
102       {
103         //printf("%1.5f ", *((float*)&(*(baseaddr_p+6))) );
104         sprintf(Buffer_logger, "%1.15f               \n", *((float*)&(*(baseaddr_p+6))) );
105         result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
106       }
107     }
108     //printf("\n");
109   }
110
111   for ( kn = 0; kn < N; kn++ )
112   {
113     for ( kj = 0; kj < L; kj++ )
114     {
115       k = L*kn + kj;
116       *(baseaddr_p+1) = 0x20000000 + k;
117       printf("%1.5f ", *((float*)&(*(baseaddr_p+6))) );
118     }
119     printf("\n");
120   }
121
122   //Close file.
123   result = f_close(&file1);
124
125   result = f_mount(NULL, Path, 0);
126
127   *(baseaddr_p+1) = 0x00000000;
128
129   printf("Data written to log Successfully\r\n");
130
131   C1m = *(baseaddr_p+8);
132   C1l = *(baseaddr_p+9);
133   C2  = *(baseaddr_p+10);
134
135   printf("Medidas de rendimiento\n");
136   printf("Numero de ciclos total: %8X%8X\n",C1m,C1l);
137   printf("MSB: %8X\n",C1m);
138   printf("LSB: %8X\n",C1l);
139   printf("Tiempo total: %u ns\n",10*C1l);
140   printf("Numero de ciclos de proceso: %d (%d ns)\n",C2,10*C2);
141
142   printf("Heat Equation Test Finished\n\r");
143   return 0;
144 }
```

C.3. Arquitectura paralela ecuación de calor: 14 EP, 16 RAM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity c_up_heat_concurrent_ram512x16_rx16 is
6  Port (  data_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
7                  clk             : IN STD_LOGIC;
8                  control : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
9                  K1, K2  : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
10                 N               : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
11                 RESULTADOS : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
12                 R0, R1     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
13                 R2, R3     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
14                 R4, R5     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
15                 R6, R7     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
16                 R8,  R9    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
17                 R10, R11   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
18                 R12, R13   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
19                 R14, R15   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
20                 estado  : INOUT STD_LOGIC;
21                 n_estados  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
22                 C1         : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
23                 C2         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
24 end c_up_heat_concurrent_ram512x16_rx16;
25
26 architecture estructural of c_up_heat_concurrent_ram512x16_rx16 is
27
28 component deco4a16 is
29 port(   I:  in  std_logic_vector(3 downto 0);
30        en: in  std_logic;
31        Y:  out std_logic_vector(15 downto 0));
32 end component;
33
34 component ep_heat is
35 port ( X_0N  : in STD_LOGIC_VECTOR (31 downto 0);
36       X_1N  : in STD_LOGIC_VECTOR (31 downto 0);
37       X_2N  : in STD_LOGIC_VECTOR (31 downto 0);
38       K_1   : in STD_LOGIC_VECTOR (31 downto 0);
39       K_2   : in STD_LOGIC_VECTOR (31 downto 0);
40       X_1Np : out STD_LOGIC_VECTOR (31 downto 0));
41 end component;
42
43 component mux2a1_nb is
44 generic(N: integer range 1 to 64 := 32 );
45 port( D1, D0: in  std_logic_vector(N-1 downto 0);
46         s:      in  std_logic;
47         Y:      out std_logic_vector(N-1 downto 0));
48 end component;
49
50 component mux16a1_nb is
51 port(   I0,  I1,  I2,  I3:      in  std_logic_vector(31 downto 0);
52        I4,  I5,  I6,  I7:      in  std_logic_vector(31 downto 0);
53        I8,  I9,  I10, I11:     in  std_logic_vector(31 downto 0);
54     I12, I13, I14, I15:        in  std_logic_vector(31 downto 0);
55        S:                              in  std_logic_vector(3 downto 0);
56        Y:                              out std_logic_vector(31 downto 0) );
57 end component;
58
59 component sp_ram_512x32 is
60 port (  addra : in STD_LOGIC_VECTOR ( 8 downto 0 );
```

```
61                clka : in STD_LOGIC;
62                dina : in STD_LOGIC_VECTOR ( 31 downto 0 );
63                douta : out STD_LOGIC_VECTOR ( 31 downto 0 );
64                wea : in STD_LOGIC);
65 end component sp_ram_512x32;
66
67 component registro is
68 generic(N: integer range 1 to 32 := 32 );
69 port(   D:       in  std_logic_vector(N-1 downto 0);
70         clk:     in  std_logic;
71         en, clr: in  std_logic;
72         Q:       out std_logic_vector(N-1 downto 0));
73 end component;
74
75 component unidad_control_concurrente_nram_x16 is
76 port(   control:                in  std_logic_vector(31 downto 0);
77         N:                     in  STD_LOGIC_VECTOR(8 downto 0);
78         clk:                   in  std_logic;
79         ram_address:           out STD_LOGIC_VECTOR(8 downto 0);
80      s_ram:                    out std_logic_vector(3 downto 0);
81      ram_parallel_wr_en:       out std_logic;
82      ram_general_wr_en:        out std_logic;
83         datai_src:             out std_logic;
84         r_wr_en:               out std_logic;
85         r_clr:               out std_logic;
86      datao_src:                out std_logic_vector(3 downto 0);
87         n_estados:             out STD_LOGIC_VECTOR(3 downto 0);
88         e_control:             out std_logic );
89 end component unidad_control_concurrente_nram_x16;
90
91 component performance_counter is
92 port(   EN1, EN2: in  std_logic;
93         clk, clr: in  std_logic;
94         C1:       out std_logic_vector(63 downto 0);
95      C2:       out std_logic_vector(31 downto 0));
96 end component;
97
98 signal datai_src:          std_logic;
99 signal datao_src:          std_logic_vector(3 downto 0);
100 signal r_wr_en:            std_logic;
101 signal r_clr:              std_logic;
102 signal ram_address:        STD_LOGIC_VECTOR(8 downto 0);
103 signal ram_wr_en:          std_logic_vector(15 downto 0);
104 signal sram_wr_en:         std_logic_vector(15 downto 0);
105 signal s_ram:              std_logic_vector(3 downto 0);
106 signal ram_parallel_wr_en: std_logic;
107 signal ram_general_wr_en:  std_logic;
108 signal clr:                std_logic;
109
110 signal  X0,  X1: std_logic_vector(31 downto 0);
111 signal  X2,  X3: std_logic_vector(31 downto 0);
112 signal  X4,  X5: std_logic_vector(31 downto 0);
113 signal  X6,  X7: std_logic_vector(31 downto 0);
114 signal  X8,  X9: std_logic_vector(31 downto 0);
115 signal X10, X11: std_logic_vector(31 downto 0);
116 signal X12, X13: std_logic_vector(31 downto 0);
117 signal X14, X15: std_logic_vector(31 downto 0);
118
119 signal  Y0,  Y1: std_logic_vector(31 downto 0);
120 signal  Y2,  Y3: std_logic_vector(31 downto 0);
121 signal  Y4,  Y5: std_logic_vector(31 downto 0);
```

```
122 signal  Y6,  Y7: std_logic_vector(31 downto 0);
123 signal  Y8,  Y9: std_logic_vector(31 downto 0);
