UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Implementación del algoritmo Differential Evolution en OpenCL

Mauricio Raúl Palavecino Nicotra

Director: Rafael Gadea Gironés

Trabajo de Fin de Máster para optar por el título de *Máster Universitario en Ingeniería de Sistemas Electrónicos*

Universitat Politécnica de Valencia,
Departamento de Ingeniería Electrónica

Valencia, 30 de octubre de 2016

Revisión 1.02

Índice de contenidos

Resumen y objetivos

Este trabajo describe la implementación del algoritmo de Evolución Diferencial (*Differential Evolution, DE*) cómo método de entrenamiento de una red neuronal tipo Perceptrón Multicapa sobre una plataforma basada en una FPGA-SoC, la placa DE1-SoC de la marca Terasic, utilizando aceleración de los cómputos por medio de procesamiento en paralelo, a través del lenguaje OpenCL.

El desarrollo de este trabajo se desprende de la investigación realizada por Rafael Gadea en el marco del área de *Diseño de Sistemas Electrónicos* del grupo de investigación del *Instituto de Instrumentación para Imagen Molecular* (I3M), la cual se centra en el estudio del proceso de optimización de la topología de redes neuronales artificiales utilizando computación distribuida sobre plataformas basadas en FPGAs [1][2].

Como principal objetivo se propuso implementar el algoritmo DE como método para evolucionar los pesos de un conjunto de redes neuronales de tipo Perceptrón Multicapa, acelerando la computación por medio de kernels descritos en OpenCL e implementados en hardware sobre un dispositivo acelerador FPGA.

Como objetivo secundario se planteó incorporar el módulo IP que computa la tangente hiperbólica, descrito en HDLs, como función auxiliar en la descripción de los kernels de OpenCL.

El trabajo concluyó sobre una correcta implementación del algoritmo DE como método de entrenamiento de redes tipo Perceptrón Multicapa, a la vez que se consiguieron resultados de aceleración más que interesantes (en algunos casos de casi un orden de magnitud) y sobre todo alentadores para continuar la temática de este trabajo u otros afines.

Además, el kernel desarrollado para acelerar el cómputo de la salida del Perceptrón Multicapa frente a una entrada estímulo, que computa el valor de salida de una neurona tipo perceptrón lineal con una función de transferencia de tangente hiperbólica, posee características muy interesantes de versatilidad, flexibilidad y portabilidad para la migración hacia otros dispositivos.

Descripción del documento

El presente documento se presenta ordenado cuatro capítulos básicos: *Introducción*, *Desarrollo*, *Resultados* y *Conclusiones*; más los capítulos correspondientes a un anexo (*Anexo I*) y a la bibliografía, ubicados sobre el final del mismo.

En primer lugar, se encuentra el capítulo de introducción, que da el marco teórico del trabajo y desarrolla los conceptos básicos de cada una de las temáticas. Se introducen los conceptos vinculados a: las redes neuronales artificiales, el algoritmo de evolución diferencial, las FPGA, OpenCL, y las implementaciones de OpenCL sobre FPGAs. Además, se menciona el marco de trabajo de la investigación dirigida por Rafaél Gadea en el área del grupo de investigación, del cual surge la idea y el objetivo de este trabajo, y por el cual se justifica el desarrollo.

Seguidamente, el capítulo de desarrollo se dedica a explicar cómo se implementó el algoritmo de evolución diferencial como método de entrenamiento de las redes neuronales, las estrategias utilizadas para el desarrollo de los kernels aceleradores y los principales pasos implementados en el programa del host.

El capítulo de resultados presenta, en primer lugar, el problema modelo a resolver con las redes neuronales, y con el cual se trabajó para realizar las mediciones temporales presentadas posteriormente. Además, se realiza un análisis de dichos resultados y se cierra el capítulo con los resultados de implementación de los kernels sobre la plataforma.

Por último, en el capítulo de conclusiones se establecen los objetivos alcanzados y se presentan las principales conclusiones extraídas del análisis de resultados. También, de este análisis surgen ideas y propuestas de trabajos futuros a desarrollar, las cuales son introducidas.

Introducción

Redes Neuronales Artificiales

Introducción

Las *Redes Neuronales Artificiales* (RNAs), o *Artificial Neural Networks* (ANNs) en inglés, son una clase de modelos computacionales que intentan emular el comportamiento del cerebro humano [3], utilizando una estructura en red en la cual los nodos, denominados *neuronas*, son procesos numéricos que involucran el estado de otros nodos [4].

Una de las definiciones que se estima más certera de 'Red de Neuronas Artificiales' es: “Las redes neuronales son conjuntos de elementos de cálculo simples, usualmente adaptativos, interconectados masivamente en paralelo y con una organización jerárquica que le permite interactuar con algún sistema, del mismo modo que lo hace el sistema nervioso biológico” [5].

Vale la pena aclarar que en términos de escala, un cerebro animal es mucho mayor que cualquier RNA implementada hasta la actualidad, a la vez que las neuronas artificiales también son más simples que su contrapartida biológica.

Las ANNs son ampliamente utilizadas en muchas áreas de la investigación, entregando resultados prometedores, dado que con ellas se pueden obtener soluciones a problemas que muchas veces no poseen solución aparente por métodos algorítmicos comunes, es decir, por la programación convencional. Además, se destacan por presentar una facilidad de uso e implementación notables.

Para poder resolver los problemas, las soluciones basadas en las RNAs parten de un conjunto de datos de entrada suficientemente significativo, el que se utiliza para enseñar a la red, con el objetivo de conseguir que la red aprenda automáticamente las propiedades deseadas. En contraposición con la programación convencional, el diseño de la red tiene menos que ver con cuestiones tal como los flujos de datos y la detección de condiciones, y más que ver con cuestiones como la selección del modelo de red, de las variables a utilizar y del tipo pre-procesamiento que se realizará a la información que formará el conjunto de

entrenamiento. Asimismo, el proceso por el que los parámetros de la red se adecuan a la resolución de cada problema no se denomina genéricamente programación sino que se suele denominar *entrenamiento neuronal*.

Aunque existe un gran número de modelos de RNAs, todas se pueden caracterizar por tres partes fundamentales, las que se desarrollan más adelante:

- La topología de la red,
- La regla de aprendizaje, y
- El tipo de entrenamiento

Para ubicar a las RNAs dentro de la ciencia, éstas forman parte de una de las dos ramas de la *Inteligencia Artificial* (IA), la denominada *Inteligencia Computacional*, o también conocida como *IA subsimbólica inductiva*. Esta rama de la IA está inspirada en las redes neuronales biológicas, es decir, en cómo funciona el cerebro, ya que utiliza un aprendizaje interactivo basándose en datos empíricos. Además de las RNAs, también forman parte de la IA inductiva la computación por lógica difusa y los algoritmos genéticos, las que muchas veces se combinan con las RNAs para conseguir sus mejores resultados [1].

Por otra parte, existe la *IA Convencional* o *Simbólico Deductiva*, entre los que se pueden mencionar los sistemas expertos, el razonamiento basado en casos o las redes bayesianas. Esta rama de la IA busca imitar el razonamiento humano a través de una lógica deductiva o la manipulación de símbolos, es decir, se basa en lo que hace el cerebro, y no en cómo éste funciona.

Es importante señalar, que ambas ramas de la IA son complementarias y no excluyentes, existiendo problemas en los cuales la aplicación de una u otra resulta más conveniente. También, existen otros problemas donde la aplicación conjunta de ambas ramas lleva a mejores resultados que los conseguidos con la aplicación de sólo una rama de forma independiente; e incluso, algunos para los que no se encuentran resultados favorables ni siquiera con la aplicación conjunta de ambas ramas de la IA.

Ventajas

Dado que las RNAs se inspiran en el cerebro humano e imitan su estructura y forma de funcionamiento, comparten algunas de sus características y se benefician de las siguientes ventajas:

- **Aprendizaje adaptativo**, dado que poseen la capacidad de obtener respuestas frente a situaciones desconocidas, ya que se basan en un entrenamiento o experiencia inicial.
- **Auto-organización**, dado que una RNA crea su propia organización o representación de la información, que recibe durante la etapa de aprendizaje.

- **Tolerancia a fallos**, debido a que la información está distribuida por todas las conexiones de la red, se genera cierta redundancia. Con ello, si se daña parcialmente la red, aunque su desempeño puede verse degradado, algunas capacidades son retenidas.
- **Flexibilidad**, ya que permiten procesar información difusa, con ruido, inconsistente o con otros cambios.
- **Capacidad de generalización**, siendo esta una de las principales ventajas, dado que ante el ingreso de nuevos datos, una RNA es capaz de producir resultados coherentes de acuerdo a la naturaleza del problema para el cual ha sido entrenada.
- Son susceptibles de realizarse con **procesamiento en paralelo**, lo que permite el diseño de hardware específico a fin obtener mejores respuestas temporales.

Muchos autores incluyen como una ventaja la capacidad de las redes de funcionar o generar respuestas en tiempo real. Sin embargo, el autor de este trabajo considera que esta capacidad es muy dependiente de la implementación, es decir, de la plataforma de procesamiento y sus capacidades de cómputo, el tipo y las dimensiones de la red, etc.; y de los requerimientos temporales del problema objetivo sobre el cual se aplicarán las RNAs como solución.

Desventajas

Por otro lado, como cualquier método de resolución de problemas existen limitaciones, desventajas e inconvenientes, algunos de los cuales también dependen del tipo red, es decir, de cómo se implementan las tres partes fundamentales: topología, regla de aprendizaje y tipo de entrenamiento. Algunas desventajas comunes de las RNAs son:

- No siempre se encuentra la topología adecuada para un problema,
- El entrenamiento puede producir un problema de sobre-ajuste: aprender muy bien el conjunto de entrenamiento, pero no ser capaces de procesar correctamente nueva información, perdiendo así su característica de generalización,
- Su entrenamiento es susceptible de ser engañado al caer en mínimos locales, lo que detiene su aprendizaje,
- Su entrenamiento posee una alta dependencia de los valores iniciales de los pesos de la red,
- El entrenamiento suele demandar una gran cantidad de procesamiento, que conlleva una gran demanda de tiempo antes de que la red sea funcional.

Por último, una característica muy criticada de las RNAs, independientemente de qué tipo de red se trate,

es que funcionan como una 'caja negra', dado que habitualmente realizan un complejo procesamiento que supone una gran cantidad de operaciones, por lo que no siempre es posible seguir paso a paso el razonamiento que las ha llevado a extraer sus conclusiones.

Aplicaciones

La utilización de las RNAs se ha expandido sobre diferentes ramas de la ciencia y la industria, tal como las ciencias biológicas, las financieras, las medio ambientales, la salud, la industria en general e incluso el ámbito militar [6]. Las siguientes son sólo algunas aplicaciones en las que se utilizan las RNAs:

- Ámbito Financiero:
  - Predicción de la bolsa
  - Predicción de quiebras
  - Detección de fraude
  - Predicción de precios
- Minería de datos:
  - Predicción
  - Clasificación
  - Detección de cambios y desviación
  - Modelado de respuestas
  - Análisis de series temporales
- Medicina:
  - Diagnósticos médicos
  - Detección y evaluación de fenómenos médicos
  - Estimación de costos de trataiento
- Ciencia:
  - Reconocimiento de patrones
  - Reconocimiento de voz [7]
  - Reconocimiento de imágenes
  - Identificación de compuestos químicos
  - Modelado de sistemas físicos
  - Evaluación de ecosistemas
  - Identificación de polímeros
  - Reconocimiento de genes
  - Clasificación botánica
  - Procesado de señal: filtrado neuronal
  - Análisis de sistemas biológicos
  - Análisis e identificación de olores
  - Recuperación de datos corruptos

- Compresión de datos
- Encriptación

- Ámbito Energético:
  - Predicción de demanda eléctrica
  - Estimación de carga a corto y largo plazo
  - Predicción de precios del gas y carbón
  - Control de sistemas de potencia
  - Monitoreo de presas eléctricas

- Industria:
  - Control de procesos
  - Control de calidad
  - Predicción de temperaturas

- Otros:
  - Predicción de apuestas deportivas
  - Previsión del tiempo
  - Desarrollo de juegos
  - Optimización de problemas de enrutamiento
  - Estimaciones de producciones agrícola

Existen muchas otras aplicaciones de las RNAs en funcionamiento, y otras tantas que continuamente se van desarrollando.

Topología de la red

Emulando a las redes neuronales biológicas, las RNAs se componen de unidades elementales denominadas *neuronas*. Cada neurona recibe una serie de entradas a través sus interconexiones y emite una señal de salida. El valor de esta salida viene dado por tres funciones:

- Una **función de propagación** (también conocida como *función de excitación*), que por lo general consiste en el sumatorio de los valores de todas las entradas, cada una multiplicada previamente por el peso de su interconexión, denominado habitualmente como *peso sináptico*. Si valor del peso es positivo, la conexión se denomina *excitatoria*; mientras que si es negativo, *inhibitoria*.
- Una **función de activación**, que modifica la anterior función, y que en ocasiones no se implementa.
- Una **función de transferencia**, que se aplica al valor devuelto por la función de activación. Esta función se utiliza para acotar la salida de la neurona y generalmente se define según la interpretación que se desea dar a los valores de salida. Las funciones de transferencia más habituales son la función escalón, para producir salidas binarias de 0 o 1; sigmoidea, para acotar la salida entre 0 y 1; y la función tangente hiperbólica, para obtener valores entre -1 y 1.

Estas funciones definen el modelo funcional de una neurona artificial, es decir, definen el tipo de neurona). Algunos de los más conocidos son el perceptrón simple (salida binaria) y el Adaline (salida lineal). La siguiente figura resume la estructura de un perceptrón simple, en este caso de cinco entradas x_1, x_2, x_3, x_4 y x_5, con una función de propagación definida como el sumatorio de las entradas ponderadas por sus correspondientes pesos sinápticos w_1, w_2, w_3, w_4 y w_5, y las funciones de activación y de transferencia implementadas en conjunto como la función escalón, con lo que se obtiene una salida binaria de 0 o 1.

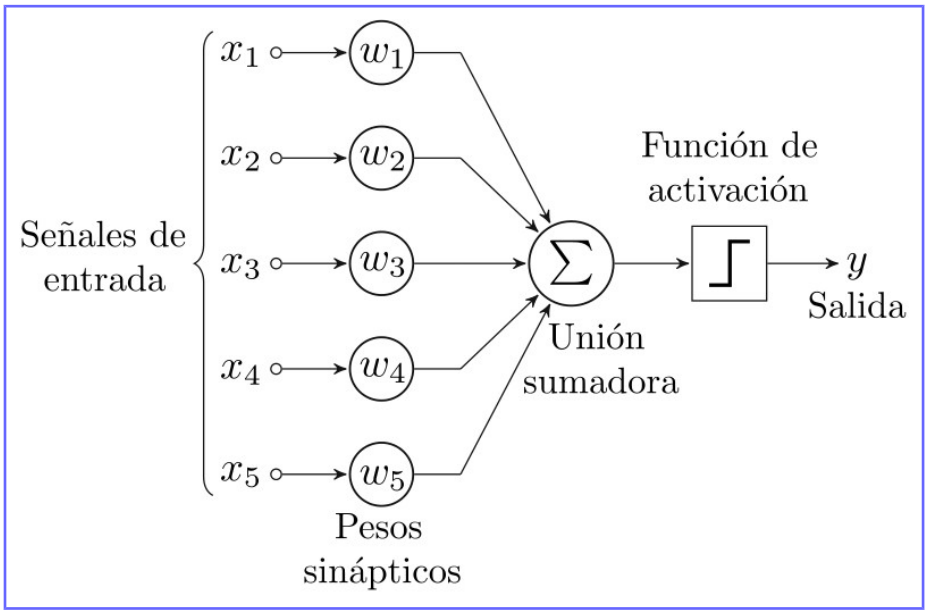

Ilustración 1: Perceptrón simple de cinco entradas.

Entonces, la **topología de red** de una red neuronal artificial se define por la **arquitectura de la red**, y por la **inteligencia de la red**. La estructura o arquitectura de la red queda definida por el tipo de red y la cantidad de neuronas implementadas; mientras que la inteligencia de la red reside en el valor de los pesos de cada una de sus conexiones, es decir, en el valor de ponderación del aporte que una neurona le realiza a otra.

Los tipos de redes se diferencian básicamente por:

- El tipo de neuronas implementadas en la red, y
- Su distribución y la forma en que se conectan entre sí. Esto puede ser:
  - Totalmente conectadas, localmente conectadas o parcialmente conectadas,
  - Re-alimentadas (con retro-propagación) o de sólo propagación hacia adelante,
  - Con aporte o sin aporte de *bias*,
  - Con memoria o no, etc.

Elaborar una lista completa de todos los tipos de redes neuronales es prácticamente imposible, dada la cantidad de nuevas arquitecturas que se inventan en todo momento. Incluso, si la lista se elaborara, sería todo

un desafío leerla y no pasar por alto algún tipo. A continuación se presentan algunas arquitecturas de redes neuronales:

- Perceptrón, que también es considero por sí solo como una red neuronal;
- Feed Forward (FF);
- Deep Feed Forward (DFF);
- Recurrent Neural Networks (RNN);
- Long/Short Term Memory (LSTM);
- Auto Encoder (AE);
- Hopfield Network (HN);
- Boltzmann Machine (BM);
- Deep Convolutional Networks (DCN);
- Deep Convolutional Inverse Graphics Networks (DCIGN);
- Deep Residual Networks (DRN);
- Kohonen Network (KN).

Cada tipo de red posee sus características y por ello, es más adecuada para solucionar una clase de problemas por sobre otros. Si se desea conocer brevemente las características de estas redes, e incluso encontrar otros tipos de redes, puede servir como punto de partida [8].

Una de las arquitecturas de RNAs más empleadas es la denominada *Perceptrón Multicapa*, o MLP del inglés *Multi-Layer Perceptron*. Esta arquitectura es ampliamente utilizada debido a que es relativamente simple, a la vez que permite resolver problemas que no son linealmente separables, siendo ésta la principal limitación del perceptrón simple o del Adaline.

La siguiente imagen presenta un MLP de cuatro entradas y una salida. En esta se pueden visualizar (diferenciados por color) los tres tipos de capas característicos de un MLP:

- Capa de entrada (verde): constituida por aquellas neuronas que introducen los patrones de entrada en la red. En estas neuronas no se realiza ningún procesamiento.
- Capas ocultas (azul): formada por aquellas neuronas cuyas entradas provienen de capas anteriores y cuyas salidas pasan a neuronas de capas posteriores.
- Capa de salida (rojo): neuronas cuyos valores de salida se corresponden con las salidas de toda la red.

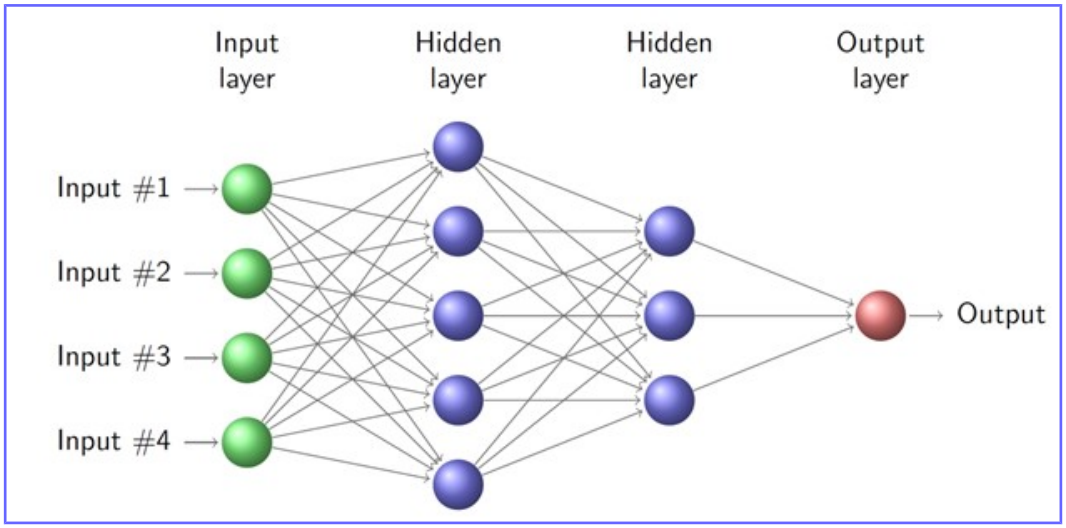

Ilustración 2: Perceptrón multicapa de cuatro entradas, una salida y dos capas ocultas.

Así, las estructuras o arquitecturas de los MLP pueden variar según:

- El tamaño de la red y la distribución de las neuronas, que involucran el número de capas ocultas y el número de neuronas utilizadas en cada una de ellas. Por su lado, el número de neuronas de las capas de entrada y la de salida quedan definidas por el problema a resolver (número de entradas y salidas), mientras que las capas y neuronas ocultas forman parte del proceso de búsqueda de la topología óptima de la red.
- El grado de conexión de las neuronas, pudiendo ser localmente o totalmente conectadas. En este último caso, todas las neuronas de una capa se conectan a todas las neuronas de la capa siguiente, tal como sucede en la figura de ejemplo.
- La presencia de bias o no, que se implementa como un valor independiente en cada neurona, que siempre se añade al sumatorio.

Para un determinado problema, la estructura del MLP se define con el proceso de optimización, es decir, con el proceso de búsqueda de la mejor arquitectura que se ajuste a la resolución del problema. Normalmente, el objeto de entrada se transforma en un vector de características, cuyos valores son descriptivos del objeto y que pasan a ser los elementos de entrada de la red. El número de características no debe ser demasiado grande, a causa de la maldición de la dimensionalidad, pero debe ser lo suficientemente grande como para predecir la salida con una buena precisión.

Entrenamiento

Una vez establecida la estructura de una RNA, la misma debe ser entrenada para dotarla de inteligencia, es decir, que se deben definir los valores de los pesos de las conexiones de la red. Luego, tras el

entrenamiento, la red debería ser capaz de procesar nueva información y generar resultados correctos o aceptables, dentro de ciertos márgenes.

Existen dos tipos de entrenamiento o procesos de aprendizaje de la RNA, el supervisado y el no supervisado, y la elección de uno u otro depende de: la colección de datos con que se dispone para entrenar la red, y también del tipo de red implementada.

El aprendizaje supervisado es una técnica utilizada para deducir una función a partir de un conjunto de datos de entrenamiento. Estos datos de entrenamiento consisten en pares de objetos (normalmente vectores) donde: una componente está formada por los datos de entrada o estímulos, y el otro, por los resultados deseados para sus correspondientes estímulos. La salida de la función puede ser un valor numérico (como en los problemas de regresión) o una etiqueta de clase (como en los de clasificación). El objetivo del aprendizaje supervisado es el de crear una función capaz de predecir el valor de salida correspondiente a cualquier objeto de entrada después de haber visto una serie de ejemplos: los datos de entrenamiento. Para poder lograr esto, la red tiene que generalizar, a partir de los datos presentados, las respuestas a las situaciones no analizadas previamente.

Entonces, con el proceso de entrenamiento la red se dota de su inteligencia, es decir, que con este proceso se definen los pesos sinápticos de red, con lo que la misma se convierte en una red funcional.

Para las RNA del tipo MLP, es habitual el empleo del algoritmo de Retropropagación, BP o BPA de sus siglas en inglés, o alguno de sus derivados, como el *Resilent Backpropagation Algorithm*, RBP de sus siglas en inglés, u otros métodos basados en gradientes. Sin embargo, existen estudios que han encontrado que se puede entrenar la red con algoritmos evolutivos, consiguiendo buenos resultados [9][10]. Tal es el caso del empleo del algoritmo de Evolución Diferencial, que se describe a continuación.

El algoritmo de Evolución diferencial

El algoritmo de *Evolución Diferencial*, o simplemente DE (*Differential Evolution*), es un algoritmo de optimización, estocástico y basado en poblaciones, desarrollado para optimizar funciones reales evaluadas con parámetros reales.