124 signal Y10, Y11: std_logic_vector(31 downto 0);
125 signal Y12, Y13: std_logic_vector(31 downto 0);
126 signal Y14, Y15: std_logic_vector(31 downto 0);
127
128 signal  DB0,  DB1: std_logic_vector(31 downto 0);
129 signal  DB2,  DB3: std_logic_vector(31 downto 0);
130 signal  DB4,  DB5: std_logic_vector(31 downto 0);
131 signal  DB6,  DB7: std_logic_vector(31 downto 0);
132 signal  DB8,  DB9: std_logic_vector(31 downto 0);
133 signal DB10, DB11: std_logic_vector(31 downto 0);
134 signal DB12, DB13: std_logic_vector(31 downto 0);
135 signal DB14, DB15: std_logic_vector(31 downto 0);
136
137 signal  QR0,  QR1: std_logic_vector(31 downto 0);
138 signal  QR2,  QR3: std_logic_vector(31 downto 0);
139 signal  QR4,  QR5: std_logic_vector(31 downto 0);
140 signal  QR6,  QR7: std_logic_vector(31 downto 0);
141 signal  QR8,  QR9: std_logic_vector(31 downto 0);
142 signal QR10, QR11: std_logic_vector(31 downto 0);
143 signal QR12, QR13: std_logic_vector(31 downto 0);
144 signal QR14, QR15: std_logic_vector(31 downto 0);
145
146 begin
147
148 u_uc: unidad_control_concurrente_nram_x16 port map (control,
149 N, clk, ram_address, s_ram, ram_parallel_wr_en,
150 ram_general_wr_en, datai_src, r_wr_en, r_clr,
151 datao_src, n_estados, estado);
152
153 u_deco: deco4a16 port map ( s_ram, ram_general_wr_en, sram_wr_en );
154
155 ram_wr_en(15) <= sram_wr_en(15) or ram_parallel_wr_en;
156 ram_wr_en(14) <= sram_wr_en(14) or ram_parallel_wr_en;
157 ram_wr_en(13) <= sram_wr_en(13) or ram_parallel_wr_en;
158 ram_wr_en(12) <= sram_wr_en(12) or ram_parallel_wr_en;
159 ram_wr_en(11) <= sram_wr_en(11) or ram_parallel_wr_en;
160 ram_wr_en(10) <= sram_wr_en(10) or ram_parallel_wr_en;
161 ram_wr_en(9)  <= sram_wr_en(9)  or ram_parallel_wr_en;
162 ram_wr_en(8)  <= sram_wr_en(8)  or ram_parallel_wr_en;
163 ram_wr_en(7)  <= sram_wr_en(7)  or ram_parallel_wr_en;
164 ram_wr_en(6)  <= sram_wr_en(6)  or ram_parallel_wr_en;
165 ram_wr_en(5)  <= sram_wr_en(5)  or ram_parallel_wr_en;
166 ram_wr_en(4)  <= sram_wr_en(4)  or ram_parallel_wr_en;
167 ram_wr_en(3)  <= sram_wr_en(3)  or ram_parallel_wr_en;
168 ram_wr_en(2)  <= sram_wr_en(2)  or ram_parallel_wr_en;
169 ram_wr_en(1)  <= sram_wr_en(1)  or ram_parallel_wr_en;
170 ram_wr_en(0)  <= sram_wr_en(0)  or ram_parallel_wr_en;
171
172 u_r0:  registro port map ( DB0,  clk, r_wr_en, r_clr, QR0 );
173 u_r1:  registro port map ( DB1,  clk, r_wr_en, r_clr, QR1 );
174 u_r2:  registro port map ( DB2,  clk, r_wr_en, r_clr, QR2 );
175 u_r3:  registro port map ( DB3,  clk, r_wr_en, r_clr, QR3 );
176 u_r4:  registro port map ( DB4,  clk, r_wr_en, r_clr, QR4 );
177 u_r5:  registro port map ( DB5,  clk, r_wr_en, r_clr, QR5 );
178 u_r6:  registro port map ( DB6,  clk, r_wr_en, r_clr, QR6 );
179 u_r7:  registro port map ( DB7,  clk, r_wr_en, r_clr, QR7 );
180 u_r8:  registro port map ( DB8,  clk, r_wr_en, r_clr, QR8 );
181 u_r9:  registro port map ( DB9,  clk, r_wr_en, r_clr, QR9 );
182 u_r10: registro port map ( DB10, clk, r_wr_en, r_clr, QR10 );
```

```
183 u_r11: registro port map ( DB11, clk, r_wr_en, r_clr, QR11 );
184 u_r12: registro port map ( DB12, clk, r_wr_en, r_clr, QR12 );
185 u_r13: registro port map ( DB13, clk, r_wr_en, r_clr, QR13 );
186 u_r14: registro port map ( DB14, clk, r_wr_en, r_clr, QR14 );
187 u_r15: registro port map ( DB15, clk, r_wr_en, r_clr, QR15 );
188
189 u_mux_r0:  mux2a1_nb port map ( QR0,  data_in, datai_src, Y0 );
190 u_mux_r1:  mux2a1_nb port map ( QR1,  data_in, datai_src, Y1 );
191 u_mux_r2:  mux2a1_nb port map ( QR2,  data_in, datai_src, Y2 );
192 u_mux_r3:  mux2a1_nb port map ( QR3,  data_in, datai_src, Y3 );
193 u_mux_r4:  mux2a1_nb port map ( QR4,  data_in, datai_src, Y4 );
194 u_mux_r5:  mux2a1_nb port map ( QR5,  data_in, datai_src, Y5 );
195 u_mux_r6:  mux2a1_nb port map ( QR6,  data_in, datai_src, Y6 );
196 u_mux_r7:  mux2a1_nb port map ( QR7,  data_in, datai_src, Y7 );
197 u_mux_r8:  mux2a1_nb port map ( QR8,  data_in, datai_src, Y8 );
198 u_mux_r9:  mux2a1_nb port map ( QR9,  data_in, datai_src, Y9 );
199 u_mux_r10: mux2a1_nb port map ( QR10, data_in, datai_src, Y10 );
200 u_mux_r11: mux2a1_nb port map ( QR11, data_in, datai_src, Y11 );
201 u_mux_r12: mux2a1_nb port map ( QR12, data_in, datai_src, Y12 );
202 u_mux_r13: mux2a1_nb port map ( QR13, data_in, datai_src, Y13 );
203 u_mux_r14: mux2a1_nb port map ( QR14, data_in, datai_src, Y14 );
204 u_mux_r15: mux2a1_nb port map ( QR15, data_in, datai_src, Y15 );
205
206 u_ram0:  sp_ram_512x32 port map ( ram_address, clk, Y0,  X0,  ram_wr_en(0) );
207 u_ram1:  sp_ram_512x32 port map ( ram_address, clk, Y1,  X1,  ram_wr_en(1) );
208 u_ram2:  sp_ram_512x32 port map ( ram_address, clk, Y2,  X2,  ram_wr_en(2) );
209 u_ram3:  sp_ram_512x32 port map ( ram_address, clk, Y3,  X3,  ram_wr_en(3) );
210 u_ram4:  sp_ram_512x32 port map ( ram_address, clk, Y4,  X4,  ram_wr_en(4) );
211 u_ram5:  sp_ram_512x32 port map ( ram_address, clk, Y5,  X5,  ram_wr_en(5) );
212 u_ram6:  sp_ram_512x32 port map ( ram_address, clk, Y6,  X6,  ram_wr_en(6) );
213 u_ram7:  sp_ram_512x32 port map ( ram_address, clk, Y7,  X7,  ram_wr_en(7) );
214 u_ram8:  sp_ram_512x32 port map ( ram_address, clk, Y8,  X8,  ram_wr_en(8) );
215 u_ram9:  sp_ram_512x32 port map ( ram_address, clk, Y9,  X9,  ram_wr_en(9) );
216 u_ram10: sp_ram_512x32 port map ( ram_address, clk, Y10, X10, ram_wr_en(10) );
217 u_ram11: sp_ram_512x32 port map ( ram_address, clk, Y11, X11, ram_wr_en(11) );
218 u_ram12: sp_ram_512x32 port map ( ram_address, clk, Y12, X12, ram_wr_en(12) );
219 u_ram13: sp_ram_512x32 port map ( ram_address, clk, Y13, X13, ram_wr_en(13) );
220 u_ram14: sp_ram_512x32 port map ( ram_address, clk, Y14, X14, ram_wr_en(14) );
221 u_ram15: sp_ram_512x32 port map ( ram_address, clk, Y15, X15, ram_wr_en(15) );
222
223 u_ep1:  ep_heat port map ( X0,  X1,  X2, K1, K2, DB1);
224 u_ep2:  ep_heat port map ( X1,  X2,  X3, K1, K2, DB2);
225 u_ep3:  ep_heat port map ( X2,  X3,  X4, K1, K2, DB3);
226 u_ep4:  ep_heat port map ( X3,  X4,  X5, K1, K2, DB4);
227 u_ep5:  ep_heat port map ( X4,  X5,  X6, K1, K2, DB5);
228 u_ep6:  ep_heat port map ( X5,  X6,  X7, K1, K2, DB6);
229 u_ep7:  ep_heat port map ( X6,  X7,  X8, K1, K2, DB7);
230 u_ep8:  ep_heat port map ( X7,  X8,  X9, K1, K2, DB8);
231 u_ep9:  ep_heat port map ( X8,  X9, X10, K1, K2, DB9);
232 u_ep10: ep_heat port map ( X9, X10, X11, K1, K2, DB10);
233 u_ep11: ep_heat port map (X10, X11, X12, K1, K2, DB11);
234 u_ep12: ep_heat port map (X11, X12, X13, K1, K2, DB12);
235 u_ep13: ep_heat port map (X12, X13, X14, K1, K2, DB13);
236 u_ep14: ep_heat port map (X13, X14, X15, K1, K2, DB14);
237
238 u_mux_o: mux16a1_nb port map( X0, X1, X2, X3, X4, X5, X6, X7,
239 X8, X9, X10, X11, X12, X13, X14, X15,
240 datao_src, RESULTADOS);
241
242 u_k:   performance_counter port map ( control(29), estado,
243 clk, control(30), C1, C2 );
```

```
244
245 DB0  <= X0;
246 DB15 <= X15;
247
248 R0  <= X0;
249 R1  <= X1;
250 R2  <= X2;
251 R3  <= X3;
252 R4  <= X4;
253 R5  <= X5;
254 R6  <= X6;
255 R7  <= X7;
256 R8  <= X8;
257 R9  <= X9;
258 R10 <= X10;
259 R11 <= X11;
260 R12 <= X12;
261 R13 <= X13;
262 R14 <= X14;
263 R15 <= X15;
264
265 end estructural;
```

C.3.1. Unidad de control arquitectura paralela 14 EP, 16 RAM, ecuación de calor 1D

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity unidad_control_concurrente_nram_x16 is
6  port(   control:                in  std_logic_vector(31 downto 0);
7          N:                      in  STD_LOGIC_VECTOR(8 downto 0);
8          clk:                in  std_logic;
9          ram_address:            out STD_LOGIC_VECTOR(8 downto 0);
10     s_ram:                      out std_logic_vector(3 downto 0);
11     ram_parallel_wr_en:         out std_logic;
12     ram_general_wr_en:          out std_logic;
13        datai_src:               out std_logic;
14        r_wr_en:                 out std_logic;
15        r_clr:                   out std_logic;
16     datao_src:                  out std_logic_vector(3 downto 0);
17        n_estados:               out STD_LOGIC_VECTOR(3 downto 0);
18        e_control:               out std_logic );
19 end entity unidad_control_concurrente_nram_x16;
20
21 architecture funcional of unidad_control_concurrente_nram_x16 is
22
23 signal kn: unsigned(8 downto 0);
24 signal ka: unsigned(8 downto 0);
25 type estados is (e0, e1, e2, e3, e4, e5, e6, e7);
26 signal estado: estados;
27
28 signal u_N: unsigned(8 downto 0);
29
30 begin
31
```

```
32 u_N <= unsigned(N);
33
34 process(clk, control, estado, u_N, ka, kn)
35 begin
36         if ( control(30) = '1' ) then
37                 ka        <= ( others => '0' );
38                 kn      <= ( others => '0' );
39                 estado <= e0;
40         elsif( rising_edge(clk) ) then
41
42                 case estado is
43
44                         when e0 =>
45                                 ka <= ( others => '0' );
46                                 kn <= ( others => '0' );
47                                 if ( control(31) = '1' ) then
48                                         estado <= e1;
49                                 else
50                                         estado <= e0;
51                                 end if;
52
53                         when e1 =>
54                  if (kn < u_N-1) then
55                     estado <= e2;
56                  else
57                     kn      <= (others => '0');
58                     estado <= e7;
59                  end if;
60
61                         when e2 =>
62                                 estado <= e3;
63
64                     when e3 =>
65                  estado <= e4;
66
67                         when e4 =>
68                                 kn          <= kn + 1;
69                                 ka          <= ka + 1;
70                                 estado <= e5;
71
72             when e5 =>
73                  estado <= e6;
74
75             when e6 =>
76                  estado <= e1;
77
78             when e7 =>
79                                 if ( control(31) = '1' ) then
80                                         estado <= e7;
81                                 else
82                                         estado <= e0;
83                                 end if;
84
85                 end case;
86
87         end if;
88
89 end process;
90
91
92 with estado select ram_parallel_wr_en <= '1' when e5,
```

```
                                        '0' when others;

with estado select ram_address <= control(8 downto 0) when e0,
                                  std_logic_vector(ka(8 downto 0)) when others;


with estado select datai_src <= '0' when e0,
                                '1' when others;

with estado select r_wr_en <=   '1' when e3,
                                                '0' when others;

r_clr <= control(30);

ram_general_wr_en <= control(16);
s_ram             <= control(15 downto 12);
datao_src         <= control(23 downto 20);

with estado select n_estados <= "0000" when e0,
                                                "0001" when e1,
                                                "0010" when e2,
                                                "0011" when e3,
                                                "0100" when e4,
                                                "0101" when e5,
                                "0100" when e6,
                                "0101" when e7;

with estado select e_control <=  '0' when e0,
                                       '1' when others;

end architecture;
```

C.3.2. Codigo C para el host

```
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xsdps.h"
#include "xil_printf.h"
#include "ff.h"
#include "xil_cache.h"

static FATFS FS_instance; // File System instance
static FIL file1;   // File instance
FRESULT result;     // FRESULT variable
static char FileName[32] = "SOLR16.txt"; // name of the log
static char *Log_File; // pointer to the log
char *Path = "0:/";  //  string pointer to the logical drive number
unsigned int BytesWr, BytesRd;
u8 Buffer_logger[32] __attribute__ ((aligned(32))); // Buffer should be word aligned (
    multiple of 4)
u32 Buffer_size = 32, line_size;