El algoritmo DE surge en base a los intentos de Ken Price de resolver el problema de ajuste del polinomio de Chebychev planteado a él por Rainer Storn. Price, tuvo la idea de perturbar la población de vectores utilizando diferencias ponderadas entre ellos, idea que compartió con Storn y que concluyó entre 1994 y 1996 en lo que hoy se conoce como Evolución Diferencial.

El algoritmo DE puede verse como un método de búsqueda del valor mínimo de una función real evaluada en un espacio dimensional perteneciente al dominio $\mathbb{R}^{\mathbf{D}}$, es decir, con D parámetros de entrada reales. Este parámetro D del algoritmo queda definido por el problema, siendo igual al número de variables de la función, y lleva el nombre de *dimensionalidad del problema*.

La optimización global es necesaria en diversos campos de las ciencias: ingenierías, estadísticas, financieras, entre otras. Sin embargo, muchos problemas tienen funciones objetivo que son: no diferenciables, no contínuas, no lineales, ruidosas, multi-dimensionales, con mínimos locales, y/o incluso de naturaleza estocástica. Estos problemas llegan a poseer soluciones analíticas muy difíciles de conseguir, o incluso imposibles. Entonces, por sus características, el algoritmo DE se presenta como una excelente alternativa para buscar soluciones numéricas aproximadas a estos problemas.

Vale la pena señalar que el algoritmo DE pertenece a la categoría de la *computación evolutiva*, la cual es una rama de la *inteligencia artificial*, que involucra problemas de optimización combinatoria, y que se inspira en los mecanismos de la Evolución Biológica. Como tal, el algoritmo DE se basa en una población de soluciones candidatas, las cuales se recombinan y mutan para producir nuevos individuos, los cuales serán elegidos de acuerdo al valor de su función de desempeño. Sin embargo, lo que caracteriza al algoritmo DE es el uso de vectores de prueba, los cuales compiten con los individuos de la población actual a fin de sobrevivir en la próxima generación.

Aunque el algoritmo DE es considerado un algoritmo con una estrategia evolutiva excepcionalmente simple, a la vez resulta en un algoritmo significativamente rápido y robusto. Posee una baja cantidad de parámetros de control, que son:

- N o NP: el número de vectores padre, el tamaño de la población o el número de individuos de la misma, que debe ser igual o mayor a 4;
- F: el factor de mutación o factor de escala, que habitualmente se selecciona en un valor comprendido en el rango de [0, 2];
- CR: el factor de recombinación (*crossover rate*), un valor comprendido entre (0,1); y,
- G: es el máximo número de generaciones que se realizarán;

El desempeño del algoritmo es muy sensible a estos valores de control, y cada problema puede responder de diferente manera a cada conjunto de estos parámetros.

A su vez, existen modificaciones, que buscan mejorar la calidad de los resultados conseguidos o la velocidad en que convergen sus soluciones [11][12]. Por ejemplo, algunos métodos se centran en modificar el factor de escalado entre generaciones, ya sea de forma predefinida (determinista), adaptativa (en base a

alguna retro-alimentación), o incluso de forma evolutiva (evolucionando como parte de otro algoritmo evolutivo) [13].

Funcionamiento

El algoritmo asume que las variables del problema a optimizar están codificadas como un vector de números reales. La longitud de estos vectores (D) es igual al número de variables del problema. Se define entonces una población de vectores padre Px, compuesta de NP vectores o individuos. Estos vectores, reciben el nombre de vectores padre, y se designan como $\mathbf{x}^{g}_{p}$, en donde p es el índice del individuo dentro de la población (p = 0...NP-1) y g es la generación correspondiente. Cada vector padre está a su vez compuesto de las variables del problema $x^{g}_{p,m}$, siendo m el índice de la variable en el individuo p (m = 0...D-1).

Entonces, se tiene, que para una dada generación g (índice que no se coloca para simplificar la descripción), la población Px está compuesta por el conjunto de vectores padre **x**:

$$Px = \{ \mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_{NP-2}}, \mathbf{x_{NP-1}} \}$$

A su vez, cada vector $\mathbf{x}^{g}_{p}$, está formado por un conjunto de D variables, que se corresponden con las variables del problema, $x^{g}_{p,m}$, siendo m el índice de la variable del vector p (m = 0…(D-1)), mientras que p y g, conservan el mismo significado que anteriormente. Esto es, para una dada generación g:

$$\mathbf{x_p} = \{ x_{p,0}, x_{p,1}, x_{p,2}, \dots, x_{p,D-2}, x_{p,D-1} \}$$

Como se explicará seguidamente, existen tres conjuntos de poblaciones más, con sus respectivos individuos, que poseen las mismas dimensiones que Px, y los índices conservan su significado. Estas poblaciones son:

- La población de vectores donadores o vectores mutados Pv, formada por NP vectores donadores $\mathbf{v}^{g}_{p}$, de tamaño D;
- La población de vectores de prueba Pu, formada por NP vectores de prueba $\mathbf{u}^{g}_{p}$, de tamaño D;
- La población objetivo Px^{*}, que se corresponde con la población de vectores padre de la siguiente generación.

El algoritmo DE, como la mayoría de los algoritmos genéticos, posee cuatro etapas bien definidas:

- La **inicialización**, que se ejecuta al comienzo del algoritmo una única vez. Se encarga de darle los valores iniciales a los parámetros de cada uno de los individuos de toda la población. En el caso del DE, los valores de los parámetros en esta inicialización se acotan habitualmente a un rango que depende del problema a resolver.

- La **mutación**, se encarga de generar nuevos vectores a partir de la combinación lineal de otros tres vectores seleccionados aleatoriamente, expandiendo así el espacio de búsqueda. En esta etapa se calculan los NP vectores mutados, que habitualmente se designan como $\mathbf{v}^{g}_{p}$, siendo g y p, el índice del individuo dentro de la población y la generación correspondiente, respectivamente, y se define una nueva población Pv de vectores mutados o vectores donantes. Cada vector mutado $\mathbf{v}^{g}_{p}$ se calcula como:

$$\mathbf{v}^{g}_{p} = \mathbf{x}^{g}_{r0} + F \cdot (\mathbf{x}^{g}_{r1} - \mathbf{x}^{g}_{r2})$$

  siendo F el factor de mutación; mientras que r0, r1, y r2, son índices aleatorios, diferentes entre sí y comprendidos en el rango [0,(NP-1)].

- La **recombinación** (*crossover*), se encarga obtener los vectores de prueba $\mathbf{u}^{g}_{p}$, mezclando los parámetros de $\mathbf{x}^{g}_{p}$ y $\mathbf{v}^{g}_{p}$, en base al sorteo de un número aleatorio que se compara contra el factor de recombinación CR, como sigue:

$$u_{p,m} = \begin{cases} v_{p,m} & si \quad rand_{p,m(float(0,1))} \leqslant CR \quad o \quad si \quad m = rand_{p,m(integer(0,D-1))} \\ x_{p,m} & en\,caso\,contrario \end{cases}$$

  Para todo p = {0,1,2,..(NP-1)}; y todo m = {0,1,2..(D-1)}. Esto es que, los elementos o variables del vector donador $\mathbf{u}_{p}$, se insertan en el vector de prueba $\mathbf{v}_{p}$ con una probabilidad CR. La segunda condición elección: m = $rand_{p,m(integer(0,D-1))}$, se establece para que bajo ninguna circunstancia el vector de prueba sea idéntico al vector padre, es decir, que nunca se cumpla que $\mathbf{u}_{p} = \mathbf{x}_{p}$.

- La **selección**, se encarga de formar la población objetivo Px^{*}, es decir, la nueva población de vectores padres Px para la siguiente generación g+1, a partir de cada par $\{\mathbf{x}_{p}, \mathbf{v}_{p}\}$. Para ello, inserta como vector de la nueva generación $\mathbf{x}^{g+1}{}_{p}$ al vector $\mathbf{x}^{g}_{p}$ o al vector $\mathbf{v}^{g}_{p}$, según el que produzca el menor valor de la función objetivo. Pese a que es una etapa muy simple de comparación y selección, existe un paso implícito y no tan simple que se corresponde con la **evaluación de la función objetivo** con cada uno de los vectores padres $\mathbf{x}^{g}_{p}$ y cada uno de los vectores de prueba $\mathbf{v}^{g}_{p}$. Entonces si se puede comparar el resultado de la función objetivo evaluada por cada vector padre $\mathbf{x}^{g}_{p}$ con el resultado de la función objetivo evaluada por sus correspondientes vectores de prueba correspondientes $\mathbf{v}^{g}_{p}$. Esto es:

$$x_{p}^{g+1} = \begin{cases} u_{p}^{g} & si \quad f_{obj}(u_{p}^{g}) \leqslant f_{obj}(x_{p}^{g}) \\ x_{p}^{g} & en\,caso\,contrario \end{cases}$$

  Para todo p = {0,1,2, … ,(NP-1)}.

Las últimas tres etapas se repiten generación tras generación en el mismo orden en que se expusieron

hasta que una condición de término sea satisfecha. Esto puede ser hasta que se complete el número máximo de generaciones, se llegue a un tiempo máximo transcurrido, se alcance una cierta calidad en la solución, etc. La siguiente imagen representa el diagrama de flujo básico del algoritmo de Evolución Diferencial.

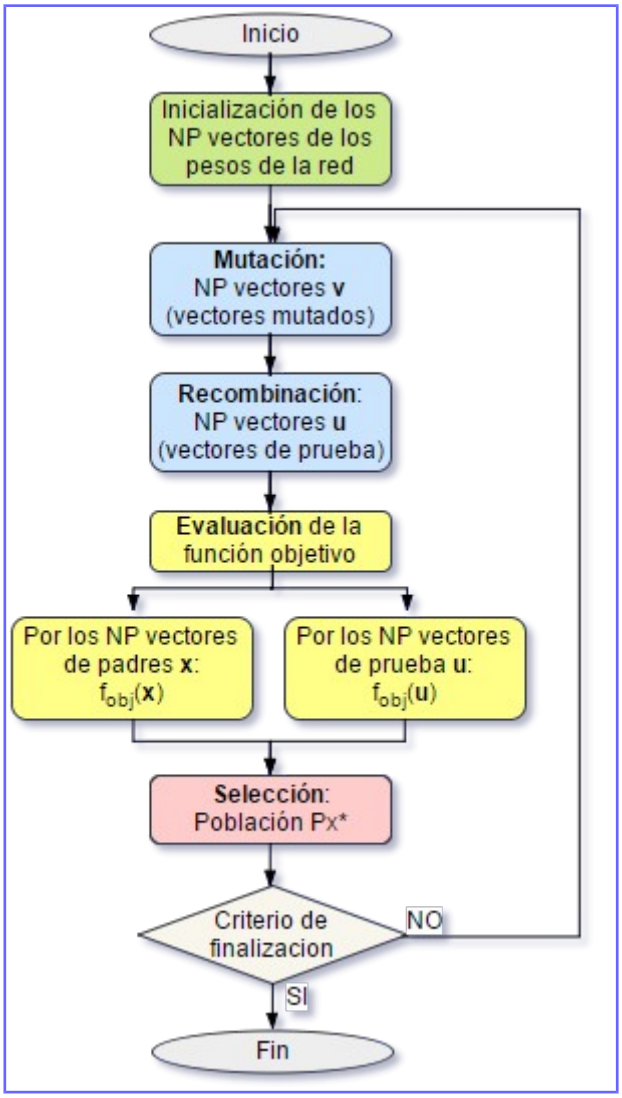

Ilustración 3: Fiagrama de Flujo del algoritmo DE.

En la siguiente ilustración se presenta el flujo de datos para el conjunto de los NP individuos de la población, involucrando las tres etapas fundamentales del algoritmo DE: la mutación, la recombinación y la selección, hasta que se obtiene la población objetivo, correspondiente a la nueva generación.

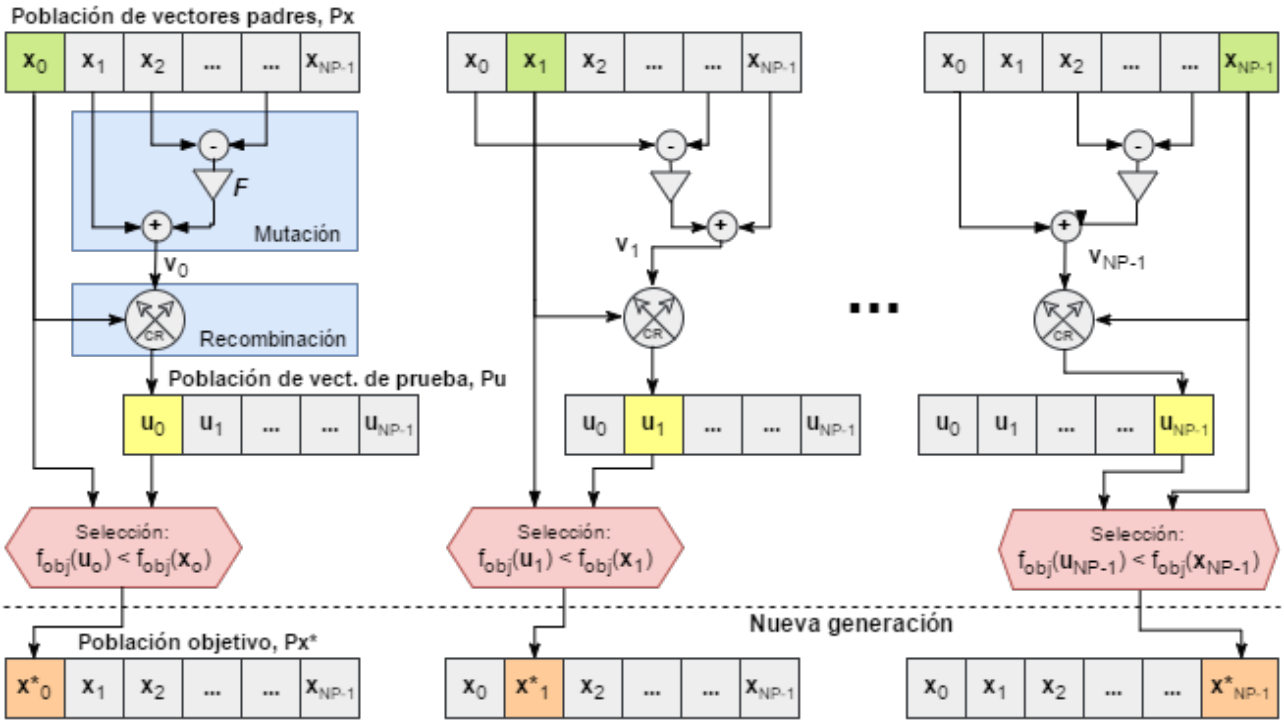

Ilustración 4: Flujo de datos del algoritmo DE.

Este diagrama, representa muy bien el detalle de las operaciones realizadas de cada una de las etapas del algoritmo, y cómo se distribuyen en torno a todos los individuos de cada una de las poblaciones.

Al momento de poner a funcionar el algoritmo para resolver un problema, se debe realizar la elección de los parámetros del mismo, esto es, se debe definir el tamaño de las poblaciones NP, el factor de escalado F, y el factor de recombinación CR. Aunque la mejor selección de estos parámetros varía de un problema a otro, existen estudios que recomiendan rangos entre los cuales es conveniente comenzar el estudio de una optimización. Por ejemplo, en [14] se mencionan varias sugerencias, tales como:

- El número de vectores padres debe configurarse en torno a 10 veces el número de parámetros de la función a optimizar (esto sería NP ≈ 10·D), cuando sea posible. Sin embargo, también se advierte que cuando NP crece por encima del orden de 40, la convergencia en los resultados no mejora sustancialmente;
- El valor de F debe configurarse en 0.8, o bien aplicarse aleatoriamente en el rango [0.5 , 1.0] en cada nueva generación; o,
- El valor de CR puede configurarse en 0.9 cuando existe dependencia entre los parámetros del problema, lo que es la situación más habitual de los problemas reales.

Incluso, los autores recomiendan que, siempre es bueno revisar la función objetivo definida para el problema, dado que una buena elección de la misma puede marcar la diferencia en los resultados, sobre todo cuando aparecen problemas de falta de convergencia.

FPGAs

Los *Arreglos de Compuertas Programable por Campo* o simplemente FPGAs, por siglas en inglés de *Field-Programmable Gate Array*, son circuitos integrados que no poseen un uso específico tras su fabricación, aunque si están orientados a aplicaciones digitales. Forman parte de la familia de dispositivos denominados: *Dispositivos Lógicos Programables*, o PLD (*Programmable Logic Devices*), y como tales, deben ser configuradas por el usuario para convertirse en dispositivos funcionales, consiguiendo de esta manera que la FPGA pase a ser un sistema de propósito específico. Existe una inmensa cantidad de diseños que pueden ser implementados en este tipo de dispositivos, que se encuentran sólo limitados por los recursos disponibles en la FPGA utilizada.

La arquitectura de las FPGAs se basa en tres elementos básicos: una matriz de **celdas** o **elementos lógicos**, a menudo agrupadas en bloques lógicos; una **red de interconexión** interna; y los **módulos de entrada-salida** (E/S), conectados a los pines del encapsulado para la comunicación con el exterior. Estos tres elementos son programables (configurables), lo que flexibiliza en gran medida los diseños posibles de implementar en cualquier FPGA. Esta arquitectura básica se representa en la siguiente figura.

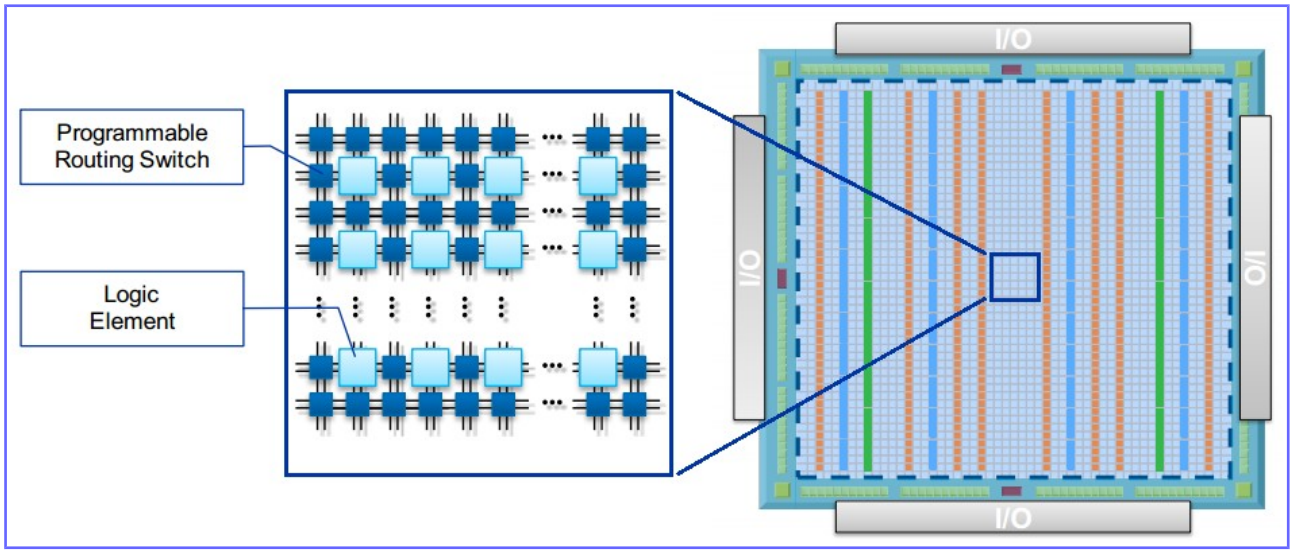

Ilustración 5: Arquitectura básica de las FPGA.

Estos tres elementos difieren entre sí en sus implementaciones internas según la tecnología, familia o fabricante, aunque la tecnología más ampliamente utilizada es la basada en SRAM para mantener la configuración del dispositivo. En cuanto a la estructura interna de los elementos lógicos, la mayor parte de los dispositivos presentan celdas lógicas basadas en una *Look-Up Table*, LUT, seguida de un registro del tipo *Flip-Flop*, tal como se ilustra en la siguiente figura.

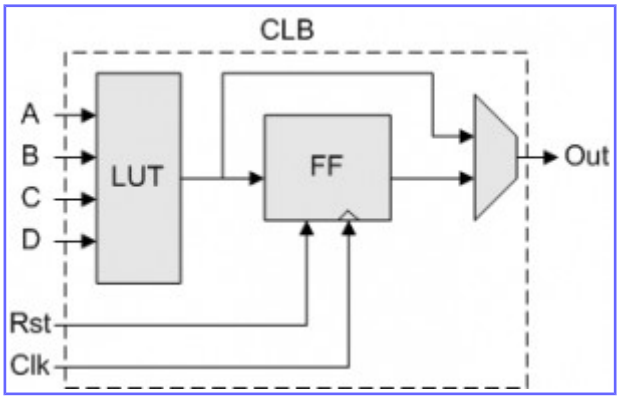

Ilustración 6: Estructura genérica de las celdas lógicas en dispositivos FPGAs.

Desde hace ya varios años las FPGAs se han convertido en los dispositivos lógicos programables por excelencia, abarcando la mayor parte del mercado de los PLDs, salvo por una pequeña porción que aún mantienen los CPLDs (*Complex Programmable Logic Devices*); tanto para su uso en investigación y desarrollo, como en la industria. Por ello, año tras año, conforme han ido evolucionado las tecnologías, la arquitectura básica antes descrita se ha ido enriqueciendo con más elementos embebidos en silicio, tal como módulos de memoria RAM y FLASH, módulos de procesamiento digital de la señal (DSPs), administradores de señales de reloj, convertidores analógico-digital (ADC, *Analog-Digital Converter*) y digital-analógico (DAC, *Digital-Analog Converter*), e incluso microprocesadores completos, dando origen a las familias de los AP SoC (*All Programmable System on Chip*) para Xilinx, y FPGA SoC (*FPGA System on Chip*) para Altera, los dos fabricantes con mayor cuota de mercado de las FPGAs. A su vez, cada año se incrementa la densidad de integración de transistores, lo que facilita el aumento en la cantidad de recursos integrados en las FPGAs y la máxima frecuencia de trabajo soportada, aumentando entonces tanto la capacidad como la velocidad de procesamiento estos los dispositivos.

Por su arquitectura, las FPGAs, son dispositivos que alientan a trabajar con un paralelismo masivo de datos, ya pueden implementarse tantos elementos de cómputo funcionando de forma concurrente como lo permitan sus capacidades, logrando un paralelismo a nivel de instrucción; como también, por su alto número de registros distribuidos, permiten implementar caminos de datos con un elevado número de etapas de segmentación (*pipeline*).

Por otro lado, dada la complejidad y la cantidad de recursos de estos dispositivos, es necesario el uso de sofisticadas herramientas de software para poder completar todo el ciclo de diseño que se requiere para implementar un sistema en una FPGA, a la vez que se hace un buen uso de los recursos disponibles. Por ello, cada fabricante dispone de un conjunto de herramientas, que van desde el administrador de proyectos y los simples editores de texto, hasta avanzadas herramientas de diseño de muy alto nivel. Entre estas últimas, se destacan:

- *System Generator* de la empresa Xilinx, y *DSP Builder* de Altera; ambas desarrolladas en conjunto con la empresa MATLAB e integradas con Simulink. Son herramientas gráficas orientadas al procesamiento de señales.
- *SDAccel Development Environment* de Xilinx [15], y *Altera SDK for OpenCL* de Altera [16], para el desarrollo de plataformas aceleradoras basadas en FPGAs utilizando el modelo de programación OpenCL que se presenta a continuación.