Xuint32 *baseaddr_p = (Xuint32 *)XPAR_HEQ_CRX16_RAMX16_0_S00_AXI_BASEADDR;

int main (void)
```

```
24 {
25   float x=0, dx=0, ic;
26   float k1=0.25, k2;
27   float l = 16;
28   unsigned int N = 16, L = 16;
29   unsigned int k, kn, kj;
30   unsigned int C1m, C1l, C2;
31   unsigned int estado = 0x00000001;
32   unsigned int n_estado;
33
34   printf("Heat Equation Test\n");
35
36   init_platform();
37
38   *(baseaddr_p+1) = 0x40000000;
39   *(baseaddr_p+1) = 0x00000000;
40
41   k2 = 1-2*k1;
42   dx = 1/(l-1);
43
44   // Mount SD Card and initialize device
45   result = f_mount(&FS_instance,Path, 0);
46
47   // Open log for writing
48   Log_File = (char *)FileName;
49   result = f_open(&file1, Log_File, FA_WRITE | FA_CREATE_ALWAYS);
50
51   *(baseaddr_p+2) = *((unsigned int*)&k1);
52   *(baseaddr_p+3) = *((unsigned int*)&k2);
53   *(baseaddr_p+4) = N;
54
55   *(baseaddr_p+0) = 0x00000000;
56
57   for ( kn = 0; kn < N; kn++ )
58   {
59     *(baseaddr_p+1) = 0x00010000 + kn;
60     *(baseaddr_p+1) = 0x0001F000 + kn;
61   }
62
63   for ( kj = 0; kj < L; kj++ )
64   {
65     if ( x < 0.5 )
66       ic = 2*x;
67     else
68       ic = 2*(1-x);
69
70     *(baseaddr_p+1) = 0x00010000 + 0x00001000 * kj;
71     *(baseaddr_p+0) = *((unsigned int*)&ic);
72
73     x = x + dx;
74   }
75
76   *(baseaddr_p+1) = 0x80000000;
77
78   while( estado == 0x00000001 )
79   {
80     *(baseaddr_p+1) = 0x00000000;
81     estado = *(baseaddr_p+7) && 0x0000000F;
82   }
83
84   for ( kn = 0; kn < N; kn++ )
```

```
85    {
86      for ( kj = 0; kj < L; kj++ )
87      {
88        *(baseaddr_p+1) = 0x00000000 + kn;
89        if ( kj < L-1 )
90        {
91          sprintf(Buffer_logger, "%1.15f                ", *((float*)&(*(baseaddr_p+11+kj))) );
92          result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
93        }
94        else
95        {
96          sprintf(Buffer_logger, "%1.15f              \n", *((float*)&(*(baseaddr_p+11+kj))) )
      ;
97          result = f_write(&file1, (const void*)Buffer_logger, Buffer_size, &BytesWr);
98        }
99      }
100   }
101
102   for ( kn = 0; kn < N; kn++ )
103   {
104     *(baseaddr_p+1) = 0x00000000 + kn;
105
106     for ( kj = 0; kj < L; kj++ )
107     {
108       printf("%1.5f ", *((float*)&(*(baseaddr_p+11+kj))) );
109     }
110     printf("\n");
111   }
112
113   //Close file.
114   result = f_close(&file1);
115
116   result = f_mount(NULL, Path, 0);
117
118   *(baseaddr_p+1) = 0x00000000;
119
120   printf("Data written to log Successfully\r\n");
121
122   C1m = *(baseaddr_p+8);
123   C1l = *(baseaddr_p+9);
124   C2  = *(baseaddr_p+10);
125
126   printf("Medidas de rendimiento\n");
127   printf("Numero de ciclos total: %8X%8X\n",C1m,C1l);
128   printf("MSB: %8X\n",C1m);
129   printf("LSB: %8X\n",C1l);
130   printf("Tiempo total: %u ns\n",10*C1l);
131   printf("Numero de ciclos de proceso: %d (%d ns)\n",C2,10*C2);
132
133   printf("Heat Equation Test Finished\n\r");
134   return 0;
135 }
```

C.4. Ecuación de Laplace

C.4.1. Codigo C implementación base 1 iteración

```
void laplace2D(float *U_0, int *a, float *V)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=U_0
#pragma HLS INTERFACE axis port=a
#pragma HLS INTERFACE axis port=V

  int i, j, k;
  int X, Y, N, c;
  static float U0[65536];
  float c1, c2, cf;

  X = *a++;
  Y = *a++;
  N = *a++;
  c = *a++;
  cf = *((float *)&c);

  for (k = 0; k < 65536; k++){
    U0[k] = *U_0++;
  }

  for (j = 1; j < 255; j++){
    for (i = 1; i < 255; i++){
      c1 = U0[j*X+i] + U0[(j+1)*X+i];
      c2 = U0[j*X+i+1] + U0[j*X+i-1];
      V[j*X+i] = cf*(c1 + c2);
    }
  }
}
```

C.4.2. Codigo C implementación base N iteraciones arreglo 2D

```
void laplace2D_full(float *U_0, int *a, float *V)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=U_0
#pragma HLS INTERFACE axis port=a
#pragma HLS INTERFACE axis port=V

  int i, j, k;
  int X, Y, N, c;
  static float U0[128][128], U_XY[128][128];
  float c1, c2, cf;

  X = *a++;
  Y = *a++;
  N = *a++;
  c = *a++;
  cf = *((float *)&c);

  for (j = 0; j < Y; j++){
    for (i = 0; i < X; i++){
      U0[j][i] = *U_0++;
    }
  }

```

```
  for (k = 0; k < N; k++){
    for (j = 1; j < Y-1; j++){
      for (i = 1; i < X-1; i++){
        c1 = U0[j+1][i] + U0[j-1][i];
        c2 = U0[j][i+1] + U0[j][i-1];
        U_XY[j][i] = cf*(c1 + c2);
      }
    }

    for (j = 1; j < Y-1; j++){
      for (i = 1; i < X-1; i++){
        U0[j][i] = U_XY[j][i];
        if ( k == N-1){
          V[j*X+i] = U_XY[j][i];
        }
      }
    }
  }
}
```

C.4.3. Codigo C implementación base N iteraciones vector

```
void laplace2D(float *U0, float a, float *V)
{
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE axis register both port=V
#pragma HLS INTERFACE port=a
#pragma HLS INTERFACE axis register both port=U0

  int i, j, k;
  static float c1, c2;
  static float U_0[65536], U_XY[65536];

  int X = 256, Y = 256, N = 100;

  for (k = 0; k < 65536; k++){
    U_0[k] = *U0++;
  }

  for (k = 0; k < N; k++)
  {
    for (j = 1; j < Y-1; j++)
      for ( i = 1; i < X-1; i++ )
      {
        c1 = U_0[(j+1)*X+i] + U_0[(j-1)*X+i];
        c2 = U_0[j*X+i+1] + U_0[j*X+i-1];
        U_XY[j*X+i] = a * (c1 + c2);
      }

    for (j = 1; j < Y-1; j++)
      for ( i = 1; i < X-1; i++ ){
        U_0[j*X+i] = U_XY[j*X+i];
      }
  }

  for (j = 1; j < Y-1; j++)
    for ( i = 1; i < X-1; i++ ){
      *V++ = U_0[j*X+i];
```

```
37     }
38 }
```

C.4.4. Codigo C implementación paralelización

```
1  #include <hls_stream.h>
2  #include <ap_axi_sdata.h>
3  #include "axi_stream_dma_example.h"
4
5  #define PY  4       // 4
6  #define PYI 256/PY      // 256/4 = 64
7  #define NYI PY*(PYI-1)    // 4*(64-1) = 252
8  #define PYE (254-NYI)   // 254 - 252 = 2
9  #define PYF PYE/2     // 2/2 = 1
10 #define PYL 254-PYF     // 254-1 = 253
11
12 void core(hls::stream<FloatAxis> &inStream, hls::stream<FloatAxis> &outStream, float gain)
13 {
14 #pragma HLS INTERFACE axis register both port=outStream
15 #pragma HLS INTERFACE axis register both port=inStream
16 #pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
17 #pragma HLS INTERFACE s_axilite port=gain bundle=CRTL_BUS
18
19 int i, j, k, n;
20 static float U_0[65536], U_XY[65536];
21 #pragma HLS ARRAY_PARTITION variable=U_0 block factor=64 dim=1
22 #pragma HLS ARRAY_PARTITION variable=U_XY block factor=64 dim=1
23
24 int X = 256, Y = 256;
25 int N = 1000000;
26 float f = gain;
27
28 FloatAxis valIn;
29 FloatAxis valOut;
30
31   for (j = 0; j < Y; j++)
32     for (i = 0; i < X; i++) {
33     #pragma HLS PIPELINE
34       valIn = inStream.read();
35       U_0[j * X + i] = valIn.data;
36       U_XY[j * X + i] = valIn.data;
37     }
38
39   for (n = 0; n < N / 2; n++) {
40
41     for (j = 1; j <= PYF; j++)
42       for (i = 1; i < X - 1; i++) {
43       #pragma HLS PIPELINE
44         U_XY[j * X + i] = f
45             * (U_0[(j + 1) * X + i] + U_0[(j - 1) * X + i]
46                 + U_0[j * X + i + 1] + U_0[j * X + i - 1]);
47       }
48
49     for (j = 0; j < PY; j++) {
50       for (i = 1; i < X - 1; i++) {
51       #pragma HLS PIPELINE
52         for (k = 0; k < PYI - 1; k++) {
53           U_XY[(j + PYF + PY * k) * X + i] = f
```

```
54                * (U_0[(j + PYF + PY * k + 1) * X + i]
55                    + U_0[(j + PYF + PY * k - 1) * X + i]
56                    + U_0[(j + PYF + PY * k) * X + i + 1]
57                    + U_0[(j + PYF + PY * k) * X + i - 1]);
58        }
59      }
60    }
61
62    for (j = 0; j <= PYF; j++)
63      for (i = 1; i < X - 1; i++) {
64      #pragma HLS PIPELINE
65        U_XY[(j + PYL) * X + i] = f
66            * (U_0[(j + PYL + 1) * X + i]
67                + U_0[(j + PYL - 1) * X + i]
68                + U_0[(j + PYL) * X + i + 1]
69                + U_0[(j + PYL) * X + i - 1]);
70      }
71
72    for (j = 1; j <= PYF; j++)
73      for (i = 1; i < X - 1; i++) {
74      #pragma HLS PIPELINE
75        U_0[j * X + i] = f
76            * (U_XY[(j + 1) * X + i] + U_XY[(j - 1) * X + i]
77                + U_XY[j * X + i + 1] + U_XY[j * X + i - 1]);
78      }
79
80    for (j = 0; j < PY; j++) {
81      for (i = 1; i < X - 1; i++) {
82      #pragma HLS PIPELINE
83        for (k = 0; k < PYI - 1; k++) {
84          U_0[(j + PYF + PY * k) * X + i] = f
85              * (U_XY[(j + PYF + PY * k + 1) * X + i]
86                  + U_XY[(j + PYF + PY * k - 1) * X + i]
87                  + U_XY[(j + PYF + PY * k) * X + i + 1]
88                  + U_XY[(j + PYF + PY * k) * X + i - 1]);
89        }
90      }
91    }
92    for (j = 0; j <= PYF; j++)
93      for (i = 1; i < X - 1; i++) {
94      #pragma HLS PIPELINE
95        U_0[(j + PYL) * X + i] = f
96            * (U_XY[(j + PYL + 1) * X + i]
97                + U_XY[(j + PYL - 1) * X + i]
98                + U_XY[(j + PYL) * X + i + 1]
99                + U_XY[(j + PYL) * X + i - 1]);
100     }
101 }
102
103 for (j = 1; j < Y - 1; j++)
104   for (i = 1; i < X - 1; i++) {
105     #pragma HLS PIPELINE
106     valOut.data = U_0[j * X + i];
107     valOut.keep = valIn.keep;
108     valOut.strb = valIn.strb;
109     valOut.user = valIn.user;
110     valOut.last = (j * i == ((X - 2) * (Y - 2))) ? 1 : 0;
111     valOut.id = valIn.id;
112     valOut.dest = valIn.dest;
113     outStream.write(valOut);
114   }
```