OpenCL

El estándar *Open Computing Language* (OpenCL) es un modelo de programación en paralelo, abierto, libre de regalías (royalty-free) y unificado, orientado a la aceleración de algoritmos en plataformas heterogéneas, entendiéndose como tales a aquellas basadas en un host del tipo *Central Processing Unit* (CPU), y uno o más dispositivos aceleradores: *Graphic Processor Units* (GPUs), *Digital Signal Processors* (DSPs), *Field-Programmable Gate Arrays* (FPGAs), incluso otras CPUs, o cualquier otro dispositivos de cómputo; pudiendo incluso co-existir en un mismo sistema más de un tipo de acelerador. Un ejemplo de ello puede ser un ordenador de escritorio, al cual se le adiciona una placa gráfica basada en GPUs y una placa aceleradora basada en FPGAs.

El entorno OpenCL incluye un lenguaje, el OpenCL C (un subconjunto del lenguaje ISO C99 con extensiones para paralelismo), una interfaz de programación (API, *Application Programming Interface*), librerías y un sistema de ejecución para soportar el desarrollo de software.

OpenCL fue creado por la compañía Apple Inc. y desarrollado posteriormente en conjunto con AMD, Intel, y NVIDIA. Desde el año 2008 forma parte del Grupo Khronos, que creó el *Compute Working Group* para que se encargue de mantenerlo y de llevar a cabo los procesos de estandarización. Actualmente el número de empresas que participan es mucho mayor, entre las que se pueden destacar Altera y Xilinx, las dos empresas con mayor cuota en el mercado de las FPGAs.

Dos características muy importantes de OpenCL, que lo diferencian de otros modelos y que han marcado su éxito, son que:

- Permite una gran portabilidad de los kernels o aceleradores descritos en OpenCL C, entre diferentes plataformas.
- Permite la utilización de más de un tipo de acelerador, pudiendo coexistir en el mismo sistema más de una plataforma aceleradora, que a su vez pueden contar con más de un dispositivo acelerador. Cada plataforma puede estar basada en diferentes tecnologías (CPU, GPGPU o FPGA) e incluso ser de diferentes fabricantes.

La principal idea del estándar OpenCL es proveer una interfaz y un lenguaje de programación común para todas las plataformas, mientras que son los fabricantes quienes lo adaptan y se encargan de su implementación sobre sus propias plataformas y dispositivos. Así es que, en el año 2013, Altera introduce su conjunto de herramientas de desarrollo (SDK , de S*oftware Development Kit*) para OpenCL, para sus dispositivos FPGA (familias Cyclone V. Stratix V y Arria 10). Este SDK permite desarrollar aceleradores a partir del lenguaje OpenCL C, sintetizarlos en hardware y posteriormente implementarlos en la FPGA, en un proceso mucho más ágil que el conseguido con las implementaciones tradicionales con Lenguajes de Decripción de Hardware (HDLs, de *Hardware Description Language*).

De esta forma, OpenCL permite a los equipos de ingeniería elegir productos basados en tecnologías FPGA sin llegar al nivel de detalle al que los ingenieros en hardware deben llegar con las descripciones en HDLs. Con ello, OpenCL se convierte en una poderosa herramienta que reduce el tiempo al mercado para los productos aceleradores basados en FPGAs.

La arquitectura OpenCL (*v1.0* [17]) se organiza en cuatro niveles o jerarquías de modelos:

- Modelo de Plataforma
- Modelo de Ejecución
- Modelo de Memoria
- Modelo de Programación

Modelo de plataforma

El modelo de plataforma en OpenCL se ilustra en la siguiente figura, en la cual se percibe la presencia de un host, conectado a uno o más dispositivos OpenCL (adoptando la terminología en inglés: *devices*), los cuales se dividen internamente en una o más unidades de cómputo (CUs, *compute units*), que a su vez se dividen en una o más unidades de procesamiento (PEs, *processor units*). Así, el sistema queda claramente definido en dos partes: el **host** y los **devices**.

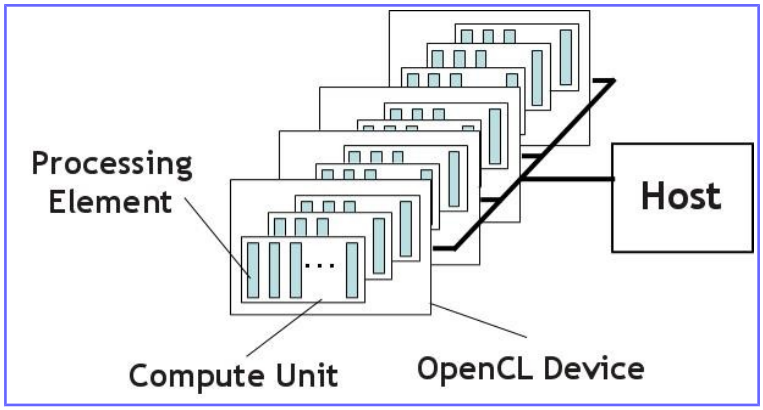

Ilustración 7: Modelo de plataforma de OpenCL.

En una aplicación OpenCL, es el host quien maneja y administra el sistema, ya que se comunica con los devices por medio de colas de comandos, envía y recibe datos hacia y desde la memoria de los mismos, y pone en ejecución a las unidades de cómputo para que procesen los datos que correspondan.

Modelo de ejecución

El modelo de ejecución de OpenCL queda definido en términos de dos unidades de ejecución: el **programa de host**, ejecutado en el host; y, los **kernels**, ejecutados en uno o más devices. Los kernels representan la parte de la programación en paralelo de OpenCL, y son los encargados de acelerar los cómputos del sistema mientras que por su lado, el programa del host es quien define el contexto para los kernels y administra su ejecución.

El programa del host puede ser implementado en cualquier lenguaje de programación soportado por el mismo y para el cual esté definida la API de OpenCL. En la mayoría de los casos, el lenguaje utilizado para implementar el programa del host es C/C++, aunque también existen desarrollos con el uso de Java y Python. Entonces, el programa del host se compila con el compilador tradicional, y será el que haga llamados a las funciones de la API de OpenCL para transferir datos a los devices y para ordenar la ejecución de los kernels.

El comportamiento de los kernels se describe en lenguaje OpenCL C, en general en uno o más archivos independientes del programa del host, archivos de extensión .cl. Sin embargo, también existen implementaciones donde se definen como un string dentro del mismo programa del host, pero no son las más habituales. Luego, la compilación de los kernels puede realizarse antes de la ejecución del programa del host, de forma *offline* (en el caso de las FPGAs), o durante su ejecución, de forma *online* (en el caso de las GPUs). En ambos casos se utiliza un compilador proporcionado por el fabricante del device que traduce el programa fuente en un programa objeto ejecutable.

Los kernels se ejecutan dentro de un contexto definido y manejado por el host. Este contexto define el entorno en el cual los kernels se ejecutan e incluye los siguientes recursos:

- *Devices*: uno o más dispositivos aceleradores a ser utilizados por el host.
- *Kernel objects*: las funciones OpenCL, con sus respectivos argumentos, que correrán en los devices.
- *Program objects*: la fuente o el ejecutable del programa OpenCL que implementa los kernels.
- *Memory objects*: el conjunto de objetos de memoria visibles al host y a los devices. Los objetos de memoria contienen valores y datos que son operados por las instancias los kernels.

El programa del host crea y maneja el contexto utilizando la API de OpenCL. Estas funciones le permiten al host interactuar con un device enviando comando a través de las colas de comandos, o *command-queues*. Existen tres tipos básicos de comandos:

- Kernel-enqueue commands: inician la ejecución de un kernel en un dispositivo.
- Memory commands: para transferir datos entre la memoria del host y la de los devices, o para mapear memoria compartida.
- Synchronization commands: facilitan la sincronización, para establecer restricciones particulares en el orden de ejecución de comandos.

La siguiente figura representa la comunicación entre el host y los devices, las cola de comandos, y el contexto, y donde queda bien expuesto que cada dispositivo posee su propia cola de comandos, ya que una cola no debe asociarse a más de un dispositivo.

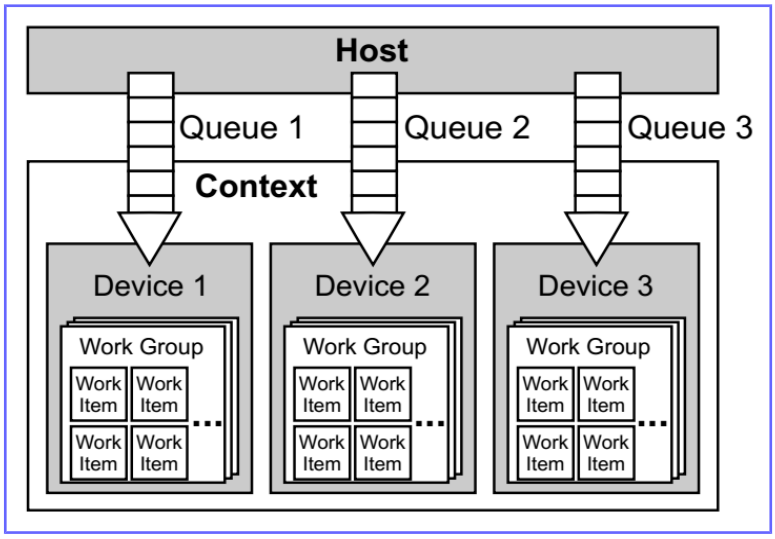

Ilustración 8: Modelo de ejecución: relación entre host, colas de comandos, contextos y devices.

El modelo de ejecución de OpenCL admite dos tipos de procesamiento en paralelo: la paralelización de las tareas (*task parallel*) y la paralelización de los datos (*data parallel*), como también varias combinaciones de ellas. Un programa de paralelización de tareas se divide en tareas individuales que pueden ser enviadas a diferentes aceleradores para su ejecución. Mientras que un programa de paralelización de datos es separado en work-groups y work-items, para ser ejecutados en un mismo dispositivo, tal como se explicará a continuación.

Cuando un kernel es puesto en ejecución por el host en un sólo device, un espacio de índices es definido, el cual puede ser también denominado como *espacio de ejecución*. Una instancia de un kernel es ejecutada entonces por cada punto en el espacio de índices, y se denomina **work-item**. Cada uno estos se identifica por su ubicación en el espacio de ejecución, el cual le provee un **global ID** a cada work-item. Cada work-item del llamado a ejecución del kernel ejecuta el mismo código, es decir, que posee el mismo comportamiento (definido por el código fuente .cl), con la diferencia que los datos computados pueden variar para cada uno, según cómo se haya definido en la descripción del kernel (en base a sus IDs en el espacio de índices).

A su vez, los work-items se organizan en **work-groups**, con lo que se obtiene una descomposición de grano grueso del espacio de índices. Al igual que los work-items, cada work-group posee su número único de identificación, es decir, su work-group ID, lo que otorga una mayor flexibilidad para describir direccionamientos e identificar de elementos puestos en ejecución. Con esto, cada work-item puede ser identificado de dos maneras: una por su índice global, y otra, por su índice de grupo más su índice local dentro del grupo.

Este espacio de ejecución es denominado como NDRange, pudiendo ser un espacio definido en una, dos o tres dimensiones (1D, 2D o 3D). Esto es que, el llamado a ejecución de los kernels puede indexarse a través de índices en hasta tres dimensiones, por lo que se pueden crear desde arreglos lineales hasta matrices de ejecución de los kernels, por decirlo de alguna manera. La siguiente figura representa la ejecución de un kernel en un device sobre un espacio de ejecución de una dimensión, con lo que los work-items y los work-groups se indexan sobre una sola dimensión o coordenada, la dimensión '0', o coordenada x para mantener la nomenclatura de los sistemas de ejes cartesianos.

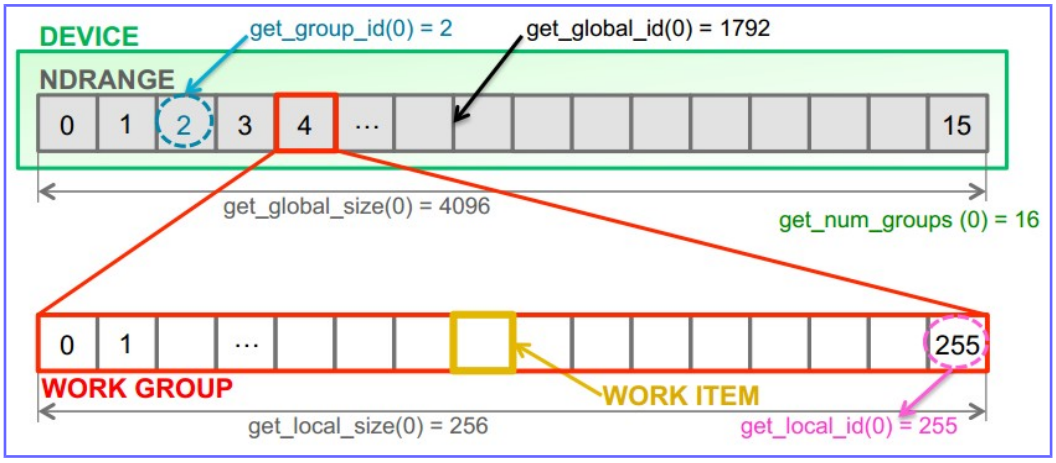

Ilustración 9: Espacio de ejecución de 1 dimensión.

La figura también ilustra las funciones de OpenCL por las cuales, desde la descripción del comportamiento del kernel, se puede acceder a la información relativa a los índices de los work-items y los work-groups.

Otro ejemplo, pero de un espacio de ejecución de dos dimensiones se muestra en la siguiente figura, donde también se ilustran las funciones por las cuales los kernels pueden obtener los parámetros de ejecución relativos a su índice.

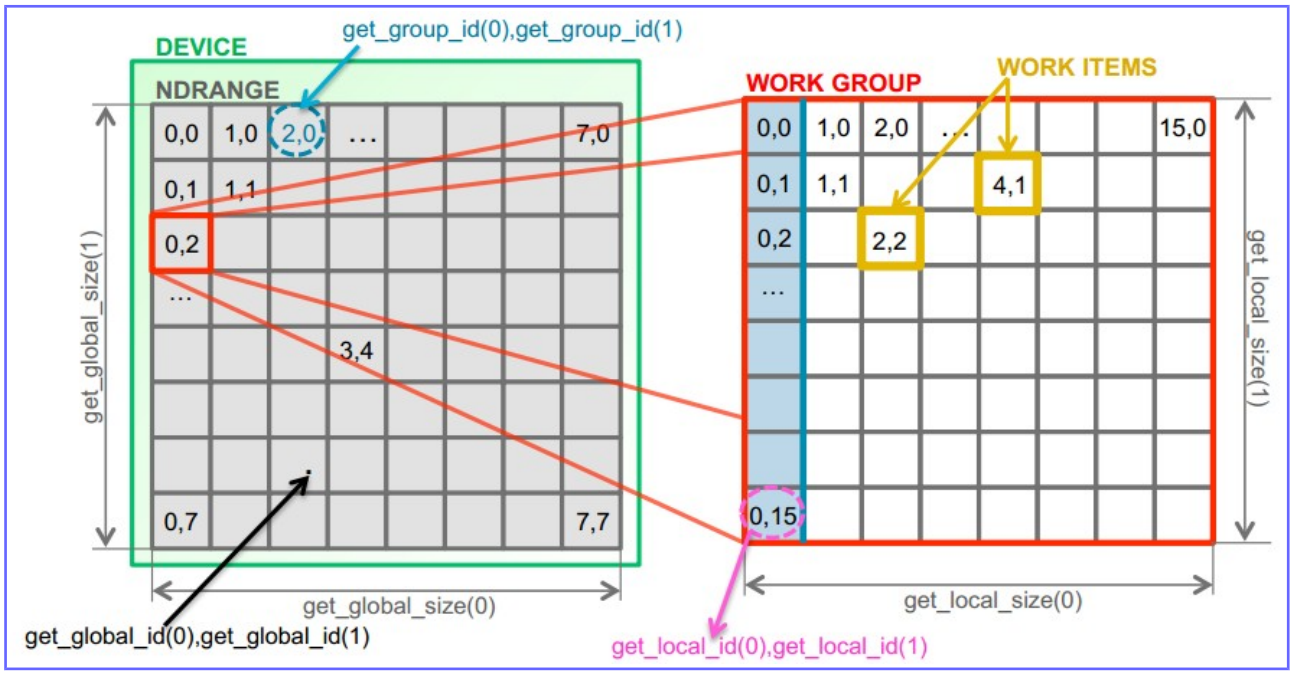

Ilustración 10: Ejecución

Mapeando el modelo de ejecución de un kernel sobre el modelo de plataforma antes presentado, se llega a

que todos los work-items en un dado work-group se ejecutan de forma concurrente en una unidad de cómputo, mientras que cada work-item es ejecutado por un elemento de procesamiento (*processing element*) o elemento de cómputo (*compute element*), tal como se ilustra en la siguiente figura.

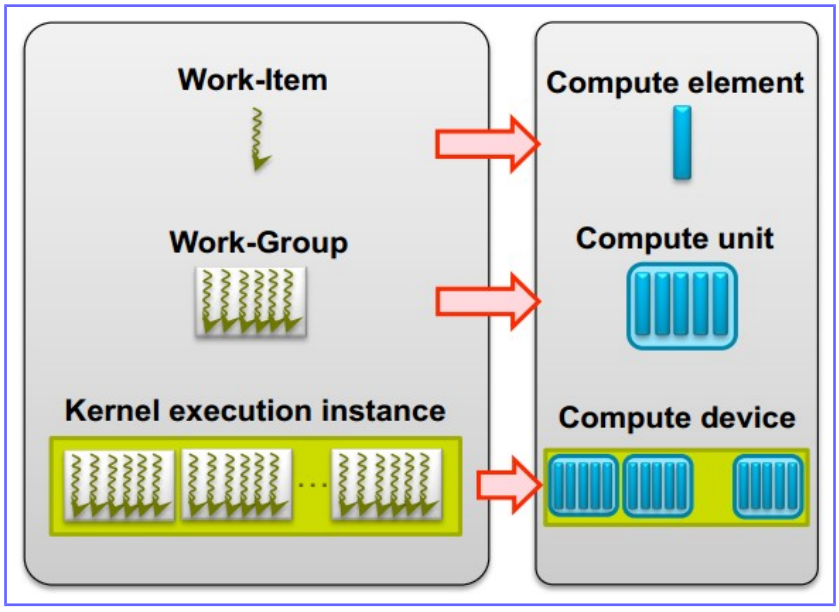

Ilustración 11: Mapeo del modelo de ejecución de un kernel sobre el modelo de plataforma.

Además, para dar una mejor idea de cómo se pueden distribuir las ejecuciones de los kernels en las diferentes unidades de cómputo, se presenta la siguiente figura, en donde se ejecutan ocho instancias de un mismo kernel en dos dispositivos diferentes, uno con dos unidades de cómputo, y el otro con cuatro.

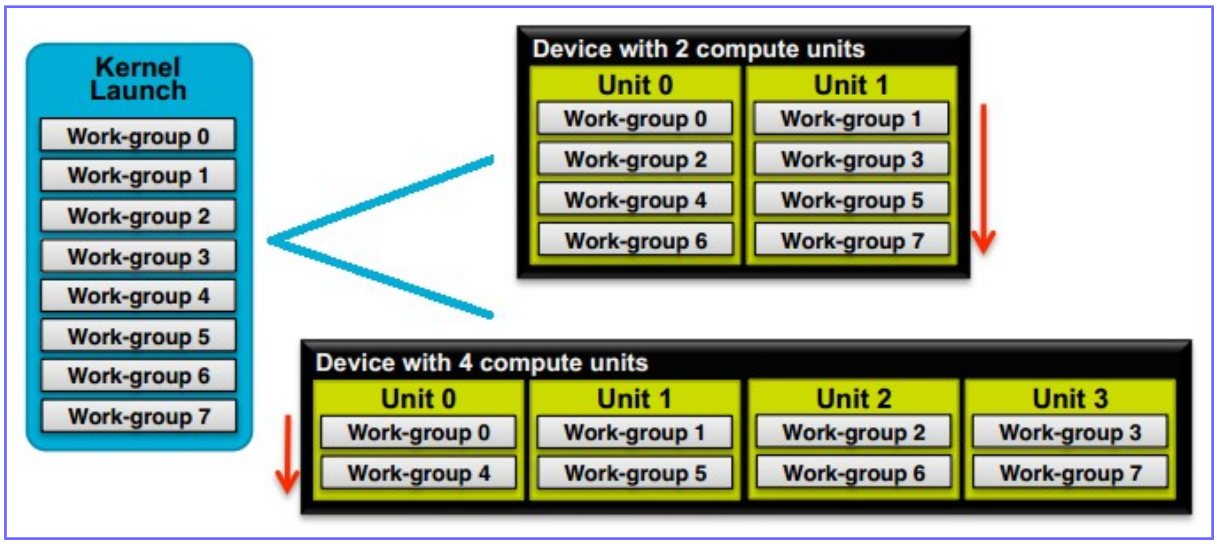

Ilustración 12: Distribución de la ejecución de un kernel sobre dos devices con un número diferente de unidades de cómputo.

El número de work-groups y de work-items por grupo, se define durante la ejecución del programa de host de manera dinámica, según los parámetros con que se llama a la función que pone en ejecución a los kernels. Por su lado, el número de unidades de cómputo, queda definido en la descripción de los kernels en su archivo fuente y posteriormente en el objeto de programa compilado, por lo que su número no puede variar de la misma manera, y es fijo en el programa objeto con los kernels compilados.

En la figura anterior, también se puede percibir las posibilidades de OpenCL, que permite el paralelismo tanto en tiempo (diferentes instancias de ejecución sobre una misma unidad de cómputo) como en hardware (diferentes unidades de cómputo).

Modelo de memoria

El modelo de memoria de OpenCL describe la estructura, el contenido, y el comportamiento de la memoria disponible en una plataforma OpenCL. La memoria del sistema se divide en dos partes:

- Memoria del host: es la memoria estándar del host, a la que el mismo accede de forma tradicional. El host puede mover contenido de esta memoria desde y hacia la memoria de los dispositivos por medio de las funciones de la API de OpenCL; o también, cuando la plataforma lo permite, puede compartir una parte de esta memoria con el o los dispositivos.
- Memoria del dispositivo: es la memoria a la que pueden acceder los kernels en ejecución en el device.

Además, el modelo de memoria define cuatro diferentes tipos de memoria del dispositivo:

- La *memoria global*, es la memoria visible a todos los work-items de cualquier instancia de ejecución de uno o más kernels en el dispositivo. Los work-items puede leer o escribir en cualquier posición de este espacio de memoria. Es la memoria de mayor tamaño, y también la más lenta.
- La *memoria constante*, es aquella memoria global que permanece inalterada durante la ejecución de un kernel, es decir, que es una memoria de sólo lectura desde la perspectiva de los work-items. Por otro lado, es el host quien escribe sobre esta memoria para que funcione como entrada de datos en la ejecución de los kernels.
- La *memoria local*, es aquella memoria compartida por todos los work-items en un mismo work-group.
- La *memoria privada*, es una memoria disponible sólo para un work-item, ya que su contenido sólo puede ser accedido por el mismo work-item en ejecución, mientras que no es visible para los demás work-items.

La siguiente figura ilustra los tipos de memoria, su jerarquía y la forma en que interactúan con cada elemento del modelo de plataforma.

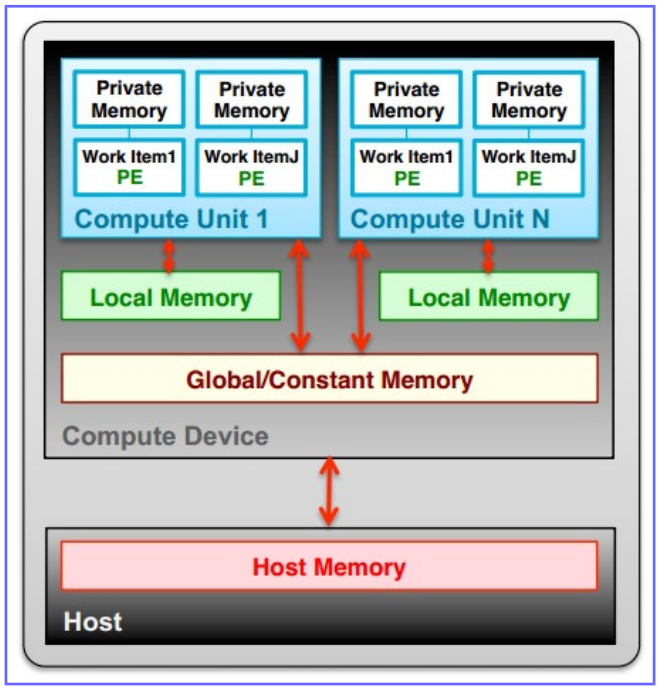

Ilustración 13: Modelo de memoria de OpenCL.

Modelo de programación

Como ya se había adelantado antes en la descripción del modelo de ejecución, el modelo de programación de OpenCL soporta los dos tipos de procesamiento en paralelo: la paralelización de las tareas (*task parallel*) y la paralelización de los datos (*data parallel*). Sin embargo, es este último tipo, el modelo principal que impulsa el diseño de OpenCL.

El modelo de programación de datos en paralelo, del inglés *data parallel programing model*, define los cómputos en términos de una secuencia de instrucciones aplicadas a múltiples elementos de un objeto de memoria. El espacio de índices asociado con el modelo de ejecución de OpenCL define los work-items y cómo los datos se mapean en cada uno de ellos. Estrictamente, en un modelo de programación de datos en paralelo, hay un mapeo uno a uno entre los work-items y los elementos en un objeto de memoria sobre los cuales un kernel puede ser ejecutado en paralelo. Sin embargo, OpenCL implementa una versión relajada del modelo, donde no es un requerimiento un mapeo estricto de uno a uno.

Por otro lado, el modelo de programación de tareas en paralelo de OpenCL define un modelo en el cual una única instancia de un kernel es ejecutada de forma independiente de cualquier índice de espacios. Lógicamente es equivalente a ejecutar un kernel en una unidad de cómputo con un sólo work-group que a su vez contiene sólo un work-item. Aunque el paralelismo no es evidente, el mismo puede estar expresado por:

- El uso de datos de tipo vector,
- La puesta en ejecución de múltiples tareas por medio de una misma cola, y/o
- La ejecución kernels nativos, desarrollados por un modelo de programación ortogonal a OpenCL.

Por último, vale la pena señalar que, el modelo de programación de OpenCL posee dos dominios para sincronizar la ejecución de los kernels:

- Sincronización de todos los work-items en un work-group, y
- Por comandos enviados a la cola de comandos en un contexto (esto es desde el lado del host).

OpenCL y FPGAs

Como ya se anticipó anteriormente, las dos empresas más importantes de FPGAs poseen su conjunto de herramientas para integrar el desarrollo con OpenCL sobre sus dispositivos, y por ende facilitan el diseño de aceleradores basados en FPGAs [16][15]. Particularmente, el desarrollo de este trabajo fue realizado con la herramienta de Altera, denominada *Altera Software Development Kit* (SDK) *for OpenCL* (AOCL). La herramienta como tal, provee un compilador y un conjunto de utilidades que permiten construir, depurar, perfilar y poner en marcha aplicaciones OpenCL utilizando FPGAs de la marca.

Otra herramienta de Altera para OpenCL es la denominada *Altera Runtime Enviroment* (RTE) *for OpenCL*, que se trata del conjunto de utilidades requeridas para que el host pueda programar la FPGA y hacer uso de la misma como acelerador.

Un aspecto importante a destacar es que existen dos tipos de plataformas aceleradoras basadas en FPGA, aquellas en las que el procesador host se encuentra embebido en el mismo dispositivo que la FPGA, y aquellas en las que el procesador host es externo al dispositivo, donde la FPGA se encuentra ubicada en una placa aceleradora y la comunicación se produce por medios externos, siendo los más comunes el PCIe y los puertos Ethernet. La siguiente figura ilustra estas dos distribuciones de las FPGAs como dispositivos aceleradores de un host.

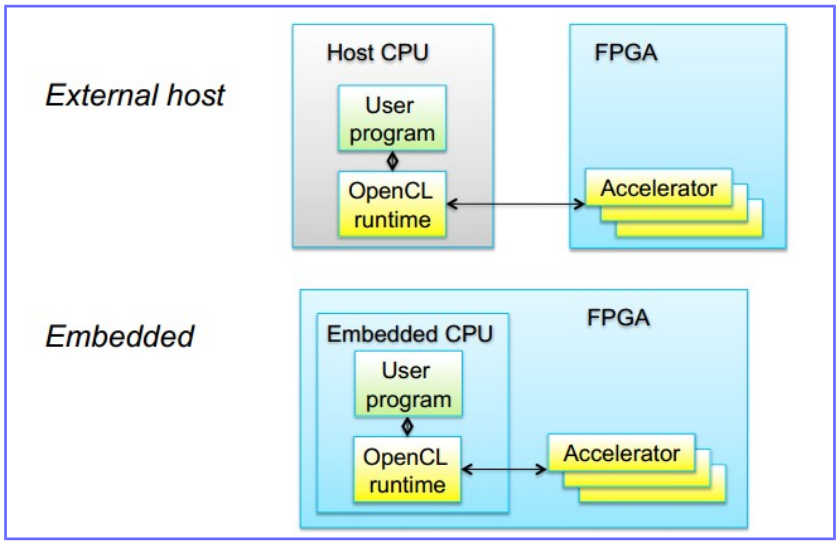

Ilustración 14: Las dos posibles disposiciones de las FPGA como dispositivos aceleradores de un host.

Por otro lado, el fabricante de la plataforma aceleradora pone a disposición el *Board Support Package* (BSP) para OpenCL, a fin de que el host pueda comunicarse con la FPGA, sea cual sea la implementación. Los BSPs son diseños de referencia que incluyen archivos de descripción de la plataforma, librerías, drivers, IP cores, interfaces IO en HDLs, APIs, descripción de la configuración de los pines, y demás archivos que forman parte de la capa más baja de diseño. Por un lado, estos archivos son utilizados por el compilador de OpenCL para compilar los kernels sobre un diseño base que se ajusta adecuadamente al dispositivo, haciendo un correcto uso del hardware disponible en el dispositivo y en la plataforma: las memorias RAM externas (habitualmente DDR), los buses de comunicación con el host, buses de comunicación con otros periféricos como puertos Ethernet o convertidores AD y DA, etc. Por su lado, el usuario se sirve de estos archivos para desarrollar sus aplicaciones sobre un diseño base que reduce el tiempo de desarrollo y lo libera de desarrollar tareas de bajo valor añadido.

La utilización de OpenCL en el diseño con FPGAs posee muchas ventajas, alguna de las cuales son:

- Frente al método tradicional de diseño con *Lenguajes de Descripción de Hardware* (HDLs), se obtienen tiempos de desarrollo mucho más ágiles, a la vez que se facilita el desarrollo de diseños complejos.
- Permite migrar con cierta facilidad códigos descritos en lenguaje C a una implementación hardware sobre FPGAs.
- Permite una depuración (debug) relativamente simple durante el desarrollo de los kernels.
- Abstrae al diseñador de los detalles de la lógica de control y de los buses de comunicaciones.

Flujo de diseño de OpenCL con el SDK de Altera

El siguiente diagrama presenta el modelo de programación del SDK de Altera para OpenCL, para llevar a cabo un diseño sobre una plataforma basada en una FPGA de la marca. El proceso cuenta con dos caminos bien definidos para el usuario estándar: el primero, que cuenta con la descripción y desarrollo del o los kernels OpenCL y su compilación con el compilador de Altera, el *Altera Offline Compiler* (AOC); y el segundo, que involucra el desarrollo de la aplicación para el host, en la que se realiza la compilación del programa, generalmente descrito en lenguaje C/C++, y que es realizada por el compilador de C/C++ habitual.

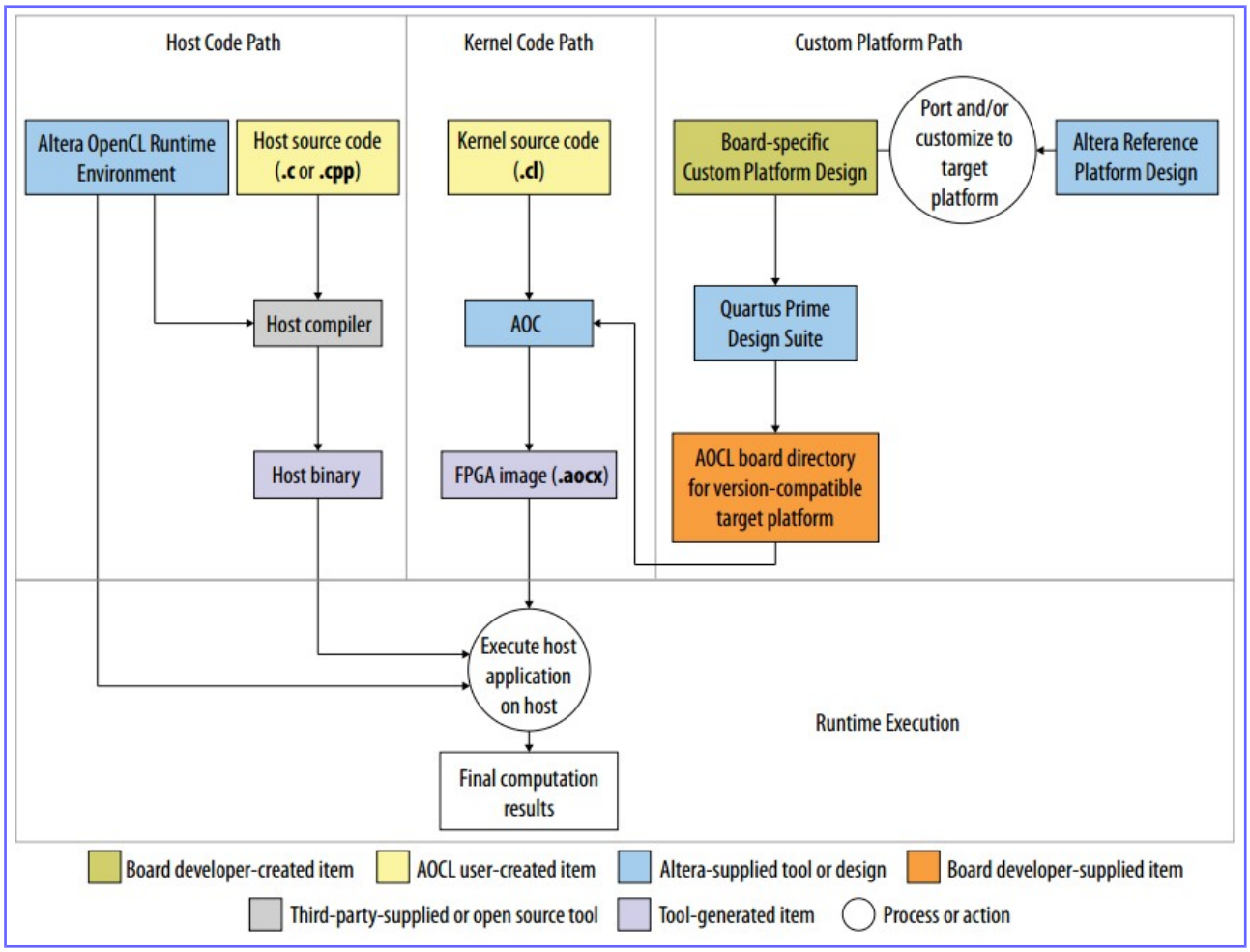

Ilustración 15: Flujo de diseño con el SDK de Altera para OpenCL.

El tercer camino del diagrama, involucra el desarrollo del BSP y la demás información vinculada a la plataforma. En la mayor parte de los diseños, es el fabricante quien realiza este proceso y pone a disposición para el usuario todos los archivos necesarios que permiten trabajar con su plataforma. Aún así, esto no excluye que este proceso pueda ser llevado a cabo por el usuario, partiendo del conocimiento del dispositivo

FPGA objetivo y la placa sobre la que esté montado, ya sea que se trate de una mejora del BSP proporcionado por el fabricante, o por el desarrollo de una plataforma propia.

Flujo de compilación de kernels con el compilador de Altera para OpenCL

El compilador de Altera, AOC, crea un archivo de configuración hardware de la FPGA (.aocx) a partir del o de los archivos fuentes que describen los kernels (.cl), o bien a partir de archivos AOC objeto temporales (.aoco) que han sido pre-compilados incluyendo uno o más kernels. Estos son los tres tipos de archivos que se manejan a nivel de compilación de los kernels por parte del AOC, y se describen mejor a continuación:

- *OpenCL Kernel Source File* (.cl): se trata de los archivos fuentes descritos en el lenguaje OpenCL C, y que describen el comportamiento de los kernels. El compilador puede servirse de uno o más archivos de este tipo, a la vez que cada uno puede contener la descripción de uno o más kernels.
- *Altera Offline Compiler Object file* (.aoco): se trata de un archivo objeto que contiene información sobre los kernels que se utilizan en etapas posteriores de la compilación.
- *Altera Offlilne Compiler Executable file* (.aocx): es el archivo de configuración hardware de la FPGA, y contiene la información que utiliza el host para programar la FPGA.

La siguiente imagen ilustra el proceso de compilación de uno o más kernels descritos en un archivo fuente. Es importante señalar, que el proceso de compilación de kernels OpenCL para FPGA se realiza de forma *offline*, esto es, antes de que la aplicación se ponga en funcionamiento en el host.

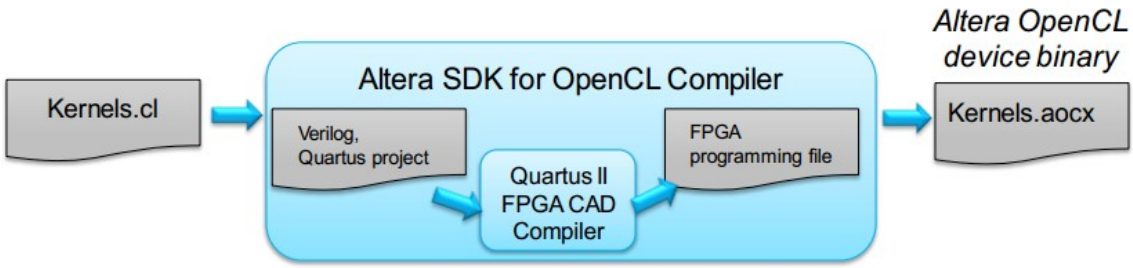

Ilustración 16: Flojo de compilación de un kernel con el compilador OpenCL de Altera.

En principio, para el usuario, la compilación se realiza en un sólo paso. Sin embargo, internamente, el compilador AOC ejecuta tres procesos que resultan transparentes para el usuario: la traducción del o los kernels a descripciones HDLs, la compilación del diseño con Quartus, y la creación del archivo de programación en el formato .aocx.

En el momento de la compilación, el usuario puede darle instrucciones al compilador sobre optimizaciones o preferencias en la compilación, tal como realizar un mayor esfuerzo en el emplazamiento

del diseño en la FPGA, configurar el tamaño de las memorias cache para constantes, utilizar implementaciones de árboles balanceados que reducen los requerimientos de hardware en operaciones con punto flotante, reducir las operaciones de redondeo en punto flotante, entre otras.

Plataforma de desarrollo: DE1-SoC Board

La plataforma de desarrollo utilizada es la placa DE1-SoC Board de la empresa Terasic. Ésta es una plataforma de desarrollo multipropósito, que posee una FPGA Cyclone V, y diversos periféricos que van desde simples interfaces para la interacción con el usuario como switches, pulsadores, LEDs y displays de 7 segmentos, hasta un CODEC de audio, un decodificador de video analógico, un convertidor DAC para VGA, un integrado de la capa física de Ethernet, controladores de memoria micro-SD, sólo por mencionar algunos.

La siguiente imagen ilustra el layout de la plataforma, señalando los recursos y periféricos disponibles.

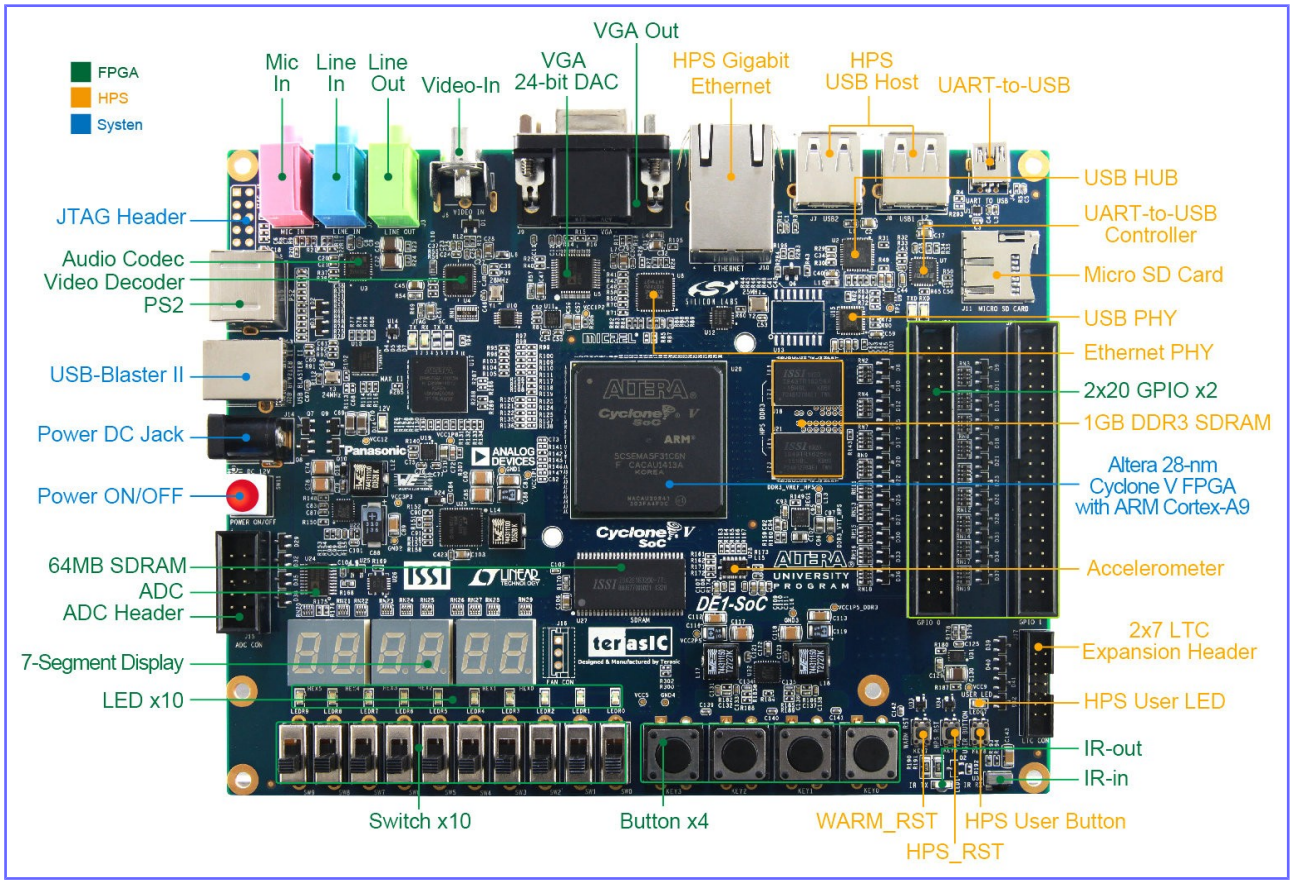

Ilustración 17: DE1-SoC Board Layout.

Por su lado, el dispositivo Cyclone V cuenta con:

- Un procesador embebido en silicio con arquitectura ARM Cortex-A9 de dos núcleos, capaz de funcionar hasta una frecuencia de 800 MHz, denominado por Altera como *Hard Processor System* (HPS).
- Una parte de lógica reconfigurable, la parte propiamente FPGA, con un equivalente de 85k *logic elements* y 4450 Kbytes de memoria embebida; y
- Elementos de interconexión interna entre el HPS y la FPGA.

La siguiente imagen es genérica de toda la familia de dispositivos del tipo SoC de Altera, sin embargo representa muy bien la estructura interna de la Cyclone V y la forma en que el HPS se comunica con la FPGA.

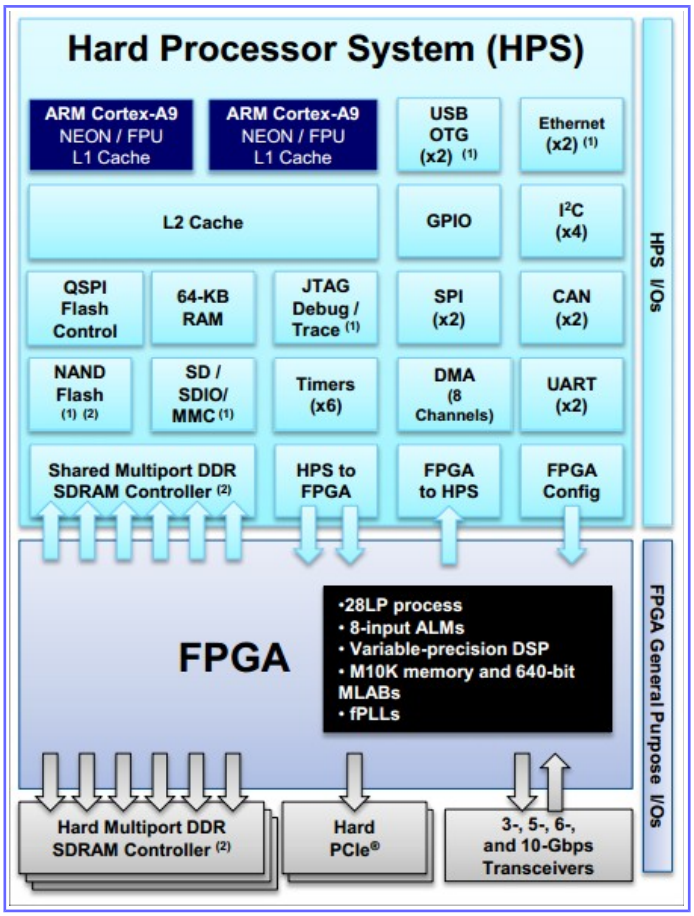

Ilustración 18: Estructura interna de la familia de dispositivos FPGA SoC de Altera.

La elección de esta plataforma de desarrollo quedó definida por varios motivos, entre los que se destacan: las capacidades e integración de la plataforma para el desarrollo de OpenCL con FPGAs; y, sus capacidades de conectividad en red (puerto Ethernet) junto a estar basada en un dipositivo del tipo FPGA-SoC, permite

que sea utilizada como punto de cómputo integrándose así en la investigación del grupo de I3M.

Por otro lado, dado que el desarrollo del trabajo se basó en una aplicación .C corriendo sobre un sistema operativo Linux, y el diseño de la FPGA se realizó por medio de OpenCL, no viene al caso seguir detallando las características de la plataforma, dado que no resultan de mayor relevancia en cuanto al desarrollo del presente trabajo se refiere. Sin embargo, si es importante señalar que el Linux utilizado es proporcionado por la empresa Terasic, tratándose de una versión construida con las herramientas del Proyecto Yocto sobre la que se instalaron las librerías necesarias para utilizar OpenCL (RTL de Altera para OpenCL).

Para más información y acceso a los recursos disponibles se puede acceder a [18].

Justificación

Cuando se comienza a trabajar con RNAs en un problema en concreto, se ha observado que redes muy grandes habitualmente tienden a sobre-ajustar la información de entrenamiento, afectando su capacidad de generalización; mientras que redes muy pequeñas usualmente encuentran problemas en aprender de las muestras de entrenamiento debido a su limitada capacidad de representación. Además, cuando se consiguen buenos resultados, siempre existen dudas de si estos resultados son óptimos para todas las estructuras de redes que podrían darse.