```
115 }
```

C.4.5. Codigo C para el host

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #include <errno.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9  #include <termios.h>
10 #include <sys/mman.h>
11 #include <sys/time.h> //gettimeofday()
12 #include "xcore.h"
13 // DMA ADRESSES MM2S
14 #define MM2S_CONTROL_REGISTER 0x00
15 #define MM2S_STATUS_REGISTER 0x04
16 #define MM2S_START_ADDRESS 0x18
17 #define MM2S_LENGTH 0x28
18 // DMA ADRESSES S2MM
19 #define S2MM_CONTROL_REGISTER 0x30
20 #define S2MM_STATUS_REGISTER 0x34
21 #define S2MM_DESTINATION_ADDRESS 0x48
22 #define S2MM_LENGTH 0x58
23 //Physical addresses
24 #define DMA_BASE_ADRESSES 0x40400000
25 #define INPUT_BASE_ADDR   0x05000000
26 #define OUTPUT_BASE_ADDR  0x07000000
27 #define XCORE_ADDRESSES   0x43C00000
28 //Size
29 #define SIZE_X 256
30 #define SIZE_Y 256
31 #define N_TEST0 1
32 XCore Sbl;
33
34 void  dma_set(unsigned int* dma_virtual_address, int offset, unsigned int value) {
35     dma_virtual_address[offset>>2] = value;
36 }
37
38 unsigned int dma_get(unsigned int* dma_virtual_address, int offset) {
39     return dma_virtual_address[offset>>2];
40 }
41
42 void dma_s2mm_status(unsigned int* dma_virtual_address) {
43     unsigned int status = dma_get(dma_virtual_address, S2MM_STATUS_REGISTER);
44     printf("Stream to memory-mapped status (0x%08x@0x%02x):", status, S2MM_STATUS_REGISTER);
45     if (status & 0x00000001) printf(" halted"); else printf(" running");
46     if (status & 0x00000002) printf(" idle");
47     if (status & 0x00000008) printf(" SGIncld");
48     if (status & 0x00000010) printf(" DMAIntErr");
49     if (status & 0x00000020) printf(" DMASlvErr");
50     if (status & 0x00000040) printf(" DMADecErr");
51     if (status & 0x00000100) printf(" SGIntErr");
52     if (status & 0x00000200) printf(" SGSlvErr");
53     if (status & 0x00000400) printf(" SGDecErr");
54     if (status & 0x00001000) printf(" IOC_Irq");
```

```
55      if (status & 0x00002000) printf(" Dly_Irq");
56      if (status & 0x00004000) printf(" Err_Irq");
57      printf("\n");
58  }
59
60  void dma_mm2s_status(unsigned int* dma_virtual_address) {
61      unsigned int status = dma_get(dma_virtual_address, MM2S_STATUS_REGISTER);
62      printf("Memory-mapped to stream status (0x%08x@0x%02x):", status, MM2S_STATUS_REGISTER);
63      if (status & 0x00000001) printf(" halted"); else printf(" running");
64      if (status & 0x00000002) printf(" idle");
65      if (status & 0x00000008) printf(" SGIncld");
66      if (status & 0x00000010) printf(" DMAIntErr");
67      if (status & 0x00000020) printf(" DMASlvErr");
68      if (status & 0x00000040) printf(" DMADecErr");
69      if (status & 0x00000100) printf(" SGIntErr");
70      if (status & 0x00000200) printf(" SGSlvErr");
71      if (status & 0x00000400) printf(" SGDecErr");
72      if (status & 0x00001000) printf(" IOC_Irq");
73      if (status & 0x00002000) printf(" Dly_Irq");
74      if (status & 0x00004000) printf(" Err_Irq");
75      printf("\n");
76  }
77
78  void  dma_mm2s_sync(unsigned int* dma_virtual_address) {
79      unsigned int mm2s_status =  dma_get(dma_virtual_address, MM2S_STATUS_REGISTER);
80      while(!(mm2s_status & 1<<12) || !(mm2s_status & 1<<1) ){
81          mm2s_status =  dma_get(dma_virtual_address, MM2S_STATUS_REGISTER);
82      }
83  }
84
85  void  dma_s2mm_sync(unsigned int* dma_virtual_address) {
86      unsigned int s2mm_status = dma_get(dma_virtual_address, S2MM_STATUS_REGISTER);
87      while(!(s2mm_status & 1<<12) || !(s2mm_status & 1<<1)){
88          s2mm_status = dma_get(dma_virtual_address, S2MM_STATUS_REGISTER);
89      }
90  }
91
92  void memdump(void* virtual_address, int byte_count) {
93      char *p = virtual_address;
94      int offset;
95      for (offset = 0; offset < byte_count; offset++) {
96          printf("%02x", p[offset]);
97          if (offset % 4 == 3) { printf(" "); }
98      }
99      printf("\n");
100 }
101
102 void print_accel_status()
103 {
104   int isDone, isIdle, isReady;
105   isDone = XCore_IsDone(&Sbl);
106   isIdle = XCore_IsIdle(&Sbl);
107   isReady = XCore_IsReady(&Sbl);
108   printf("Vector Adder Status: isDone %d, isIdle %d, isReady%d\r\n", isDone, isIdle, isReady
        );
109 }
110
111 void XCore_init(unsigned long inputAddress,unsigned long outputAddress){
112     int status = XCore_Initialize(&Sbl,"core");
113     if(status != XST_SUCCESS){
114         printf("XCore_Initialize failed\n");
```

```
115     }
116 }
117
118 void Core_hw(int N0){
119   float  gain=0.25;
120   int    N = N0;
121   unsigned int gainu = *((uint32_t *) &gain);
122   XCore_Set_N0(&Sbl,N);
123   XCore_Set_gain(&Sbl,gainu);
124   XCore_Start(&Sbl);
125 }
126
127 unsigned int * assignToPhysical(unsigned long address,unsigned int size){
128   int devmem = open("/dev/mem", O_RDWR | O_SYNC);
129   off_t PageOffset = (off_t) address % getpagesize();
130   off_t PageAddress = (off_t) (address - PageOffset);
131   return (unsigned int *) mmap(NULL, size*sizeof(unsigned int), PROT_READ|PROT_WRITE,
      MAP_SHARED, devmem, PageAddress);
132 }
133
134 int main(int argc, char* argv[])
135 {
136   int N;
137   if(argc!=2) {
138     printf("usage ./laplace2d #N0. --version 1\n");
139     exit(1);
140   }
141     N = strtol(argv[1], NULL, 10);
142   XCore_init(INPUT_BASE_ADDR,OUTPUT_BASE_ADDR);
143
144   unsigned int *virtual_address, *input_hw, *output_hw;//,*virtual_address0;
145
146   struct timeval  start,end;
147
148   double time_prom[N_TEST0];
149   int h = 0;
150   for(h=0;h<N_TEST0;h++){
151     time_prom[h]=0;
152   }
153
154   virtual_address = assignToPhysical(DMA_BASE_ADRESSES,16384);
155
156   input_hw    = assignToPhysical(INPUT_BASE_ADDR,SIZE_X*SIZE_Y);
157
158   output_hw  =  assignToPhysical(OUTPUT_BASE_ADDR,(SIZE_X-2)*(SIZE_Y-2));
159
160   memset(output_hw, 0, (SIZE_X-2)*(SIZE_Y-2)*sizeof(float)); // Clear destination block
161
162   memset(input_hw, 0, SIZE_X*SIZE_Y*sizeof(float)); // Clear source block
163
164   float temp = 1.0;
165   int k;
166
167   for (k = 0; k < SIZE_X * SIZE_Y; k++)
168     input_hw[k] = 0;
169
170   for (k = 0; k < SIZE_X; k++) {
171     input_hw[k] = *((uint32_t *) &temp);
172     input_hw[k + SIZE_X * SIZE_Y - SIZE_X] = *((uint32_t *) &temp);
173     input_hw[k * SIZE_X] = *((uint32_t *) &temp);
174     input_hw[k * SIZE_X + SIZE_X - 1] = *((uint32_t *) &temp);
```

```
175   }
176
177   for(k=0;k< N_TEST0;k++){
178     gettimeofday(&start, NULL);
179     Core_hw(N);
180     dma_set(virtual_address, S2MM_CONTROL_REGISTER, 4);
181     dma_set(virtual_address, MM2S_CONTROL_REGISTER, 4);
182     dma_set(virtual_address, S2MM_CONTROL_REGISTER, 0);
183     dma_set(virtual_address, MM2S_CONTROL_REGISTER, 0);
184     dma_set(virtual_address, S2MM_DESTINATION_ADDRESS, OUTPUT_BASE_ADDR);
185     dma_set(virtual_address, MM2S_START_ADDRESS, INPUT_BASE_ADDR);
186     dma_set(virtual_address, S2MM_CONTROL_REGISTER, 0xf001);
187     dma_set(virtual_address, MM2S_CONTROL_REGISTER, 0xf001);
188     dma_set(virtual_address, S2MM_LENGTH,(SIZE_X-2)*(SIZE_Y-2)*sizeof(float));
189     dma_set(virtual_address, MM2S_LENGTH,SIZE_X*SIZE_Y*sizeof(float));
190     while(!XCore_IsDone(&Sbl));
191     dma_mm2s_sync(virtual_address);
192     dma_s2mm_sync(virtual_address);
193     gettimeofday(&end, NULL);
194     time_prom[k] =(double)1000000*(end.tv_sec-start.tv_sec)+(end.tv_usec-start.tv_usec);
195   }
196    char buffer_0 [50];
197    char buffer_1 [50];
198    sprintf (buffer_0,"Datos/Time_iteracion%d.txt", N);
199    sprintf (buffer_1,"Datos/Malla_iteracion%d.txt", N);
200    FILE *ofp_0, *ofp_1;
201    ofp_0 = fopen(buffer_0,"w");
202    ofp_1 = fopen(buffer_1,"w");
203    for(k=0;k<N_TEST0;k++)
204       fprintf(ofp_0,"%f\n",time_prom[k]);
205    k=0;
206    int j,i;
207    for (j = 1; j < SIZE_Y-1; j++) {
208     for (i = 1; i < SIZE_X-1; i++) {
209       fprintf(ofp_1,"%2.15f ", *((float *) &output_hw[k]));
210       k++;
211     }
212     fprintf(ofp_1,"\n");
213   }
214    fclose(ofp_0);
215    fclose(ofp_1);
216    printf("Completed interaction %d\n",N);
217
218    XCore_Release(&Sbl);
219    return 0;
220 }
```

Bibliografía

[1] Bandishti, V. ; Pananilath, I. ; Bondhugula, U.: Tiling Stencil Computations to Maximize Parallelism. En: *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* IEEE, 2012, p. 1–11

[2] Beauchamp, Michael J. ; Hauck, Scott ; Underwood, Keith D. ; Hemmert, K S.: Architectural modifications to enhance the floating-point performance of FPGAs. En: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16 (2008), Nr. 2, p. 177–187

[3] Belanović, Pavle ; Leeser, Miriam: A library of parameterized floating-point modules and their use. En: *International Conference on Field Programmable Logic and Applications* Springer, 2002, p. 657–666