Esta incerteza puede ser aceptable por algunos investigadores si otras condiciones, cuando trabajan con RNAs, están establecidas o acotadas. Pero cuando todos los parámetros necesitan ser evaluados como parte de la misma investigación, la determinación de la topología óptima requiere un largo periodo de experimentación. Curteanu y Cartwright listaron los principales métodos para determinar la topología óptima de redes del tipo MLP: prueba y error, métodos empíricos o estadísticos, métodos híbridos, algoritmos constructivos y destructivos, y estrategias evolutivas [19].

En el diseño de estructuras, los *Algoritmos Evolutivos*, AE o EAs del inglés *Evolutionary Algorithms*, son utilizados de dos maneras: para evolucionar sólo la estructura [20] y para evolucionar tanto la estructura como los pesos de las conexiones de forma simultánea [21]. Sin embargo, el uso de EAs para evolucionar de forma simultánea la estructura de la red y los pesos de las conexiones, enfrenta una mayor dificultad: la habilidad de una RNA para evolucionar en una red superior se basa en la supervivencia de la topología correcta.

Por otro lado, otra aproximación es evolucionar sólo las estructuras, sin ningún peso de conexión. Pero el uso de EAs para evolucionar sólo la estructura, enfrenta el problema de evaluaciones de ajuste ruidosas [22]

[23]. Los pesos de las conexiones deben ser aprendidos luego de que se ha encontrado una arquitectura casi óptima; siendo que, diferentes incializaciones aleatorias de los pesos en el entrenamiento de la red, puede producir resultados de entrenamiento bastante diferentes. Este mapeo de una estructura a muchas redes entrenadas puede introducir evaluaciones de funciones de ajuste ruidosas y por ende evoluciones engañosas. Por ello, a fin de reducir el ruido, habitualmente una misma arquitectura debe ser entrenada muchas veces, usando diferentes inicializaciones de pesos. Como contrapartida, este método introduce un gran incremento en la carga computacional, siendo este el principal inconveniente de utilizar EAs de esta forma, y es por lo tanto la principal justificación de la investigación que dirige Rafael Gadea en el área de *Diseño de Sistemas Digitales* del *Instituto de Instrumentación para Imagen Molecular,* I3M*,* siendo que su implementación permite distribuir la carga computacional a través de una red heterogénea [1].

Además, el trabajo del grupo no pierde de vista que la optimización de una RNA no sólo es conseguir una buena aproximación de desempeño con la información de entrenamiento, sino que también con la información desconocida para el mismo problema, consiguiendo la característica de generalización que deben poseer las RNA, formando así el segundo objetivo de su estudio.

Él método implementado por el grupo consiste en cinco fases:

- *Fase 1:* Selección de las mejores entradas para formar el conjunto de entrenamiento por promedios de una computación basada en el Delta Test [24].
- *Fase 2:* Filtrado de las muestras a través de un replicador de redes neuronales [25].
- *Fase 3:* Optimización de la topología de las redes neuronales por promedios de una computación evolutiva heterogénea [2].
- *Fase 4:* Optimización de los pesos iniciales de las redes neuronales por promedios de una computación evolutiva.
- *Fase 5:* Entrenamiento final de la red con la topología obtenida en la fase 3 y con los pesos inciales obtenidos en la fase 4.

Hay que aclarar, que existen ciertas condiciones de trabajo sobre las que se desarrollan los estudios del grupo, tales como las características de las muestras; a la vez que se establecen ciertas consideraciones como la división de las muestras de entrada obtenidas en la fase 2, en cuatro sub-conjuntos: entrenamiento, validación, optimización y verificación; entre otras [1].

Sin embargo, existe un problema notable con la implementación de este método, y es la gran carga computacional que se demanda. Además, se debe seleccionar un método por el cual se seleccionen los mejores individuos cuando se promedian diferentes inicialzaciones, de modo que los mejores individuos sean

los que sobreviven en el algoritmo genético y no se descartan debido a malas elecciones en las función de ajuste. Para esta solución se propusieron varias alternativas, que han cubierto diferentes resultados: la implementación GATOPOMIN, la GATOPOMEAN, la GATOPOGA, la GATOPODE y la GATOPODNN. En resumen, estas difieren en la selección de la función objetivo del algoritmo evolutivo que evoluciona los diferentes individuos con una misma topología, pero con distintas inicializaciones de pesos. Estas implementaciones se han realizado entrenando los MLP con el *Resilent Backpropagation Algorithm* (RBP), y calculando su función de desempeño con la red entrenada con este algoritmo.

Es aquí donde nace la idea de utilizar, una vez más, un algoritmo genético como herramienta en este método de búsqueda de la mejor topología de una red. En particular, y cómo ya se ha anticipado, la idea se basa en utilizar el algoritmo de Evolución Diferencial, como algoritmo de evolución de los pesos de un MLP, con una estructura de neuronas previamente definida por el algoritmo genético principal. Esto es, dejar de lado el RBP como algoritmo de entrenamiento de los MLP, y en su lugar, utilizar el DE para obtener los mejores valores de la función de desempeño de cada estructura.

Objetivos

Dado el marco teórico y entorno del estudio, el principal objetivo propuesto para este trabajo fue implementar el entrenamiento de una red neuronal de tipo Perceptrón Multicapa, utilizando el algoritmo de Evolución Diferencial para evolucionar los pesos de un conjunto de redes con la misma estructura. Así, con ello, obtener un valor de desempeño con el que el algoritmo evolutivo principal, que evoluciona las estructuras, pueda evaluar a cada una de ellas. Además, la implementación debía llevarse a cabo sobre plataformas basadas en FPGA y que soporten conexión en red Ethernet, a fin de incorporarlas en el entorno de la investigación realizada por el grupo del área de *Diseño de Sistemas Digitales*, del instituto de investigación I3M, sobre computación distribuida basada en dispositivos FPGAs. Como parte del objetivo principal, también se definió acelerar los cómputos involucrados en el algoritmo de Evolución Diferencial y en el cálculo de las salidas de las redes (fase *forward* del MLP), por medio de kernels aceleradores de OpenCL.

Se planteó como objetivo secundario, incorporar el módulo IP que computa la tangente hiperbólica, a partir de su descripción en HDLs, como función auxiliar en la descripción de los kernels de OpenCL. La motivación de este objetivo se debe a que el cálculo de la tangente hiperbólica (*tanh*), en esta aplicación en particular, no requiere la precisión que Altera implementa en su función *tanh* de OpenCL; sino que se admite una mayor tolerancia. Por ello, la *tanh* implementada por Altera requiere una notable mayor cantidad de

recursos que los que se necesitan con la implementación en HDL propuesta, a la vez que esta alternativa presenta una menor latencia, con lo que se mejora la respuesta temporal. Entonces, introduciendo el módulo HDL como función auxiliar en los kernels OpenCL permite realizar el cómputo de la tangente hiperbólica, con la precisión requerida, y a un coste de recursos notablemente menor, lo que deja más recursos disponibles para implementar otros kernels u otras funcionalidades.

Desarrollo

DE para la optimización de los pesos de la red

Para utilizar el algoritmo de *Evolución Diferencial*, DE, como método de evolución de los pesos sinápticos de un *Perceptrón Multicapa*, MLP, se deben establecer ciertas vinculaciones entre ambos. Como ya se ha anticipado, los parámetros de entrada del DE son los parámetros del problema a resolver, esto es, los pesos del MLP. Por ende, la población de individuos del DE se formará a partir de diferentes conjuntos de pesos de la red.

Por otro lado, también se debe definir una función objetivo para evaluar el desempeño de las redes al momento de realizar la selección del vector de pesos padre ($\mathbf{x_p}$) o del vector de pesos de prueba ($\mathbf{u_p}$). Esta función objetivo fue definida como el sumatorio del cuadrado de la diferencia de las salidas producidas por la red y las salidas deseadas, según el estímulo patrón con que se excita la red. Es decir:

$$f_{obj}(\boldsymbol{x}_p) = \sum_{i=0}^{numPats} \left(output_{(i)} - outputSpected_{(i)}\right)^2$$

Siendo:

- $f_{obj}(\mathbf{x_p})$: la función objetivo evaluada en un conjunto de pesos que definen la red que se está evaluando.
- NumPats: el número total de patrones disponibles en el conjunto de entrenamiento.
- Output: la salida que genera la red para el estímulo patrón *i*.
- outputSpected: la salida que se espera que la red genere frente al estímulo patrón *i*.Forma parte de conjunto de datos de entrenamiento.

Esto es que, una dada red con su conjunto de pesos (sean $\mathbf{x_p}$ o $\mathbf{u_p}$), su función objetivo se calcula como la suma del cuadrado de los errores de salida para cada uno de los patrones del conjunto de entrenamiento.

Entonces, habiendo definido los parámetros de entrada o a evolucionar con el DE, los pesos de la red; y habiendo definido la función objetivo con la que se evalúa el desempeño de una red; se establece la vinculación entre los MLP y su método de aprendizaje.

Vale la pena señalar, que es común que la suma de los errores se normalice según la cantidad de estímulos patrones, y que finalmente se calcule su raíz cuadrada, esto es, calcular el valor RMS del error generado por todos los estímulos. Sin embargo, estos cálculos no marcan una diferencia en el algoritmo DE, dado que la función objetivo sólo se utiliza a modo comparación para seleccionar el mejor individuo entre dos. La normalización y la raíz cuadrada son funciones monótonamente crecientes, por lo que su aplicación no cambia el resultado de la comparación realizada en el proceso de selección del DE, y computacionalmente demanda ya sea recursos hardware o temporales, que en un proceso de optimización temporal, resultan ser muy valiosos.

Con ello, el flujo del algoritmo DE queda definido como:

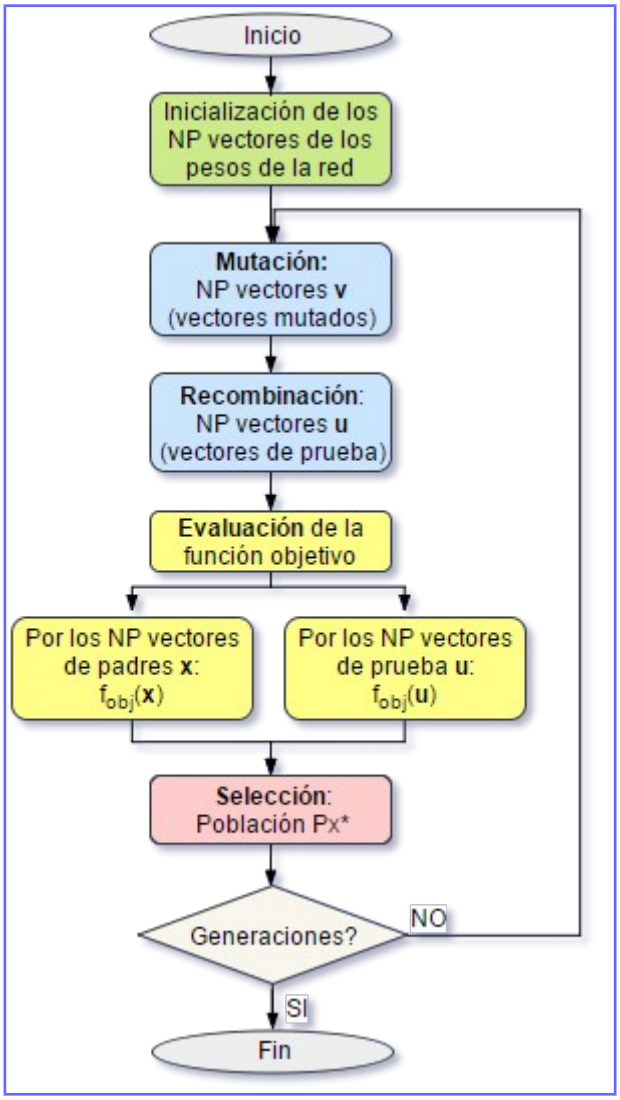

Ilustración 19: Diagrama de flujo del algoritmo DE como método de entrenamiento de un MLP.

Indudablemente, del diagrama se desprende una dependencia secuencial en cuanto a la ejecución de cada

una de las etapas del algoritmo se refiere. Sin embargo, como se analizará a continuación, el procesamiento elaborado en cada etapa es independiente para cada individuo, lo que invita a pensar en la utilización de paralelismo.

Aceleración del algoritmo de DE con OpenCL

Como se acaba de introducir, el procesamiento elaborado en cada una de las etapas del algoritmo DE (Mutación, Recombinación, Evaluación y Selección) es independiente para cada uno de los individuos. Dicho de otro modo, tanto la etapa de mutación, la de recombinación, la de evaluación y la de selección, que se realizan para cada individuo de la población dentro de una misma generación, son independientes de los resultados que se computan para los demás individuos. Esto se ilustra muy bien en el siguiente diagrama, que describe las etapas del algoritmo de forma gráfica, aplicada a los individuos de la población de vectores padre.

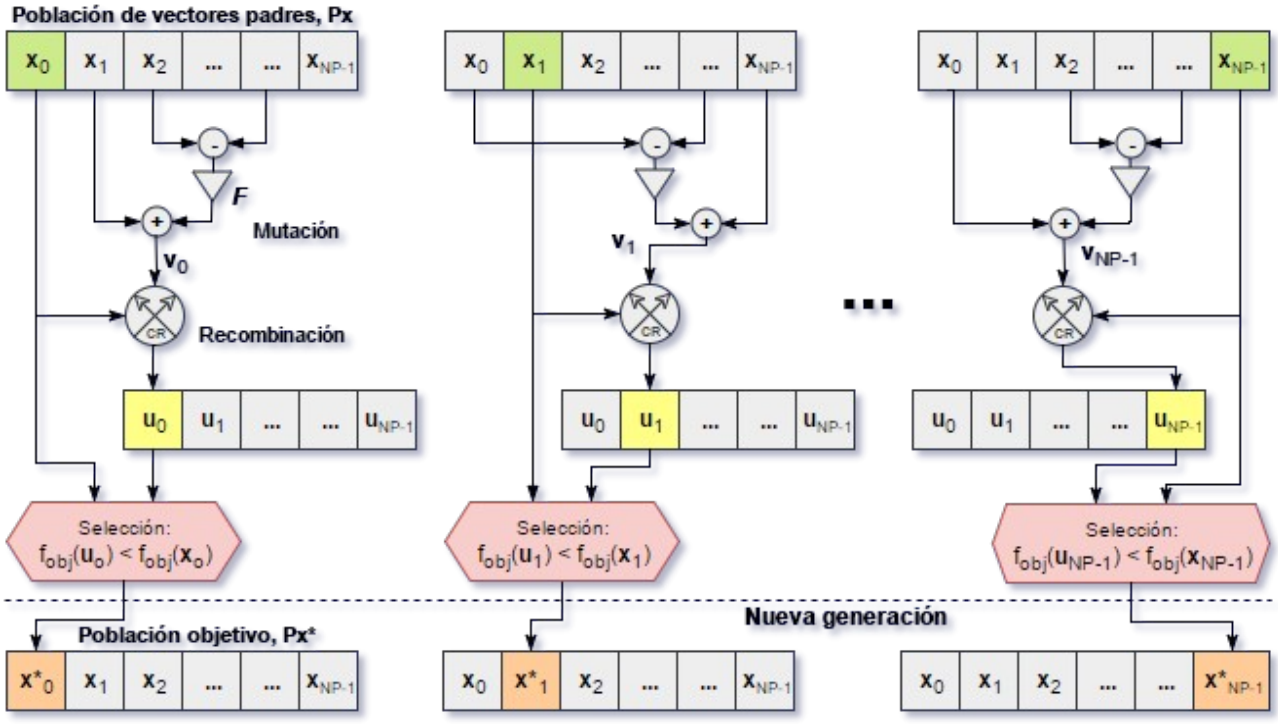

Ilustración 20: Flujo de datos del algoritmo DE.

Entonces, es intuitivo pensar que la computación de estas etapas puede paralelizarse, existiendo dos formas principales:

- Paralelización de toda la estructura (la presentada en el diagrama) en forma de vectorización o

pipeline, es decir, introduciendo sucesivamente el cómputo de nuevos individuos en una misma estructura hardware, a medida que los cálculos de individuos anteriores van siendo computados en etapas más avanzadas.

- Paralelización de cada una de las etapas de forma individual, para todos los individuos. Esto es que, la etapa de mutación se paraleliza para todos los individuos en una sola ejecución, luego se computa la etapa de recombinación en paralelo para todos los individos; luego la evaluación y finalmente la selección.

Es este último modelo es que se eligió para realizar la implementación de los aceleradores en OpenCL, en donde se desarrollaron dos kernels en lugar de tres, para computar las etapas de mutación, recombinación y selección. Por otro lado, se verá más adelante que el cómputo de la evaluación de la función objetivo se aceleró con la utilización de otros kernels OpenCL.

El primer kernel en cuestión se dedicó a la implementación conjunta de la mutación y la recombinación. La elección de implementar estas dos etapas en un mismo kernel parte de la base de que, el vector intermedio entre estas dos etapas es el vector mutado $\mathbf{v}_p$, y sólo existe en la ejecución entre estas dos etapas. Entonces, a favor de mejorar el desempeño del acelerador, se decidió que, para cada individuo, este vector se implementase sobre memoria privada (que la FPGA implementa sobre memoria RAM distribuida), consiguiendo así eliminar accesos innecesarios a memoria externa, que además de ser más lentos, requieren tambíen de la reserva de todo un espacio de memoria, el que se hace considerable en tanto las dimensiones de las redes aumentan, al incrementar exponencialmente el número de conexiones con el aumento del número de neuronas de la red.

El kernel desarrollado se transcribe a continuación:

```
__kernel void DE_Kernel_MutAndRecom(
        // Punteros a memoria para los vectores padres y los de prueba
        __global float    * restrict px,     // Parent poblation
        __global float    * restrict pu,     // Trial  poblation


        // Punt. para la realimentacion de la semilla del generador random
        __global unsigned * restrict seed,   // Semilla para el RNDGEN


        // Parametros del algoritmo DE
        float const DE_F,       // Factor de mutacion
        float const DE_CR,      // Coeficiente de recombinacion
        int   const DE_D,       // Cant. de parametros de los vectores/individuos
        int   const DE_NP       // Cantt. de individuos del algoritmo DE
        )
{
    // Si se llaman NP Kernels, cada uno trata un vector X (individuo) de la poblacion PX
    int id = get_global_id(0);


    // Variables locales //
    __local int      i;         // Indice para recorrer los parametros de cada vector
    __local int      r0,r1,r2;  // Ind. random entre[0;NP-1] para sortear los 3 vectores de la mut.
    __local float    nRnd;      // Random Normalizado: [0;1] para compara con DE_CR
```

```
    __local unsigned uRnd;      // Random Entero
    __local unsigned iRnd;      // Random Entero

    // Variable para calcular el vector donador: ya que se hace parametro por parametro,
    // por lo que no es necesario un array completo
    __local float   v;

    // Random Init: obtiene la semilla de memoria
    uRnd = rndGenU(seed[id]);

    // 0) Seleccion de los indices de los individuos para la mutacion
    do {// 1er Random Idx
        uRnd = rndGenU(uRnd);
        nRnd = rndNorm(uRnd);
        r0 = nRnd * DE_NP;
    } while (r0 == id);
    do {// 2do Random Idx
        uRnd = rndGenU(uRnd);
        nRnd = rndNorm(uRnd);
        r1 = nRnd * DE_NP;
    } while (r1 == r0 || r1 == id);
    do {// 3er Random Idx
        uRnd = rndGenU(uRnd);
        nRnd = rndNorm(uRnd);
        r2 = nRnd * DE_NP;
    } while (r2 == r0 || r2 == r1 || r2 == id);

    // Para cada elemento del vector #id (el individuo #id de las poblaciones PX y PU)
    for(i=0;i<DE_D;i++){
        //  1) Mutacion:
        // .C: v[i] = x0[i] + DE_F*(x1[i] - x2[i]);
        v = px[r0*DE_D+i] + DE_F * (px[r1*DE_D+i] - px[r2*DE_D+i]);

        //  2) Recombinacion (Crossover):
        // Se forma el vector de prueba u, combinando los elementos del vector mutado v con los del
       // vector padre x
        uRnd = rndGenU(uRnd);
        nRnd = rndNorm(uRnd);
        iRnd = nRnd * DE_D;
        if(nRnd <= DE_CR || i == iRnd){
            // Se inserta el parametro mutado
            pu[id*DE_D + i] = v;
        }else{
            // Se mantiene el parametro padre
            pu[id*DE_D + i] = px[id*DE_D + i];
        }
    }
    // Mantengo el valor del progreso del RNDGEN para sucesivas generaciones
    seed[id] = uRnd;
}
```

El kernel en cuestión está diseñado para que sea llamado en un espacio dimensional de una sola dimensión, con un número de work-groups igual al número de individuos de la población, NP. Así, cada hilo de ejecución se encarga de computar las etapas de mutación y recombinación para cada individuo de la población.

Para su funcionamiento, cada instancia de ejecución del kernel utiliza cuatro vectores padre de la memoria global del dispositivo (accedida a través del puntero *px*):

- El primer vector se corresponde con el individuo objetivo, cuyo índice en la población Px está

dada por el índice global de la instacia de ejecución, es decir, por su global ID.

- Los otros tres vectores se corresponden con tres individuos diferentes entre sí y del vector índice, que son seleccionados aleatoriamente.

Con estos últimos tres vectores, se calcula el vector mutado v, pero parámetro a parámetro, y luego se realiza la recombinación de la misma manera, formando entonces el vector de prueba v. Éste es el vector de salida de la computación de estas dos etapas, y es escrito en memoria global del dispositivo, para estar disponible en cálculo de la función de desempeño de cada una de las redes.

Vale la pena destacar la presencia de una función auxiliar para la generación de los números aleatorios, tanto para la selección de los vectores aleatorios con que se realiza la mutación, como para obtener el valor aleatorio que se utiliza en la recombinación. Esta función basa su funcionamiento en el método de registros de desplazamiento lineal re-alimentados con salto hacia adelante, (*LFSR Leap a Head*), en particular el generador denominado LA3217, que se describe en [26]. Éste utiliza como semilla de entrada y como dato de salida un valor de 32 bits, que se implementa en OpenCL como una variable de tipo *unsigned int32*.

Dado que la función implementada se trata de una adaptación en OpenCL de un generador orientado a una implementación puramente hardware, se requirieron realizar accesos a memoria a fin de mantener el valor de los registros del LA3217 entre sucesivos llamados a la función, y por parte de diferentes instancias de ejecución del kernel, siendo que cada una debe mantener su propia secuencia pseudo-aleatoria.

Por último, existe otra función que sólo se encarga de normalizar el valor obtenido por el LA3217 a un valor de tipo flotante entre 0 y 1, que se implementa simplemente retornando el valor obtenido del generador dividido por el máximo valor entero sin signo en una representación de 32 bits.

La descripción de las dos funciones implementadas en OpenCL puede observarse en el Anexo al final de este trabajo.

Aceleración del la fase forward de la ANN con OpenCL

Para poder completar el algoritmo DE, es necesario computar la función objetivo como se ha definido: el sumatorio del cuadrado del error para cada entrada o estímulo patrón del conjunto de entrenamiento. Para ello, se requiere principalmente que se pueda calcular (en cada generación del DE) la fase forward de las 2*NP redes definidas por: los NP vectores padres y los NP vectores de prueba.

Paralelismo en el cómputo de las neuronas

Si se analiza el comportamiento del MLP, se puede encontrar que el cómputo de cada neurona en una misma capa es independiente del cómputo de las demás, esto es que, el cálculo de la salida de todas las neuronas de una misma capa puede realizarse en paralelo sin ninguna dependencia de datos del resultado del cálculo de otras neuronas. Gráficamente, esta independencia se observa al no existir flujo de datos (conexiones ilustradas como flechas) entre neuronas de la misma capa, tal como se aprecia en la siguiente figura de un MLP de dos capas ocultas y una capa de salida (figura ya presentada en la introducción de redes neuronales artificiales).

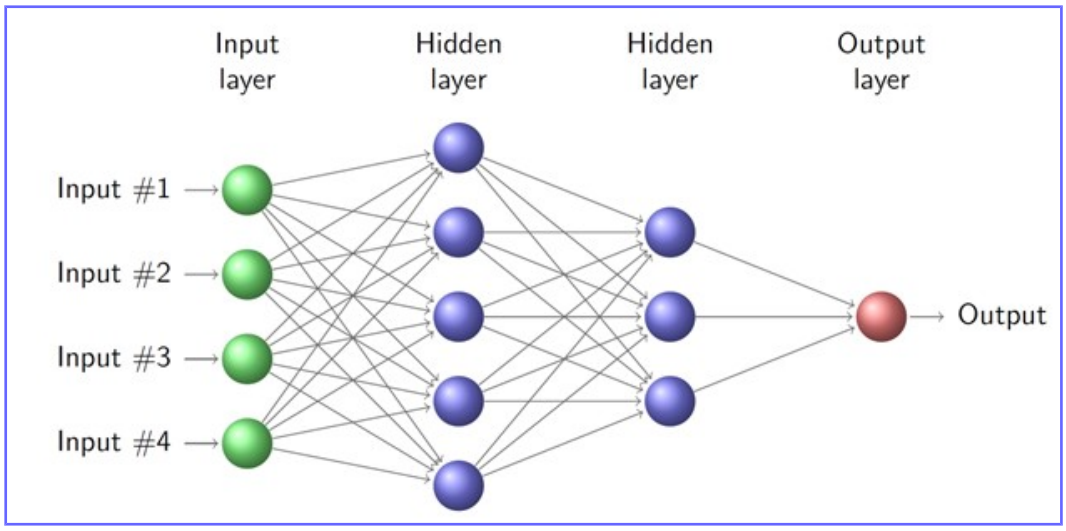

Ilustración 21: Perceptrón Multicapa de dos capas ocultas y una capa de salida. La independencia entre las neuronas de una misma capa se observa, al no existir flujos de datos (conexiones) entre dichas neuronas.

Entonces, el cálculo de la salida de todas las neuronas de una misma capa puede ser computado en paralelo sin ninguna dependencia de datos, por lo que es candidato de implementarse y acelerarse por medio de algoritmos descritos en OpenCL.

Vale la pena aclarar, que en una implementación, todos los cálculos de cada neurona acceden a los mismos espacios de memoria donde se almacenan los valores de salida de las neuronas de la capa anterior (o los valores de la capa de entrada cuando se computa el cálculo de las neuronas de la primer capa oculta). Esto puede acarrear problemas de disminución de performance, dependiendo de la implementación que se realice, por lo que conviene no pasarlo por alto.

Paralelismo en el cómputo de las diferentes redes

A su vez, el cálculo de toda la fase forward de cada red es independiente del cálculo de las demás redes,

por lo que éste es un cálculo también susceptible de ser paralelizado, ya que se pueden a calcular las 2*NP redes en paralelo sin dependencia de resultados entre ellas.

Ahora bien, considérese la nomenclatura representada en la siguiente figura para denominar al cálculo de las neuronas de una misma capa en todo el proceso de cálculo de la fase forward de una red neuronal.

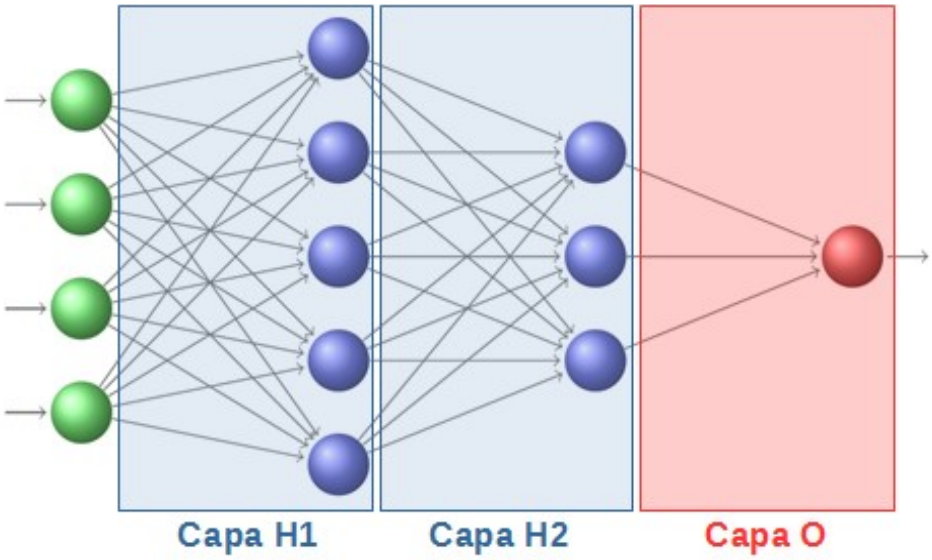

Ilustración 22: Nomenclatura en el cálculo de las neuronas en las diferentes capas de una red.

Siendo intuitivamente:

- Capa H1: el nombre que recibe el cálculo de todas las neuronas de la capa oculta 1;
- Capa H2: el nombre que recibe el cálculo de todas las neuronas de la capa oculta 2; y
- Capa O: el nombre que recibe el cálculo de todas las neuronas de la capa de salida (en este caso correspondiente a una sola neurona).

En caso de implementarse un MLP con un mayor número de capas ocultas, los cálculos de las sub-siguientes capas ocultas se denominarían Capa H3, Capa H4, etc.

Entonces, se puede percibir de otra forma el paralelismo presente entre las diferentes redes, siendo que el cálculo de la misma capa de todas las redes también involucra independencia con los demás cálculos, tal como se pretende ilustrar en la siguiente figura, representando el cálculo de todas las capas H2 para NP diferentes redes.

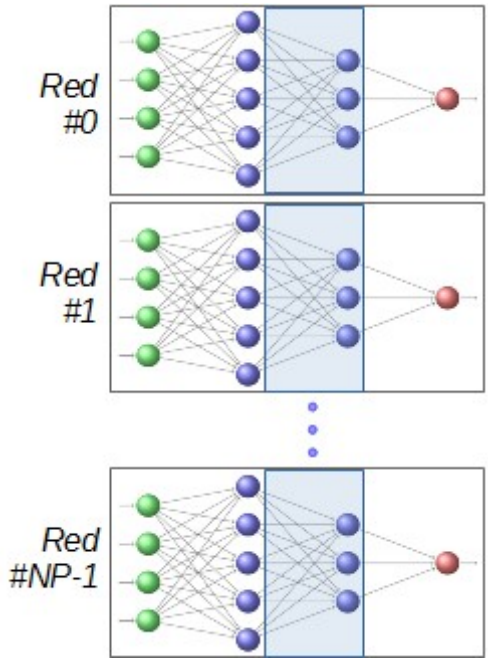

Ilustración 23: Representación de la independencia en el cálculo de todas las capas H2.

Dadas las características presentadas, surge la pregunta: ¿Cómo aprovechar estas características de paralelismo presente en los cómputos tanto de las neuronas de una misma capa y en las etapas de diferentes capas haciendo uso de las facilidades de OpenCL? Esta es la respuesta que se prende contestar a continuación, donde se explica la estrategia utilizada para acelerar el proceso de cálculo de la fase forward de todas las redes.

Aceleración de la fase Forward: el kernel perceptrón

Considerando la característica recién presentada, de realizar el cálculo de todas las neuronas de una misma capa para todas las redes de forma independiente, se puede plantear el cálculo de todas las neuronas como sigue. Dada una muestra de entrada o estímulo:

- Cálculo de todas las neuronas de la **capa H1** para las NP redes. Esto equivale a realizar el cálculo de

  $$NP \cdot numH1$$

  Siendo numH1 la cantidad de neuronas presentes en la capa oculta 1.

- Cálculo de todas las neuronas de la **capa H2** para las NP redes. Esto equivale a realizar el cálculo de

  $$NP \cdot numH2$$

  Siendo numH2 la cantidad de neuronas presentes en la capa oculta 2.

- Cálculo de todas las neuronas de la **capa O** para las NP redes. Esto equivale a realizar el cálculo de

$$NP \cdot numO$$

Siendo numO la cantidad de neuronas presentes en la capa oculta de salida.

Considerando que las neuronas del MLP son todas iguales y aprovechando las características del espacio multidimensional de ejecución de kernels de OpenCL, se diseñó un kernel que implementa el cálculo de la neurona tipo perceptrón lineal con una función de transferencia de tangente hiperbólica (*tanh*), el cual se invoca en un espacio de ejecución de 2 dimensiones de Np x numN, siendo:

- NP, la cantidad de redes, que es igual a la cantidad de individuos de una población del algoritmo DE; y,
- numN, la cantidad de neuronas de la capa que se desea calcular (numH1, numH2 o numO).

Así, las coordenadas o índices 0 y 1 de los work-groups lanzados a ejecución se utilizan para indexar la red o individuo (0 a NP-1) y la neurona de la capa (0 a numN-1), respectivamente.

La neurona implementada por el kernel es un perceptrón lineal con una función de transferencia igual a la tangente hiperbólica, tal como lo describe la siguiente ecuación:

$$nVal_i = tanh\left(Bias_i + \sum_{j=0}^{numIL} \left(w_{i,j} \cdot x_j\right)\right) \quad , \quad con \;\; i \in \{0, numN\}$$

Siendo:

- *i* el índice de la neurona de la capa que se está calculando;
- $nVal_j$, el valor de la neurona *i*;
- $Bias_i$, el valor de inicialización, offset o bias de la neurona *i*, que forma parte de los parámetros de la red a evolucionar, junto con los demás pesos sinápticos de la red.
- *numIL*, la cantidad de neuronas de la capa anterior;
- *j*, el índice de las neuronas de la capa anterior;
- x_j, el valor de salida de la neurona *j* de la capa anterior;
- $w_{i,j}$, el valor del peso sináptico que conecta la neurona *j* de la capa anterior, con la neurona *i* de la capa que se está computando; y,
- *numIL*, el número de neuronas de la capa anterior.

El comportamiento del kernel puede observarse en el siguiente cuadro en el que se transcribe la descripción del kernel, que se ha denominado *Perceptron_Kernel*, en el que se destaca la presencia de dos consultas al espacio de índices con que se debe ejecutar el kernel, debido a que, como se ha descrito, debe ser llamado a ejecución en dos dimensiones.

```
__kernel void Perceptron_Kernel(
        // Datos de entrada de la red:
        //  - Pesos de la red
        __global float * restrict w, // Puntero a memoria donde se alojan los pesos de las NP redes
        //  - Vectores de entrada de la etapa = Salidas de la etapa anterior
        __global float * restrict inLayer,// Input Vector

        // Datos de salida del kernel
        //  - Vectores de salida  de la etapa = Entradas de la etapa posterior
        __global float * restrict outLayer,     // Output Calc.

        // Parametros de la capa:
        //  - Cantidad de datos de entrada = Numero de neuronas de la capa anterior o capa de ent.
        unsigned const numIL,
        //  - Cantidad de datos de salida  = Numero de neuronas de la capa presente o capa de sal.
        unsigned const numOL,
        //  - Offset de ubicacion del conjunto de pesos de la capa desde la posicion "0" del
        // conjunto (netId*DE_D)
        unsigned const offset,  //    Puede ser {0, numIH1 o numH1H2} = {0,D1,D2}

        // Parametro del algoritmo DE:
        //  - Tam de los vect. de PX (DE_D = D1+D2+D3). Igual al nro de conexiones de toda la red
        unsigned const DE_D
        )
{
    // Se llamarian K Kernels en 2 dimensiones: [NP,numN]; siendo:
    //  - NP  : La cantidad redes o individuos del algoritmo DE
    //  - numN: La cantidad de neuronas de la capa
    int netId = get_group_id(0);   // Id. de la red = {0,...,NP-1} => Seran NP redes en general
    int   nId = get_group_id(1);   // Id. de la neurona de la red "netId"

    __local unsigned i;  // Indice para recorrer las numIL entradas

    __local float nVal;  // Valor de la neurona

    // Inicializacion del valor de la neurona con el BIAS
    //nVal = w[nBias_idx];
    nVal = w[netId*DE_D + offset + nId + (numIL*numOL)];

    // Acumulacion del producto de la neurona de la capa anterior por su correspondiente peso
    for(i=0;i<numIL;i++){
        //nVal = nVal + ( inLayer[netId*numIL+i] * w[nWeights_offset + (i*numOL)] );
        nVal = nVal + ( inLayer[netId*numIL+i] * w[netId*DE_D + offset + nId + (i*numOL)] );
    }
    nVal = transf(nVal);

    // Escribe el valor calculado en memoria
    outLayer[netId*numOL+nId] = nVal;
}
```

El kernel en cuestión presenta un sencillez notable, a la vez que admite el cálculo de redes de cualquier dimensión y tamaño. Sumado a esto, si se desean implementar otros tipos de neuronas, el kernel puede ser fácilmente modificado para computar los cambios que se necesiten. Una de las facilidades se desprende de que la función de transferencia de la neurona se implementa por medio de una función auxiliar, denominada *trasnf()*, de modo que basta con modificar el comportamiento de esta función. Un ejemplo de esto fue la utilización de una función de transferencia: unitaria para valores de entradas entre -1 y 1, y con aplicación de saturación para valores fuera del rango. Esta función de transferencia se utilizó durante las primeras etapas de desarrollo hasta que la incorporación de la función tangente hiperbólica descrita en por HDLs fue realizada.

Una pequeña desventaja o cuestión a tener en cuenta con la utilización del mismo kernel para el cálculo de todas las capas, es que debe realizarse un proceso de replicación del conjunto de muestras de entrada, a fin de que funcionen como las capas de neuronas anterior para cada una de las capas H1 de las NP redes a computar. Sin embargo, ésta es sólo en una pequeña desventaja, dado que debe realizarse una única vez, antes de que el algoritmo DE se ponga en marcha, con lo que puede ser realizada por el host como parte del setup inicial, sin afectar la performance posterior. Además, dado que esta réplica de muestras se implementa sobre memoria global (físicamente, memoria externa de los dispositivos), se dispone de grandes cantidades de la misma, un costo que se puede pagar a favor de tener una mejor utilización del hardware de la FPGA, ya que no se requieren modificaciones en el comportamiento del kernel cuando se computa una capa u otra del MLP.

Kernel adicionales para la acumulación del error

Adicionalmente a los kernels descritos anteriormente que forman la parte esencial del desarrollo de este trabajo, se encontró conveniente implementar un kernel que realice la acumulación del error generado tras el cálculo de la salida de cada patrón estímulo. Más precisamente, este kernel se encarga de acumular el cuadrado de la diferencia entre la salida de cada red con la esperada, para cada una de las NP redes.

Para ello, el kernel toma como datos de entrada, a partir de memoria global, la salida esperada para el correspondiente patrón y la salida de cada una de las redes. Se computa entonces el cuadrado de la diferencia entre estos valores y el resultado es acumulado sobre un valor de salida en memoria global. El identificador del patrón se pasa al kernel como un argumento.

La descripción del kernel, denominado *ErrCalc_Kernel*, en cuestión se transcribe a continuación.

```
__kernel void ErrCalc_Kernel(
        // Datos Entrada del Kernel
        //  - Respuestas de la red para cada el vector de entrada en cuestion
        __global float * restrict calcOut,// Calculated Output
        //  - Respuestas esperadas de la red para cada vector de entrada
        __global float * restrict specOut,// Spected Output

        // Datos de salida del kernel
        //  - Funcion de error: acumulacion del error absoluto de cada estimulo
        __global float * restrict ferr,

        // Parametro de ejecucion
        unsigned const patId
        )
{
    // Se llamarian:
    //  NP Kernels, 1 para cada red (conjunto de pesos de la poblaciones Px o Pu)
    int id = get_global_id(0);
```

```
    __local float err;
    err  = (calcOut[id] - specOut[patId]);
    ferr[id] += err * err;
}
```

Descripción del comportamiento del programa del host

En la práctica, se pusieron dos tandas de NP redes, una para calcular el error de los individuos padres, x_p, de la población padre Px; y otra para el cálculo del error de los individuos de prueba u_p, de la población de prueba Pu.

El programa del host cuenta con dos grandes partes: la primera en la que se computa todo el algoritmo DE en el host; y la segunda parte, en la que se realiza el mismo cómputo pero utilizando la aceleración en la FPGA con los kernels descritos.

La implementación realizada en el host, es llevada a cabo para verificar la validez de los datos y para realizar las mediciones temporales de ejecución de cada etapa y del total del cómputo del algoritmo, para luego ser comparadas con las implementaciones aceleradas.

Por su lado, la segunda parte involucra el uso de los aceleradores, por lo que merece la pena realizar un detalle respecto a las principales secciones y pasos realizados en cada una:

1. ***Inicialización de datos***:
   - Pesos de las NP redes a evolucionar
   - Conjunto de datos de entrenamiento
   - Semillas para las instancias del generador de número aleatorios.
2. ***Lectura y carga del archivo .aocx*** en la memoria del host, con la implementación de los kernels.
3. ***Setup de OpenCL***:
   1. Obtención de los datos de la plataforma y del dispositivo acelerador (la FPGA).
   2. Creación del contexto para el dispositivo.
   3. Creación de la cola de comandos para el dispositivo.
   4. Programación de la FPGA con el objeto programa cargado en memoria.
   5. Creación de los objetos kernels para:
      1. La fase de Mutación y Recombinación;

2. El cálculo de los errores de cada conjunto de redes (un objeto para la población de padres Px y otro para la población de vectores de prueba Pu).
3. El cálculo del MLP, es decir, que se crea un objeto kernel para el kernel perceptrón. Aquí se pueden aplicar dos soluciones: una que consiste en la re-utilización del mismo objeto para el cálculo de cada una de las capas de las redes, lo que conlleva la configuración de sus argumentos antes de realizar el cálculo de cada capa; y la segunda, en la que se crea un objeto kernel para cada una de las capas, lo que permite mantener la configuración de los argumentos de cada objeto sin modificaciones durante la ejecución de todo el algoritmo. La aplicación de una u otra solución no produce mejoras de performance significativas.

6. Creación de los buffers de memoria en el dispositivo:
    1. Para almacenar los valores de las neuronas de las capas H1, H2 y de Salida (capa O).
    2. Para almacenar los valores de los estímulos patrones y sus salidas esperadas.
    3. Para almacenar las semillas de la instancia de cada generador de números aleatorios.
    4. Para almacenar los pesos padres (población Px) y los pesos de prueba (población Pu).
    5. Para almacenar los errores de cada conjunto de redes: padres y de prueba.
7. Mapeo de la memoria compartida entre host y device:
    1. Se comparte la memoria que almacena los vectores padres y de prueba, para realizar la copia de uno u otro directamente en el lado del host, dado que la etapa de selección del DE es computada por el host.
    2. Se comparte la memoria que almacena los errores de cada conjunto de redes: padres y de prueba, a fin de disponer de dichos valores desde el lado del host para computar la etapa de selección del DE.

4. ***Configuración de argumentos***. Se deben configurar adecuadamente los argumentos de cada objeto kernel:
    1. Kernel de mutación y recombinación: direcciones de memoria de la población de padres y de vectores de prueba, dirección de memoria de las semillas del generador de números aleatorios, y los valores constantes del factor de escalado, del factor de recombinación, el tamaño de los vectores D y el tamaño de las poblaciones NP.
    2. Kernel de cálculo de los errores de la población de padres Px: direcciones de memoria donde se almacenan las salidas esperadas y las salidas generadas por las redes con los pesos padres, dirección de memoria donde se almacenan los acumuladores de error, y el valor de identificación del patrón computado.

3. Kernel de cálculo de los errores de la población de vectores de prueba Pu: equivalente a la configuración anterior, salvando que las direcciones de las salidas se corresponden con las salidas calculadas por las redes con pesos de prueba, y las direcciones donde se almacenan los errores acumulados, se corresponden con la población de vectores de prueba.
4. Kernels de cálculo de las neuronas de las capas: H1, H2 y O para los pesos padres. En estos objetos kernels se configuran las direcciones donde se almacenan los valores de entrada de las neuronas, donde se almacenan los valores calculados para cada neurona, la dirección de los pesos padres (Px), y los valores constantes de número de neuronas de entrada (de la capa anterior), número de neuronas de la capa, un offset auxiliar para el cálculo de direccionamiento en la memoria de los pesos y el tamaño de los vectores de pesos del algoritmo DE (esto es el parámetro D).
5. Kernels de cálculo de las neuronas de las capas: H1, H2 y O, para los pesos de prueba. Se configuran mismo argumento que para los kernels anteriores, sólo que relativos a la población de pesos de prueba.

5. ***Escritura de la memoria del dispositivo***. Aquella memoria que no es compartida con el device, se escribe por medio de una transferencia de datos por medio comandos enviados por la cola. Tal es el caso de:
    1. Los estímulos patrones
    2. Las salidas esperadas para cada estímulo patrón
    3. Las semillas de los generadores
6. ***Ejecución del algoritmo DE***. El mismo se computa en un bucle en el que las iteraciones se definen según el número de generaciones a calcular con el algoritmo. Dentro del bucle se computa:
    1. La mutación y recombinación por medio de un llamado a ejecución del correspondiente kernel, en un espacio dimensional de NP elementos.
    2. Puesta a cero de los acumuladores de error para la población de pesos padre.
    3. Bucle para calcular la salida de todos los estímulos patrón de las NP redes con los pesos padres (Px), en los que se ejecuta:
        1. Configuración del argumento que le indica al kernel H1 la dirección de memoria del conjunto de entradas del estímulo patrón correspondiente a la iteración.
        2. Cálculo de la capa H1, de las redes Px, por medio del llamado a ejecución del kernel perceptrón en un espacio dimensional igual a {NP,numH1}.
        3. Cálculo de la capa H2, de las redes Px, por medio del llamado a ejecución del kernel perceptrón en un espacio dimensional igual a {NP,numH2}.

4. Cálculo de la capa O, de las redes Px, por medio del llamado a ejecución del kernel perceptrón en un espacio dimensional igual a {NP,numO}.
5. Configuración del argumento del kernel que acumula el cuadrado del error, que indica la identificación del estímulo patrón correspondiente a la iteración.
6. Cálculo de la acumulación del cuadrado del error de la salida del red para el patrón de la iteración.

4. Puesta a cero de los acumuladores de error para la población de pesos de prueba.
5. Bucle para calcular la salida de todos los estímulos patrón de las NP redes con los pesos de prueba (Pu), en los que se ejecutan los mismos 6 pasos que para los pesos padres, pero con los objetos kernel (del perceptrón) correspondientes.
6. La selección, implementada completamente en el host, en la que se compara el valor acumulado de los cuadrados de los errores de cada individuo padre con su correspondiente individuo de prueba (valor de la función objetivo), y se deja sobrevivir sobre la nueva población de padres, el que menor valor haya generado.

Estos son los pasos o secciones más importantes de la parte del programa del host orientada a acelerar el cálculo del algoritmo. En cada una de las etapas: mutación y recombinación, cálculo del valor de la función objetivo, y selección, se realizan las mediciones temporales pertinentes, a fin de hacer una comparación frente a la solución conseguida con la implementación pura en el lado del host.

Del flujo del programa, se puede destacar la implementación de la etapa de selección del lado del host y no sobre la FPGA como un kernel acelerador. Esto se debe a que, como se adelantó brevemente, con la plataforma elegida, no fue posible hacer que el diseño de todos los kernels se pudiera implementar con los recursos disponibles del dispositivo. Por ello, analizado el comportamiento temporal de la etapa de selección y con la posibilidad de implementar memoria compartida entre el host y el device, se encontró conveniente que dicha etapa fuese realizada sólo por el host, consiguiendo así que el diseño de los demás kernels se pudiese implementar sobre la FPGA elegida. Como contra, puede mencionarse que, si se desea cambiar de plataforma sobre una que no tuviese memoria compartida entre host y device, se debería computar la etapa de selección con un kernel a fin de no deteriorar la performance del sistema. Esto se debe a que para implementar la selección sólo con el host, el mismo debe acceder a los datos calculados de las funciones de desempeño, y luego copiar los vectores de prueba sobre la nueva población de vectores padre cuando corresponda. Si no se dispone de memoria compartida, estas dos operaciones requerirían de muchas transferencias de datos entre memorias del host y del device, lo que traería aparejada así la disminución de performance debido a transferencias de memoria.