[4] Brodtkorb, A. R. ; Dyken, C. ; Hagen, T. R. ; Hjelmervik, J. M. ; Storaasli, O. O.: State-of-the-art in heterogeneous computing. En: *Scientific Programming, IOS Press Amsterdam* 18 (2010), p. 1–33

[5] Caffarena, Gabriel ; López, Juan A. ; Leyva, Gerardo ; Carreras, Carlos ; Nieto-Taladriz, Octavio: Architectural synthesis of fixed-point DSP datapaths using fpgas. En: *International Journal of Reconfigurable Computing* 2009 (2009), p. 8

[6] Cattaneo, Riccardo ; Natale, Giuseppe ; Sicignano, Carlo ; Sciuto, Donatella ; Santambrogio, Marco D.: On how to accelerate iterative stencil loops: a scalable streaming-based approach. En: *ACM Transactions on Architecture and Code Optimization (TACO)* 12 (2016), Nr. 4, p. 53

[7] Cecilia, J. M. ; Abellán, J. L. ; Fernández, J. ; Acacio, M. E. ; García, J. M. ; Ujaldón, M.: Stencil computations on heterogeneous platforms for the Jacobi method: GPUs versus Cell BE. En: *The Journal of Supercomputing, Springer Science+Business Media* 62 (2012), Nr. 2, p. 787–803

[8] Chong, Yee J. ; Parameswaran, Sri: Configurable multimode embedded floating-point units for FPGAs. En: *IEEE transactions on very large scale integration (VLSI) systems* 19 (2011), Nr. 11, p. 2033–2044

[9] Chugh, Nitin ; Vasista, Vinay ; Purini, Suresh ; Bondhugula, Uday: A DSL compiler for accelerating image processing pipelines on FPGAs. En: *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on* IEEE, 2016, p. 327–338

[10] Cong, Jason ; Li, Peng ; Xiao, Bingjun ; Zhang, Peng: An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers. En: *Proceedings of the 51st annual design automation conference* ACM, 2014, p. 1–6

[11] Datta, Kaushik ; Kamil, Shoaib ; Williams, Samuel ; Oliker, Leonid ; Shalf, John ; Yelick, Katherine: Optimization and performance modeling of stencil computations on modern microprocessors. En: *SIAM review* 51 (2009), Nr. 1, p. 129–159

[12] Deest, Gaël ; Estibals, Nicolas ; Yuki, Tomofumi ; Derrien, Steven ; Rajopadhye, Sanjay: Towards Scalable and Efficient FPGA Stencil Accelerators. En: *6th International Workshop on Polyhedral Compilation Techniques - IMPACT'16*, 2016

[13] Deest, Gaël ; Yuki, Tomofumi ; Rajopadhye, Sanjay ; Derrien, Steven: One size does not fit all: Implementation trade-offs for iterative stencil computations on FPGAs. En: *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on* IEEE, 2017, p. 1–8

[14] Deschamps, Jean-Pierre ; Bioul, Gery J. ; Sutter, Gustavo D.: *Synthesis of arithmetic circuits: FPGA, ASIC and embedded systems*. John Wiley & Sons, 2006

[15] Detrey, Jérémie ; de Dinechin, Florent: Parameterized floating-point logarithm and exponential functions for FPGAs. En: *Microprocessors and Microsystems* 31 (2007), Nr. 8, p. 537–545

[16] Dido, Jérôme ; Geraudie, Nicolas ; Loiseau, Ludovic ; Payeur, Olivier ; Savaria, Yvon ; Poirier, Daniel: A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. En: *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays* ACM, 2002, p. 50–55

[17] de Dinechin, Florent ; Detrey, Jérémie ; Creţ, Octavian ; Tudoran, Radu: When

FPGAs are better at floating-point than microprocessors. (2007)

[18] Dursun, Hikmet ; Nomura, Ken-Ichi ; Peng, Liu ; Seymour, Richard ; Wang, Weiqiang ; Kalia, Rajiv K. ; Nakano, Aiichiro ; Vashishta, Priya: A multilevel parallelization framework for high-order stencil computations. En: *European Conference on Parallel Processing* Springer, 2009, p. 642–653

[19] Echeverría, Pedro ; López-Vallejo, Marisa: Customizing floating-point units for FPGAs: Area-performance-standard trade-offs. En: *Microprocessors and Microsystems* 35 (2011), Nr. 6, p. 535–546

[20] Escobedo, Juan ; Lin, Mingjie: Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. En: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* ACM, 2018, p. 199–208

[21] de Fine Licht, Johannes ; Blott, Michaela ; Hoefler, Torsten: Designing scalable FPGA architectures using high-level synthesis. En: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)* Vol. 53 ACM, 2018, p. 403–404

[22] Fu, Haohuan ; Osborne, William ; Clapp, Robert G. ; Mencer, Oskar ; Luk, Wayne: Accelerating seismic computations using customized number representations on FPGAs. En: *EURASIP Journal on Embedded Systems* 2009 (2009), p. 3

[23] Ho, Chun H. ; Leong, Monk-Ping ; Leong, Philip Heng W. ; Becker, Jürgen ; Glesner, Manfred: Rapid prototyping of FPGA based floating point DSP systems. En: *Rapid System Prototyping, 2002. Proceedings. 13th IEEE International Workshop on* IEEE, 2002, p. 19–24

[24] Ho, Chun H. ; Yu, Chi W. ; Leong, Philip ; Luk, Wayne ; Wilton, Steven J.: Floating-point FPGA: architecture and modeling. En: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17 (2009), Nr. 12, p. 1709–1718

[25] Hockert, Neil ; Compton, Katherine: Improving floating-point performance in less area: Fractured floating point units (FFPUs). En: *Journal of Signal Processing Systems* 67 (2012), Nr. 1, p. 31–46

[26] Kobayashi, R. ; Takamaeda-Yamazaki, S. ; Kise, K.: Towards a Low-Power Accelerator of Many FPGAs for Stencil Computations. En: *Proceedings of the IEEE Third International Conference on Networking and Computing* IEEE, 2012, p. 343–349

[27] Kobayashi, Ryohei ; Oobata, Yuma ; Fujita, Norihisa ; Yamaguchi, Yoshiki ; Boku, Taisuke: OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing. En: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* ACM, 2018, p. 192–201

[28] László, Endre ; Nagy, Zoltán ; Giles, Michael B. ; Reguly, István ; Appleyard, Jeremy ; Szolgay, Peter: Analysis of parallel processor architectures for the solution of the Black-Scholes PDE. En: *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on* IEEE, 2015, p. 1977–1980

[29] Liu, Junyi ; Bayliss, Samuel ; Constantinides, George A.: Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. En: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* IEEE, 2015, p. 159–162

[30] Liu, Junyi ; Wickerson, John ; Bayliss, Samuel ; Constantinides, George A.: Polyhedral-based Dynamic Loop Pipelining for High-Level Synthesis. En: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017)

[31] Liu, Junyi ; Wickerson, John ; Constantinides, George A.: Loop splitting for efficient pipelining in high-level synthesis. En: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* IEEE, 2016, p. 72–79

[32] Mokhov, Andrey ; De Gennaro, Alessandro ; Tarawneh, Ghaith ; Wray, Jonny ; Lukyanov, Georgy ; Mileiko, Sergey ; Scott, Joe ; Yakovlev, Alex ; Brown, Andrew: Language and hardware acceleration backend for graph processing. En: *Specification and Design Languages (FDL), 2017 Forum on* IEEE, 2017, p. 1–7

[33] Mondigo, Antoniette ; Ueno, Tomohiro ; Tanaka, Daichi ; Sano, Kentaro ; Yamamoto, Satoru: Design and scalability analysis of bandwidth-compressed stream computing with multiple FPGAs. En: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2017 12th International Symposium on* IEEE, 2017, p. 1–8