Resultados

Problema modelo a resolver

El problema modelo a resolver elegido para realizar las mediciones pertinentes, fue el problema típico de clasificación en redes neuronales artificiales: el cálculo de la función XOR. Para ello, se definió el conjunto de entrenamiento como la tabla lógica de la compuerta XOR codificada en valores reales, tal como se representa a continuación:

Identificación de estímulo patrón	Entrada #1	Entrada #2	Valor de salida
#0	+0.95	-0.95	+0.95
#1	-0.95	+0.95	+0.95
#2	+0.95	+0.95	-0.95
#3	-0.95	-0.95	-0.95

Siendo +0.95 el valor real correspondiente al '1' lógico, y -0.95 el correspondiente al '0' lógico.

La columna más a la izquierda representa los valores con los que se identifican las muestra del conjunto de entrenamiento, existiendo en este caso, tan sólo cuatro muestras.

Resultados del entrenamiento

Independientemente de la implementación (sea sólo del lado del host o con aceleradores en el device), la aplicación del algoritmo DE como método de evolución de los pesos de las redes dio excelentes resultados numéricos, ya que se consiguió entrenar el conjunto de redes de la población padre, a tal punto que el error cuadrático medio computado para el conjunto de entrenamiento resultó ser menor que el obtenido aplicando uno de los métodos entrenamiento tradicionales del MLP: el algoritmo RBP.

Resultados temporales y de aceleración

Para medir los factores de aceleración conseguidos con el uso de la FPGA como dispositivo acelerador, se compararon los tiempos de ejecución entre una implementación pura en el host ARM de la Cyclone V, y una implementación acelerada por los kernels OpenCL antes descritos, implementados sobre el device OpenCL (la parte FPGA de la Cyclone V).

Hay que considerar que los tiempos de ejecución varían dependiendo de los siguientes parámetros o factores:

- Del Algoritmo de Evolución Diferencial:
  - El número de generaciones a realizar, parámetro *G*.
  - El número de individuos en las poblaciones o tamaño de las mismas, parámetro *NP*.
- Y de la red Perceptrón-Multicapa implementada*:
  - Número de neuronas de la capa oculta 1, parámetro *numH1*.
  - Número de neuronas de la capa oculta 2, parámetro *numH2*.
  - Número de muestras del conjunto de entrenamiento, parámetro *numPats*.

**Se considera que la cantidad de entradas y salidas están definidas por el problema, y por ende no son parámetros susceptibles de ser modificados.*

Entonces, las mediciones del factor de aceleración se realizaron modificando de uno en uno estos parámetros, a la vez que los demás se mantenían fijos en un valor *default* como sigue:

- Número de generaciones computadas, $G = 500$.
- Tamaño de la población, $NP = 10$.
- Número de neuronas de la capa oculta 1, $numH1 = 50$.
- Número de neuronas de la capa oculta 2, $numH2 = 20$.
- Número de muestras del conjunto de entrenamiento, $numPats = 4$.

Seguidamente se presentan las mediciones del factor de aceleración para variaciones de cada uno de estos parámetros.

Comportamiento del factor de aceleración para G variable

La siguiente gráfica representa los valores del factor de aceleración medido para diferentes números de generaciones del algoritmo DE computadas, para cada parte del algoritmo: mutación y recombinación, fase forward de las 2·NP redes para cada muestra del conjunto de entrenamiento (de aquí en más, cálculo de la

función objetivo) y la etapa de selección; y también para el cómputo del algoritmo completo.

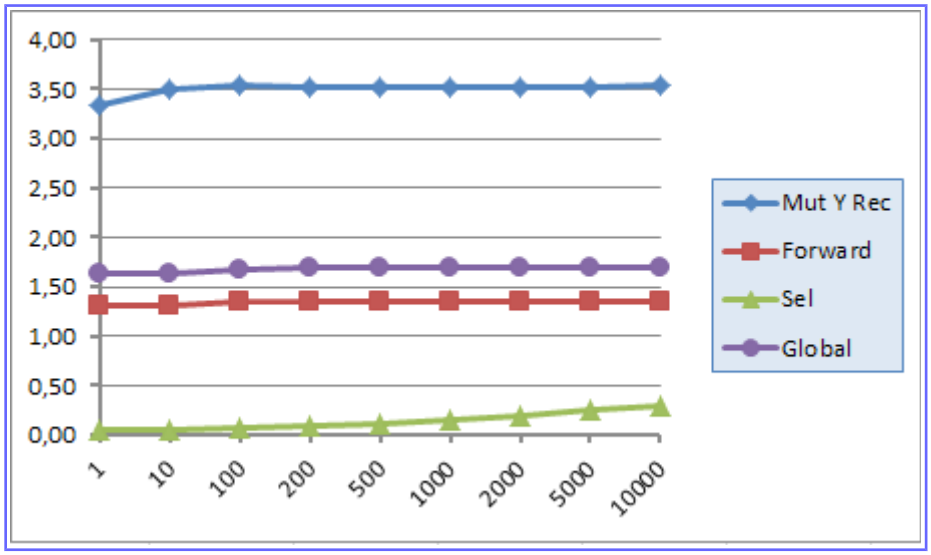

Ilustración 24: Factores de aceleración conforme varía el número de generaciones del algoritmo DE.

Como se puede apreciar, el factor de aceleración se mantiene prácticamente constante entre diferentes número de generaciones como era de esperarse, dado que la utilización de más o menos generaciones es controlada por el programa del host en un bucle *for*, en el cual todas las instrucciones son ejecutadas de forma secuencial. Por ello, al aumentar el número de iteraciones de este bucle, el tiempo de ejecución crece linealmente, con la misma pendiente con la que crece el tiempo de ejecución de la implementación en el host. Así, el cociente entre estos tiempos de ejecución se mantiene para diferentes números de generaciones, haciendo que los factores de aceleración se mantengan constantes.

El factor de aceleración global (curva *violeta*) es superior a 1, lo que indica que, para los parámetros de ejecución implementados (valores *default*), existe una grado de aceleración, pese a que el mismo no sea tan significativo (valor en torno a 1,7).

La etapa que percibe una mayor aceleración es la de mutación y recombinación (en *azul*), consiguiendo una aceleración de un factor de 3,5; mientras que el cálculo de la función objetivo para las 2·NP redes, percibe un factor de aceleración en torno a 1,4 (en *rojo*), valor que puede resultar aceptable, pero que no destaca demasiado.

Por último, se debe señalar que el factor de aceleración de la etapa de selección (curva *verde*) se ve notablemente reducido, dado que siempre se mantiene por debajo de 1, y no lega a superar el valor de 0,4. Esto se debe principalmente a que esta etapa se implementa en el lado del host pero sobre memoria compartida entre el host y el device. Más precisamente, la implementación de etapa de selección consiste en

la comparación de números reales y una posterior transferencia de datos entre dos espacios de memoria cuando corresponde. Esta transferencia se realiza con la función *memcpy*, pero en un caso entre memoria de datos del host, y la otra entre memoria de datos compartida entre el host y el device, siendo esta última implementación más lenta que la primera, y siendo este el motivo por el que la performance de la implementación acelerada es menor en comparación con la implementación puramente del lado del host.

Comportamiento del factor de aceleración para NP variable

La siguiente gráfica representa los valores del factor de aceleración medido para diferentes tamaños de las poblaciones, para cada parte del algoritmo (las tres partes mencionadas anteriormente), y también para el cómputo completo del algoritmo.

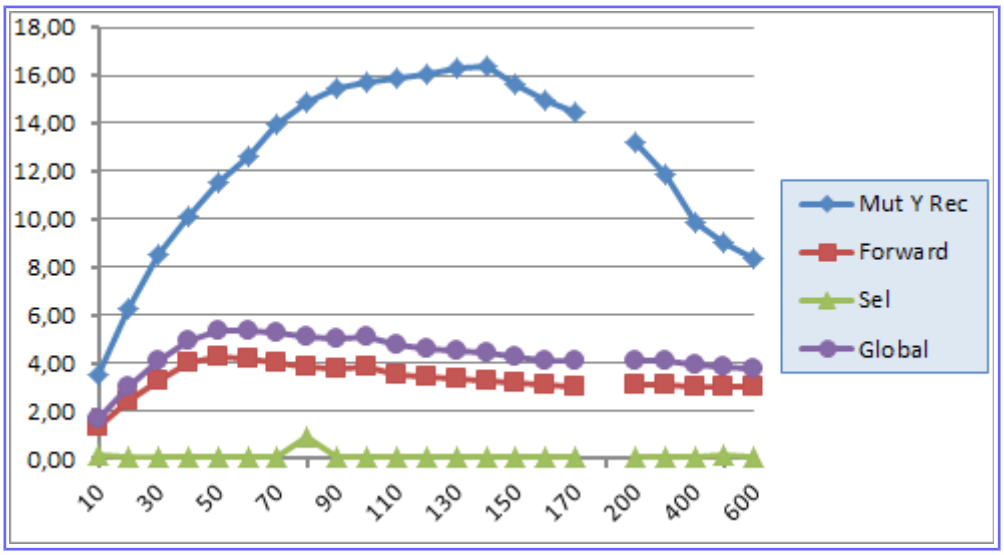

Ilustración 25: Factores de aceleración conforme varía el tamaño de las poblaciones.

En este caso, se ve cómo el factor de aceleración global aumenta considerablemente en la medida en que aumenta el tamaño de las poblaciones, hasta un tamaño de población de 50 individuos, donde alcanza un factor de aceleración máximo de 5,5 aproximadamente. Tras este máximo, el factor de aceleración cae parcialmente hasta establecerse en torno a un valor de aproximadamente 4. Ya para tamaños muy grandes de poblaciones, se percibe una tendencia de disminución en el factor de aceleración, que se acompaña de la disminución del factor de aceleración para la etapa de mutación y recombinación.

Nuevamente, la etapa que percibe una mayor aceleración y la que más se beneficia del incremento en los tamaños de las poblaciones, consiguiendo valores de aceleración por encima de 16, es la de mutación y recombinación.

Por otro lado, el cálculo de la función objetivo es el cálculo o etapa que claramente comanda la forma del factor de aceleración global, por lo que puede comenzarse a concluir que se trata de la etapa con mayor carga computacional. Si bien el factor de aceleración global se beneficia de la aceleración de la etapa de mutación y recombinación, parece ser más relevante la aceleración que percibe el cálculo de las redes.

Comportamiento del factor de aceleración para numH1 variable

La siguiente gráfica representa los valores del factor de aceleración medido para diferentes números de neuronas de la capa oculta 1, para cada parte del algoritmo y para el cómputo del algoritmo completo.

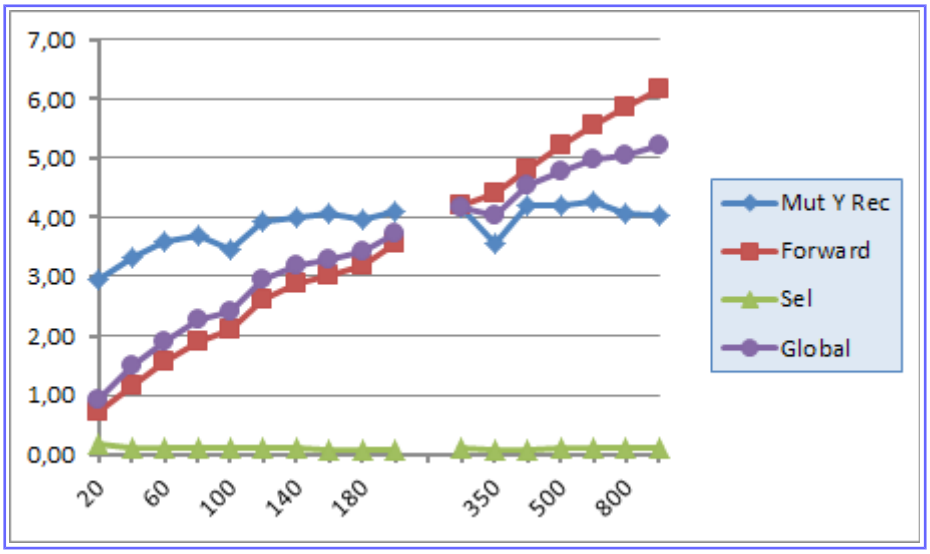

Ilustración 26: Factores de aceleración conforme varía el número de neuronas de la capa oculta H1.

Esta gráfica fortalece la conclusión anteriormente expuesta, de que el factor de aceleración global del algoritmo depende principalmente de la aceleración que involucra el cálculo de la red. Visualmente, las dos curvas de los factores de aceleración, del cálculo de la función objetivo y del cómputo global, presentan casi la misma forma, salvo pequeñas diferencias debidas a que la etapa de mutación y recombinación presenta unos mínimos locales en ciertos puntos, que influyen levemente en la forma de la curva del factor de aceleración global.

Analizando las curvas de forma independiente, se encuentra con que el factor de aceleración del cálculo de la red crece monótonamente conforme crece el número de neuronas en la capa oculta 1. Cuando se implementa una capa oculta de dimensiones más que considerables, esto es por encima de 500 neuronas, la aceleración del cálculo de la red supera el factor de 5, un valor más que considerable.

Por su lado, la etapa de mutación y recombinación también se ve acelerada, pero tiende a mantener su

aceleración estable con un factor de entre 3 y 4. Presenta dos puntos particulares, dos mínimos locales, donde el factor de aceleración cae para 80 y 350 neuronas. Dado que en principio no presentan un comportamiento lógico, se estima que estas "irregularidades" en el comportamiento se deben a cuestiones de implementación del algoritmo ya puesto en funcionamiento, esto es, a parámetros de ejecución, ya que los kernels compilados son siempre los mismos, y lo que cambian son los espacios dimensionales en los que son ejecutados. Por ejemplo, uno de los supuestos que se ha planteado en este comportamiento, es que puede deberse a la distribución de los datos en los bancos de memoria, que con ciertos números de neuronas en la capa 1, se producen desbalances entre las cargas sobre un banco de memoria y el otro; desbalances que luego desaparecen en tanto aumentan las cargas sobre los bancos.

Por último, no existe aceleración para la etapa de selección, tal como se ha presentado en los casos previos, y como se mantiene en los siguientes casos que se pueden observar a seguidamente, dado que ninguno de los parámetros modificados infiere cambios en el comportamiento del cálculo de esta etapa.

Comportamiento del factor de aceleración para numH2 variable

Es de esperar que el comportamiento del los factores de aceleración para cada parte del algoritmo, como para el total del mismo, sea igual si se modifica el número de neuronas de la capa oculta 2, en lugar de hacerlo en la capa oculta 1 como se acaba de presentar, dado que sólo se modifica la cantidad de veces que se realiza exactamente el mismo cómputo. Sin embargo, como no todas las especulaciones terminan siendo acertadas, se decidió medir los factores de aceleración para diferentes números de neuronas de la capa oculta 2, esperando obtener resultados similares a los anteriores, y que se representan en la siguiente gráfica.

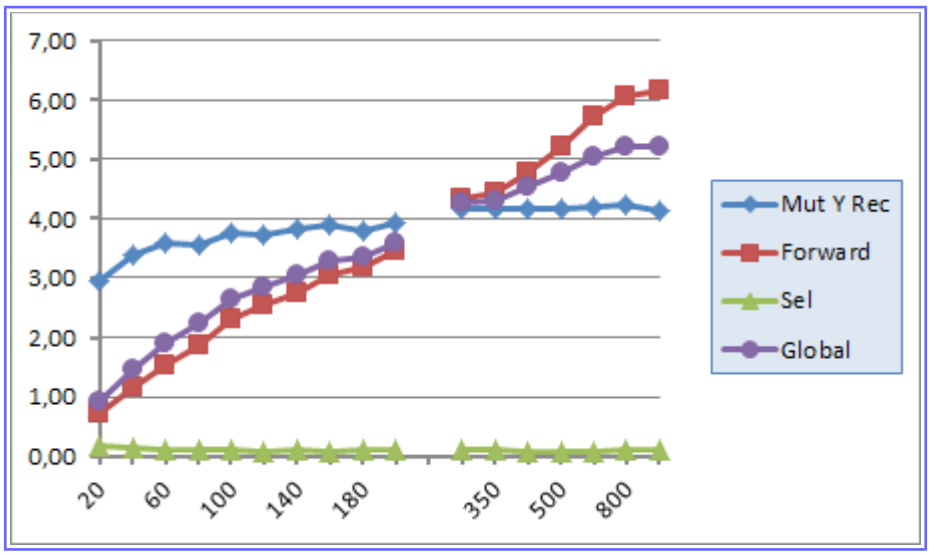

Ilustración 27: Factores de aceleración conforme varía el número de neuronas de la capa oculta H2.

Tal como puede apreciarse, el comportamiento se mantiene respecto al caso anterior, lo que confirma las especulaciones realizadas. En este caso, no existen mínimos locales tan notables en el comportamiento del factor de aceleración de la etapa de mutación y recombinación.

Por su parte, tanto el factor de aceleración global como el del cálculo de la función objetivo, mantienen casi las mismas formas que para el caso anterior, y conservan prácticamente los mismos valores. La aceleración global consigue factores de aceleración por encima de 5 para redes de dimensiones muy grandes, rango en el que el factor de aceleración del cálculo de la función objetivo está por encima de 6.

Las pequeñas diferencias que se presentan entre las dos últimas gráficas, las de variaciones en el número de neuronas de las capas ocultas 1 y 2, se deben a que, para el primer caso, la red se mantenía con 20 neuronas en la segunda capa, mientras que en este último caso, la primera capa se mantenía con las 50 neuronas definidas inicialmente como *default*, con lo que, pese a realizarse las mediciones para las mismas cantidades de neuronas, la cantidad total de neuronas y conexiones en realidad varía. Estas pequeñas diferencias entre las dos gráficas, alentaron a realizar las medidas de los factores de aceleración haciendo variar de forma conjunta el número de neuronas en las dos capas ocultas, tal como se presenta a continuación.

Comportamiento del factor de aceleración para numH1xnumH2 variable

En la siguiente gráfica se representan los valores del factor de aceleración para variaciones conjuntas de dos parámetros: el número de neuronas en la capas ocultas 1 y 2.

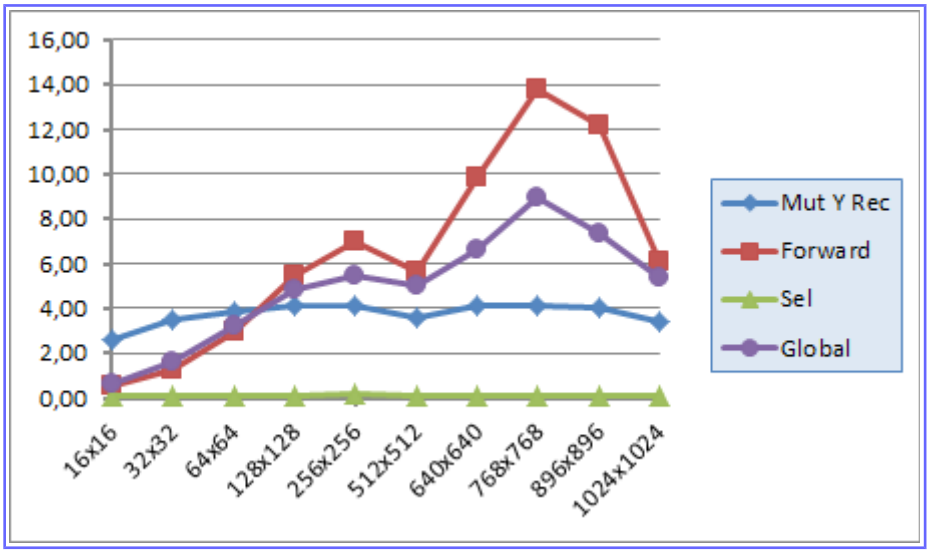

Ilustración 28: Factores de aceleración conforme varía, el número de neuronas de las capas ocultas H1 y H2.

En este caso, el número de neuronas de cada capa se ha echo crecer de forma conjunta, igualando el número de neuronas entre ellas, con lo que las dimensiones de la red crecen cuadráticamente, esto involucra el número total de neuronas en la red, y el número total de conexiones.

Nuevamente, se visualiza una gran dependencia entre la aceleración percibida por el cálculo de la red y la aceleración global de todo el algoritmo, dado que las dos curvas mantienen la misma forma. Con el aumento del número total de neuronas implementadas en la red, aumenta el factor de aceleración, tanto para el cálculo de la función objetivo, como para el cálculo de todo el algoritmo. Los valores conseguidos para la aceleración son más que notables, superando el primer orden de magnitud en lo que al cálculo de la función objetivo se refiere (un valor de casi 14), mientras que la aceleración global consigue un factor próximo a 9, muy cercano del primer orden de magnitud.

En cuanto a la aceleración de la etapa de mutación y recombinación, el comportamiento se mantiene similar a como se había comportado para variaciones en la cantidad de neuronas de la capa oculta 1 o la 2. Sin embargo, se percibe una leve mejora en el factor de aceleración con el aumento del número de neuronas en general. Este aumento se debe a que, al aumentar el número de neuronas en las capas, se aumenta el número total de conexiones de la red, que involucra un aumento en las dimensiones de los individuos. Por lo tanto, la cantidad de información que procesa el kernel de mutación y recombinación es mayor, mejorando así la aceleración ofrecida, pese a que no sea el parámetro del que más depende la aceleración de esta etapa (recordar que el parámetro que más influye en este factor de aceleración es el tamaño de las poblaciones).

Comportamiento del factor de aceleración para un número de muestras de entrenamiento variable

Como última alternativa en las pruebas, se decidió medir el factor de aceleración para un número variable de muestras en el conjunto de entrenamiento. Dado que el problema modelo con el que se ha trabajado sólo posee un conjunto de entrenamiento de cuatro muestras, se decidió aumentar el número de las mismas replicando este conjunto un número entero de veces, simplemente con el fin de aumentar la carga computacional involucrada en el cálculo de la función objetivo. Así se consigue emular el comportamiento que pudieran tener otros problemas, en los que sí se dispone de un conjunto de entrenamiento con un número mayor de muestras.

Los resultados medidos para los factores de aceleración obtenidos a partir de variaciones en el número de muestras del conjunto de entrenamiento, se presentan en la siguiente gráfica.

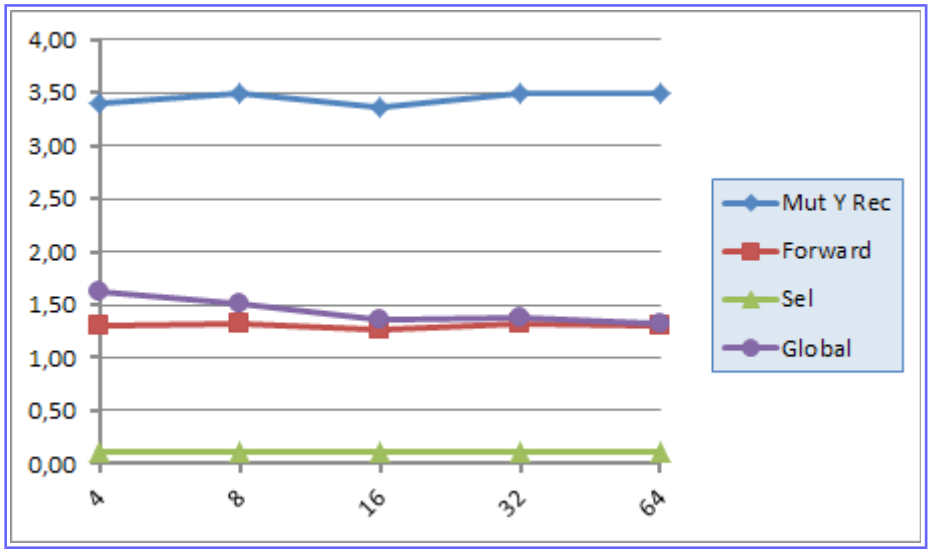

Ilustración 29: Factores de aceleración conforme varía el número de muestras del conjunto de entrenamiento.