[34] Muranushi, Takayuki ; Makino, Junichiro: Optimal temporal blocking for stencil computation. En: *Procedia Computer Science* 51 (2015), p. 1303–1312

[35] Nacci, Alessandro A. ; Rana, Vincenzo ; Bruschi, Francesco ; Sciuto, Donatella ; Beretta, Ivan ; Atienza, David: A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices. En: *Proceedings of the 50th annual*

design automation conference ACM, 2013, p. 52

[36] Natale, Giuseppe ; Stramondo, Giulio ; Bressana, Pietro ; Cattaneo, Riccardo ; Sciuto, Donatella ; Santambrogio, Marco D.: A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops. En: *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on* IEEE, 2016, p. 1–8

[37] de Oliveira, Cristiano B. ; Cardoso, Joao M. ; Marques, Eduardo: High-level synthesis from C vs. a DSL-based approach. En: *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International* IEEE, 2014, p. 257–262

[38] Peng, Liu ; Seymour, Richard ; Nomura, Ken-ichi ; Kalia, Rajiv K. ; Nakano, Aiichiro ; Vashishta, Priya ; Loddoch, Alexander ; Netzband, Michael ; Volz, William R. ; Wong, Chap C.: High-order stencil computations on multicore clusters. En: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* IEEE, 2009, p. 1–11

[39] Reagen, Brandon ; Adolf, Robert ; Shao, Yakun S. ; Wei, Gu-Yeon ; Brooks, David: Machsuite: Benchmarks for accelerator design and customized architectures. En: *Workload Characterization (IISWC), 2014 IEEE International Symposium on* IEEE, 2014, p. 110–119

[40] Reiche, Oliver ; Özkan, M A. ; Hannig, Frank ; Teich, Jürgen ; Schmid, Moritz: Loop parallelization techniques for fpga accelerator synthesis. En: *Journal of Signal Processing Systems* 90 (2018), Nr. 1, p. 3–27

[41] Rocher, Romuald ; Menard, Daniel ; Herve, Nicolas ; Sentieys, Olivier: Fixed-point configurable hardware components. En: *EURASIP Journal on Embedded Systems* 2006 (2006), Nr. 1, p. 023197

[42] Sakai, Ryotaro ; Sugimoto, Naru ; Miyajima, Takaaki ; Fujita, Naoyuki ; Amano, Hideharu: Acceleration of full-pic simulation on a cpu-fpga tightly coupled environment. En: *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium on* IEEE, 2016, p. 8–14

[43] Sano, K. ; Hatsuda, Y. ; Yamamoto, S.: Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory-Bandwidth. En: *IEEE Transactions on Parallel and Distributed Systems* 25 (2014), March, Nr. 3, p. 695–705

[44] Sano, K. ; Luzhou, W. ; Hatsuda, Y. ; Yamamoto, S.: Scalable FPGA-Array for High-Performance and Power-Efficient Computation Based on Difference Schemes. En: *Proceedings of the Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications* IEEE, 2008, p. 1–9

[45] Sano, Kentaro: FPGA-based systolic computational-memory array for scalable stencil computations. En: *High-Performance Computing Using FPGAs*. Springer, 2013, p. 279–303

[46] Schmid, Moritz ; Reiche, Oliver ; Schmitt, Christian ; Hannig, Frank ; Teich, Jürgen: Code generation for high-level synthesis of multiresolution applications on fpgas. En: *arXiv preprint arXiv:1408.4721* (2014)

[47] Schmitt, Christian ; Schmid, Moritz ; Kuckuk, Sebastian ; Köstler, Harald ; Teich, Jürgen ; Hannig, Frank: Reconfigurable Hardware Generation of Multigrid Solvers with Conjugate Gradient Coarse-Grid Solution. En: *Parallel Processing Letters* 28 (2018), Nr. 04, p. 1850016

[48] Shao, Yakun S. ; Reagen, Brandon ; Wei, Gu-Yeon ; Brooks, David: Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. En: *ACM SIGARCH Computer Architecture News* Vol. 42 IEEE Press, 2014, p. 97–108

[49] Shen, Chongfei ; Liu, Hongtao ; Xie, XB ; Luk, Keith D. ; Hu, Yong: Selection of floating-point or fixed-point for adaptive noise canceller in somatosensory evoked potential measurement. En: *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE* IEEE, 2007, p. 3274–3277

[50] del Sozzo, Emanuele ; Baghdadi, Riyadh ; Amarasinghe, Saman ; Santambrogio, Marco D.: A Common Backend for Hardware Acceleration on FPGA. En: *Computer Design (ICCD), 2017 IEEE International Conference on* IEEE, 2017, p. 427–430

[51] Strenski, Dave ; Simkins, Jim ; Walke, Richard ; Wittig, Ralph: Evaluating fpgas for floating-point performance. En: *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on* IEEE, 2008, p. 1–6

[52] Strzodka, R. ; Shaheen, M. ; Pajak, D. ; Seidel, H.: Cache oblivious parallelograms in iterative stencil computations. En: *Proceedings of the 24th ACM International Conference on Supercomputing* ACM, 2010, p. 49–59

[53] Strzodka, R. ; Shaheen, M. ; Pajak, D. ; Seidel, H.: Cache Accurate Time Skewing in Iterative Stencil Computations. En: *Proceedings of the IEEE International Conference on Parallel Processing* IEEE, 2011, p. 571–581

[54] Tang, Yuan ; Chowdhury, Rezaul A. ; Kuszmaul, Bradley C. ; Luk, Chi-Keung ; Leiserson, Charles E.: The pochoir stencil compiler. En: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* ACM, 2011, p. 117–128

[55] Te Ewe, Chun ; Cheung, Peter Y. ; Constantinides, George A.: Dual fixed-point: An efficient alternative to floating-point computation. En: *International Conference on Field Programmable Logic and Applications* Springer, 2004, p. 200–208

[56] Usui, T. ; Kobayashi, R. ; Kise, K.: A Challenge of Portable and High-Speed FPGA Accelerator. En: *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing, ARC 2015*, 2015, p. 383–392

[57] Vera, G A. ; Pattichis, Marios ; Lyke, James: A dynamic dual fixed-point arithmetic architecture for FPGAs. En: *International Journal of Reconfigurable Computing* 2011 (2011)

[58] Waidyasooriya, Hasitha M. ; Endo, Tsukasa ; Hariyama, Masanori ; Ohtera, Yasuo: OpenCL-Based FPGA Accelerator for 3D FDTD with Periodic and Absorbing Boundary Conditions. En: *International Journal of Reconfigurable Computing* 2017 (2017)

[59] Waidyasooriya, Hasitha M. ; Takei, Yasuhiro ; Tatsumi, Shunsuke ; Hariyama, Masanori: OpenCL-based FPGA-platform for stencil computation and its optimization methodology. En: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), Nr. 5, p. 1390–1402

[60] Wang, Shuo ; Liang, Yun: A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. En: *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE* IEEE, 2017, p. 1–6

[61] Williams, Samuel ; Waterman, Andrew ; Patterson, David: Roofline: an insightful visual performance model for multicore architectures. En: *Communications of the ACM* 52 (2009), Nr. 4, p. 65–76

[62] Yu, Chi W. ; Lamoureux, Julien ; Wilton, Steven J. ; Leong, Philip H. ; Luk,

Wayne: The Coarse-Grained/Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units. En: *International Journal of Reconfigurable Computing* 2008 (2008)

[63] Zohouri, Hamid R. ; Podobas, Artur ; Matsuoka, Satoshi: Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. En: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* ACM, 2018, p. 153–162