Como se podría haber esperado, los cuatro factores de aceleración se mantienen casi estables, siendo el factor de aceleración global el que más varía, con una tendencia a establecerse en torno al factor del cálculo de la función objetivo, comportamiento que se analiza en los próximos párrafos.

El echo de que los factores de aceleración se mantengan estables, se debe a que al aumentar el número de muestras, sólo se aumentan las iteraciones en las que se computan los cálculos de las tres capas de las redes (capas H1, H2 y O), proceso que, pese a estar acelerado, se itera de forma secuencial. Con ello, su tiempo de ejecución crece linealmente con el número de muestras del conjunto de entrada. Su tiempo de ejecución en la implementación del lado del host se incrementa de la misma forma, con lo que el cociente entre estos tiempos de ejecución se mantiene constante, de forma similar a cómo ocurría con el aumento del número de generaciones explicado anteriormente.

Por otro lado, el factor de aceleración global tiene un comportamiento con una tendencia a disminuir conforme aumenta el número de muestras del conjunto de entrenamiento. Sin embargo, pese a que en la gráfica no se termina de percibir, esta disminución converge sobre el valor del factor de aceleración del cálculo de la función objetivo. Este comportamiento de la curva se debe a que, con el aumento del número de muestras sólo se incrementa el tiempo de cómputo de la función objetivo, mientras que las etapas de mutación y recombinación, y de selección se computan en el mismo tiempo (dado que no tienen dependencias del conjunto de entrenamiento). Por ello, dicho cómputo es el que tiende a ocupar la mayor porción del tiempo de ejecución de todo el algoritmo, haciendo que, en lo que respecta al tiempo de ejecución, sea la única etapa que demanda un tiempo de ejecución considerable, mientras que el tiempo demandado por cálculo de las demás etapas se hace despreciable frente a este.

Por ejemplo, para el caso donde se computan 64 muestras de entrada, se obtiene que el cálculo de la función objetivo para cada individuo utiliza el 98,8% del total de tiempo del cómputo del algoritmo, relegando apenas un 1,18% para la etapa de mutación y recombinación.

Aún cuando el conjunto de muestras de entrenamiento es pequeño (en el caso del problema modelo el menor posible, de tan sólo 4 muestras), la distribución de los tiempos sigue siendo desproporcionada, donde el 83,68% del tiempo de ejecución se dedica a computar la función objetivo, mientras que la etapa de mutación y recombinación abarca el 16%.

Estos datos confirman la conclusión temprana elaborada en los análisis previos, donde se estableció que el factor de aceleración global depende mayormente de la aceleración de los cálculos que involucran la red neuronal.

Pese a todo ello, vale la pena señalar la presencia del acelerador que computa la etapa de mutación y recombinación, dado que para un conjunto de entrenamiento de cuatro muestras, el porcentaje de tiempo dedicado a esta etapa en la implementación puramente del lado del host abarca un 33,25% del tiempo total, mientras que en el caso de la implementación se reduce casi a la mitad, en 16%.

Análisis de los resultados temporales

Del análisis de los resultados temporales expuestos en el presente capítulo, se puede extraer que:

- El factor de aceleración de la mutación y recombinación crece conforme lo hace el número de individuos de la población, es decir, el parámetro NP.
- El factor de aceleración del cálculo de la función objetivo (que computa la fase forward de todas las redes individuos para cada muestra o estímulo patrón), mejora conforme crece el número total de neuronas de la red.
- El factor de aceleración de la etapa de selección está siempre por debajo de 1, en torno a 0.1, con la implementación realizada, la que no involucra la aceleración con la FPGA. Sin embargo, el tiempo de cómputo dedicado es despreciable comparado con el tiempo de cómputo dedicado a las otras dos etapas del algoritmo, por lo que la disminución en su velocidad de cómputo es aceptable.
- El factor de aceleración global, esto es del cómputo de todo el algoritmo, depende mayormente del cálculo de la función objetivo, debido a que es la etapa que mayor carga computacional requiere.
- El número de generaciones implementadas y el número de muestras del conjunto de entrenamiento no poseen injerencias en lo que a aceleración del algoritmo se refiere.

Resultados de implementación sobre la plataforma

Como se mencionó en la introducción, la implementación de los kernels sobre la FPGA se realiza en un paso muy abstracto, en el que no es posible analizar con facilidad cómo se traduce el código de los kernels en hardware de la FPGA. Sin embargo, es importante considerar los resultados finales de dicha implementación en cuanto utilización de recursos de hardware y frecuencia máxima de trabajo se refiere.

La siguiente tabla resume la utilización de recursos de la FPGA Cyclone V SoC 5CSEMA5F31C6, montada en la plataforma DE1-SoC, por parte de la implementación de los kernels. Estos recursos se extraen del archivo de reporte generado por el compilador OpenCL de Altera, el AOC.

Recurso	Utilización (%)
Utilización lógica	86 %
Registros lógicos dedicados	39 %
Bloques de memoria	92 %
Bloques DSP	22 %

Por último, entre los archivos de reporte, se encuentra que la frecuencia máxima de trabajo de la FPGA que se consigue con la implementación de los kernels descritos es de:

Frecuencia máxima de trabajo	130.75 MHz

Conclusiones

Sobre los objetivos

Objetivo principal

Ya que se ha conseguido implementar el algoritmo DE como método de evolución de los pesos de un conjunto de redes neuronales tipo MLP, consiguiendo valores de desempaño menores que los conseguidos con el entrenamiento realizado con el RBP, implementado el desarrollo sobre una plataforma basada en FPGA (la DE1-SoC), utilizando kernels OpenCL como aceleradores de cómputo implementados sobre la FPGA, y consiguiendo resultados de aceleración considerables frente a la implementación pura en el lado del host; se concluye haber alcanzado satisfactoriamente el principal objetivo propuesto para este trabajo.

Además de los resultados de entrenamiento y de aceleración conseguidos, se ha logrdo desarrollar un conjunto de kernels que poseen una alta versatilidad, flexibilidad y portabilidad, haciéndolos idóneos para ser integrados en el proyecto de investigación de computación distribuida basada en dispositivos FPGAs para la optimización de la topología de redes neuronales artificiales, del área de *Diseño de Sistemas Digitales* del *Instituto de Instrumentación para Imagen Molecuar*, I3M; ya sea sobre otras plataformas e incluso otras implementaciones de cómputos.

La flexibilidad de los kernels se desprende de que admiten una configuración en tiempo de ejecución, lo que les da la posibilidad de ser utilizados en la resolución de diversos problemas sin la necesidad de realizar re-compilaciones. Utilizando el mismo kernel perceptrón pueden implementarse redes neuronales MLP con diferentes cantidades de capas ocultas, diferentes números de neuronas en cada capa, y/o diferentes números de entradas y de salidas (según se corresponda con el problema). Por su lado, el kernel que computa la mutación y recombinación puede funcionar en los cálculos del DE para diferentes tamaños de poblaciones, admitiendo la configuración del factor de escalado o del factor de recombinación por medio de sus respectivos argumentos, incluso entre generaciones sucesivas.

La portabilidad de los kernels se presenta en torno a sus descripciones en los códigos fuentes, dado que

no se utilizan tecnologías o herramientas particulares de Altera o sus FPGAs. Así, los códigos que describen a los kernels pueden ser implementados sobre otras plataformas aceleradoras OpenCL, estén basadas en dispositivos FPGA o no. Esta portabilidad anima a implementar los kernels en otras plataformas basadas en FPGAs de Altera disponibles en el laboratorio, en las que al contar con mayor cantidad de recursos se podrían implementar los kernels con múltiples caminos de datos (dimensiones del SIMD mayores al valor por *default* de 1), esperando conseguir mejores resultados de aceleración.

Por otro lado, hay que tener en cuenta que también existen limitaciones. Una está vinculada a los requerimientos de memoria por parte del algoritmo en sí, dado que se requiere una gran cantidad de espacio en memoria para almacenar los valores de las neuronas y los pesos de las 2·NP redes, siendo que la cantidad de pesos sinápticos que se debe almacenar aumenta exponencialmente conforme aumenta el número de neuronas de una capa, y linealmente conforme aumenta el tamaño de las poblaciones.

Objetivo secundario

El objetivo secundario planteado durante la propuesta del trabajo fue parcialmente conseguido, por lo que queda pendiente de completar en posteriores proyectos o desarrollos que se desprendan de este trabajo. El objetivo en cuestión fue la integración del módulo IP tangente hiperbólica (descrito en un HDL por el director de este trabajo, Rafael Gadea) como función auxiliar en OpenCL.

El objetivo fue parcialmente conseguido dado que la integración se llevó a cabo utilizando una técnica que no se corresponde con un procedimiento estándar fácilmente replicable para implementar otros módulos IP descritos en HDLs, como tampoco se trata de una técnica ofrecida por el soporte de Altera. La técnica consiste en '*enmascarar*' la tangente hiperbólica como otra función de OpenCL ya implementada de antemano por Altera. Esta función no puede ser cualquier función, sino que debe poseer idénticas características de entrada/salida y temporales (ciclos de latencia que parcialmente se pueden llegar a ajustar en la descripción HDL). Para realizar esta técnica se debe reemplazar el contenido del archivo HDL que contiene la descripción y el comportamiento del módulo que computa la función a ser reemplazada, colocando la descripción y el comportamiento de la tangente hiperbólica (en este caso particular), manteniendo inalterada la interfaz del primer módulo. Entonces, el módulo deseado se utiliza desde OpenCL mediante el llamado a la función auxiliar de la función reemplazada, como se explica a continuación.

En este trabajo, la función OpenCL que se reemplazó fue la función de redondeo hacia arriba o 'infinito positivo', la función *ceil()*, cuyo comportamiento se encuentra descrito en el archivo Verilog *acl_fp_ceil.v*, en la carpeta “C:\altera\16.0\hld\ip”. Entonces, para computar la tangente hiperbólica, simplemente se llama a la función *ceil()* desde la descripción del kernel, siendo que luego, el compilador terminará por generar el

hardware correspondiente al módulo que se describe en *acl_fp_ceil.v*, donde se ha colocado la descripción del módulo deseado, la tangente hiperbólica.

Para implementar, de forma estándar, en OpenCL una función con un IP Core descrito en un HDL, se deben seguir los pasos descritos en el capítulo 2 "Altera SDK for OpenCL Advanced Features", sección "OpenCL Library", sub-sección "Packaging an RTL Component for an OpenCL Library" de la guía de programación del SDK de OpenCL de Altera [27]. Este trabajo fue llevado a cabo consiguiendo realizar la integración, aunque, no fue posible correr un programa en la plataforma DE1-SoC que utilizara un kernel que integre un módulo IP descrito en HDL. Esto se debe a que la integración está soportada a partir de la versión 16.0 del conjunto del paquete de desarrollo de Altera, siendo que las compilaciones para el host utilizando esta versión no son compatibles con el Sistema Linux disponible para la plataforma DE1-SoC. Es por este motivo que la integración del módulo de la tangente hiperbólica no pudo llevarse a cabo con este procedimiento.

Una descripción más detallada, que los presentados en la guía de Altera, de los pasos que se deben seguir para realizar una integración de uno o más módulos IP descritos a partir de uno o más HDLs, puede encontrarse en el documento adjunto a este trabajo, que se corresponde con el informe del trabajo realizado para la asignatura "Sistemas Integrados Digitales" [28].

Objetivos personales

En cuanto a lo personal, la realización de este trabajo ha supuesto un desafío muy interesante al involucrar e integrar, en un mismo desarrollo, gran variedad de temas de interés: el desarrollo con una FPGA-SoC (con un procesador ARM y lógica reconfigurable), el uso de un Linux Embebido, el desarrollo con plataformas híbridas junto al procesamiento en paralelo, el uso de herramientas de diseño mixto (OpenCL), la implementación Redes Neuronales Artificiales, y la utilización de Algoritmos Genéticos, en particular, el Algoritmo de Evolución Diferencial; temas sobre los que sólo se poseía una experiencia apreciable en cuanto al desarrollo de hardware para FPGAs a bajo nivel (principalmente VHDL y Verilog) y en el desarrollo de programas en lenguaje C para microcontroladores.

Dada la temática expuesta y los resultados conseguidos, la elaboración de este trabajo ha sido gratamente satisfactoria. Inicialmente, los temas involucrados propiciaron y entusiasmaron la elección de la propuesta de Rafael Gadea; mientras que posteriormente animaron continua y constantemente la labor realizada.

Nuevos objetivos y trabajo futuro

El desarrollo del presente trabajo, ha dado también como resultado ideas e inquietudes que pueden dar lugar a posteriores desarrollos. Éstos se describen brevemente a continuación y se plantean como trabajos futuros.

Implementación del algoritmo DE sobre el host y multiplicación del SIMD en el kernel perceptrón

Como se vio en el capítulo de resultados, la mayor parte del tiempo de ejecución del algoritmo está dedicada al cálculo de la función objetivo de cada individuo, esto es, al cálculo de la fase forward del MLP para cada muestra del conjunto de entrenamiento, y para cada uno de los individuos de las poblaciones Px y Pu. Entonces, surge la idea de relevar a la FPGA de realizar la etapa de mutación y recombinación, no implementado el kernel en cuestión, y por lo tanto, liberando recursos; y en su lugar, implementar uno o más caminos de datos adicionales para cómputo del kernel perceptrón (número de SIMD mayor a 1). Con ello, se espera mejorar la aceleración en el cálculo de la función objetivo, y por ende mejorar los resultados de aceleración global.

Esta propuesta se presenta para ser desarrollada sobre la misma plataforma DE1-SoC, considerando que en la misma se cuenta con memoria compartida entre el host y el device, que facilita la implementación de la mutación y recombinación del lado del host; y que además, los recursos de la FPGA se encuentran ocupados casi al 100% sólo con la implementación del kernel perceptrón y el de mutación y recombinación. Por ello, no es posible utilizar más de un camino de datos en el kernel del perceptrón si se utiliza el kernel de mutación y recombinación.

Por otro lado, para desarrollos que consideren FPGAs objetivo de mayores dimensiones se presenta la siguiente propuesta.

Multiplicación del SIMD en otras plataformas

Nuevamente, considerando los resultados temporales obtenidos y considerando implementaciones sobre plataformas basadas en FPGAs de mayores dimensiones, se propone aumentar la cantidad de caminos de datos para el cómputo del kernel del perceptrón, y en estos casos, manteniendo los kernels asociados a la mutación y recombinación, pudiendo hasta incorporar el kernel que computa la selección. El objetivo sería mejorar los resultados de aceleración en el cálculo de la fase forward de las redes individuos del algoritmo DE.

Integración del generador de números pseudo-aleatorios como módulo IP

Sería también interesante poder incorporar el generador de números pseudo-aleatorios en OpenCL como una función auxiliar, pero a través de su descripción en HDL como un módulo IP, y no por medio de una descripción a alto nivel realizada en OpenCL C. Esta implementación podría simplificar el uso del generador (no se requeriría el uso de un buffer de memoria para mantener la semilla entre ejecuciones de los kernels), y también se podrían implementar otros generadores, tal como el generador LAPAR (*Leap-Ahead Paralelo*), descrito en [26], el cual es un generador de mayores prestaciones y mejor calidad que el implementado en este trabajo (el LA3217), con apenas un mayor coste de recursos hardware.

Evaluación del desempeño de las redes entrenadas con el algoritmo DE

Aunque los errores obtenidos a partir del cómputo de los estímulos patrones se reducen casi a cero con redes entrenadas con el algoritmo DE, es importante realizar un análisis de las capacidades de generalización de las redes obtenidas.

Como se sabe, cuando la red se entrena con el algoritmo RPB, puede existir un sobre entrenamiento de la red, en la cual, la red 'aprende' demasiado bien en conjunto de entrenamiento, pero luego no es capaz de extrapolar nuevos resultados de forma correcta. Surge entonces la duda de si este mismo comportamiento podría presentarse para redes cuyos pesos sinápticos se hayan echo evolucionar con el algoritmo DE, por lo que realizar un análisis de las capacidades de generalización es muy importante.

Análisis de los pesos de las diferentes redes evolucionadas

Se sabe que, generación a generación, los pesos de las NP redes padres evolucionan hacia valores que producen que el valor de error se minimice (valor de error correspondiente con la definición de la función objetivo implementada en este trabajo). Sin embargo, resulta interesante evaluar la similitud entre los pesos de estas NP redes, y realizar un análisis detallado, pretendiendo determinar si los pesos tienden a converger a los mismos valores finales, o si por el contrario cada individuo converge en un conjunto de pesos muy diferente de los demás.

Este análisis complementaría el análisis propuesto anteriormente, lo que concluiría en un estudio de las sobre características de generalización y la similitud entre las redes evolucionadas con el empleo del algoritmo DE como método de entrenamiento.

Mejoras en los accesos a memoria

Aunque la plataforma DE1-SoC no lo soporta, es posible implementar memoria heterogénea respecto a los datos con los que trabajan los kernels, esto es: memoria secuencial (DDR), o memoria de acceso aleatorio (QDR). Con ello, puede mejorarse la velocidad con que se realizan los accesos a memoria, y por ende, mejorar la performance del sistema.

Otra posibilidad de mejorar los accesos a memoria, tiene que ver con el acceso a posiciones consecutivas de memoria por parte de cada una de las instancias de ejecución de los kernels. Esta idea nace de la elaboración del profile del comportamiento en ejecución de los kernels, en donde se observó que las ráfagas de lectura/escritura sobre memoria global se realizaban de a una palabra por vez, cuando la ráfaga como tal soporta hasta 16 palabras en una misma transferencia.

Para implementar cualquiera de estas dos mejoras, puede referirse [29].

Modificaciones del algoritmo DE

Como se ha mencionado en el capítulo de la introducción, existen modificaciones sobre la base del algoritmo DE que pretenden mejorar la convergencia o la calidad de los resultados conseguidos utilizando este algoritmo. Por ello, resulta atractivo implementar algunas alternativas y analizar los resultados obtenidos.

Integración de toda la plataforma sobre el sistema de optimización

Este trabajo se desarrolló con perspectivas para ser integrado en el proceso de computación distribuida basada en dispositivos FPGAs, para la optimización de la topología de redes neuronales artificiales. Por ello, la integración del desarrollo sobre un elemento de cómputo en el sistema distribuido definiría un objetivo importante para un posterior desarrollo, en el cual se deben integrar las funcionalidades de red y se debe establecer la comunicación con el host principal del algoritmo evolutivo general (el encargado de controlar la evolución de las topologías y de enviar los parámetros de ejecución de la red a implementar).

Anexo I

Funciones auxiliares de OpenCL para la generación de números aleatorios

La siguiente tabla resume la descripción en OpenCL C del comportamiento de la función que emula el funcionamiento del generador de números aleatorios LA3217 (*Linear Feedback Shift Register Leap A-head, with 32 registers and 17 predicted leaps*) y de otra función adicional que permite la normalizar el entero sin signo retornado por el generador, a un valor entre 0 y 1 del tipo *float*.

```
/// Funcion de normalizacion (valor entre 0 y 1) del valor aleatorio entero (unsigned)
float rndNorm(unsigned in){
    return (float)in/(4294967295);  // CL_UINT_MAX = 0xFFFFFFFF = 4294967295U
}

/// Funcion que retorna un valor aleatorio entre 0 y CL_UINT_MAX = 0xFFFFFFFF = 4294967295U
unsigned rndGenU(unsigned in){
    // Obtencion de los 32 bits del vector de entrada "in":
    const unsigned char x0  = (in & 0x00000001)>>0;
    const unsigned char x1  = (in & 0x00000002)>>1;
    const unsigned char x2  = (in & 0x00000004)>>2;
    const unsigned char x3  = (in & 0x00000008)>>3;
    const unsigned char x4  = (in & 0x00000010)>>4;
    const unsigned char x5  = (in & 0x00000020)>>5;
    const unsigned char x6  = (in & 0x00000040)>>6;
    const unsigned char x7  = (in & 0x00000080)>>7;
    const unsigned char x8  = (in & 0x00000100)>>8;
    const unsigned char x9  = (in & 0x00000200)>>9;
    const unsigned char x10 = (in & 0x00000400)>>10;
    const unsigned char x11 = (in & 0x00000800)>>11;
    const unsigned char x12 = (in & 0x00001000)>>12;
    const unsigned char x13 = (in & 0x00002000)>>13;
    const unsigned char x14 = (in & 0x00004000)>>14;
    const unsigned char x15 = (in & 0x00008000)>>15;
    const unsigned char x16 = (in & 0x00010000)>>16;
    const unsigned char x17 = (in & 0x00020000)>>17;
    const unsigned char x18 = (in & 0x00040000)>>18;
    const unsigned char x19 = (in & 0x00080000)>>19;
    const unsigned char x20 = (in & 0x00100000)>>20;
    const unsigned char x21 = (in & 0x00200000)>>21;
    const unsigned char x22 = (in & 0x00400000)>>22;
    const unsigned char x23 = (in & 0x00800000)>>23;
    const unsigned char x24 = (in & 0x01000000)>>24;
    const unsigned char x25 = (in & 0x02000000)>>25;
    const unsigned char x26 = (in & 0x04000000)>>26;
    const unsigned char x27 = (in & 0x08000000)>>27;
    const unsigned char x28 = (in & 0x10000000)>>28;
    const unsigned char x29 = (in & 0x20000000)>>29;
    const unsigned char x30 = (in & 0x40000000)>>30;
    const unsigned char x31 = (in & 0x80000000)>>31;
```

```
    unsigned out = 0;

    // Computacion del nuevo vector aleatorio
    out |= ( x9 ^ x13 ^ x15 ^  x8) << 0;  // Bit #0
    out |= ( x9 ^ x10 ^ x14 ^ x16) << 1;  // Bit #1
    out |= (x10 ^ x11 ^ x15 ^ x17) << 2;  // Bit #2
    out |= (x11 ^ x12 ^ x16 ^ x18) << 3;  // Bit #3
    out |= (x12 ^ x13 ^ x17 ^ x19) << 4;  // Bit #4
    out |= (x13 ^ x14 ^ x18 ^ x20) << 5;  // Bit #5
    out |= (x14 ^ x15 ^ x19 ^ x21) << 6;  // Bit #6
    out |= (x15 ^ x16 ^ x20 ^ x22) << 7;  // Bit #7
    out |= (x16 ^ x17 ^ x21 ^ x23) << 8;  // Bit #8
    out |= (x17 ^ x18 ^ x22 ^ x24) << 9;  // Bit #9
    out |= (x18 ^ x19 ^ x23 ^ x25) <<10;  // Bit #10
    out |= (x19 ^ x20 ^ x24 ^ x26) <<11;  // Bit #11
    out |= (x20 ^ x21 ^ x25 ^ x27) <<12;  // Bit #12
    out |= (x21 ^ x22 ^ x26 ^ x28) <<13;  // Bit #13
    out |= (x22 ^ x23 ^ x27 ^ x29) <<14;  // Bit #14
    out |= (x23 ^ x24 ^ x28 ^ x30) <<15;  // Bit #15
    out |= (x24 ^ x25 ^ x29 ^ x31) <<16;  // Bit #16
    out |= ( x0                  ) <<17;  // Bit #17
    out |= ( x1                  ) <<18;  // Bit #18
    out |= ( x2                  ) <<19;  // Bit #19
    out |= ( x3                  ) <<20;  // Bit #20
    out |= ( x4                  ) <<21;  // Bit #21
    out |= ( x5                  ) <<22;  // Bit #22
    out |= ( x6                  ) <<23;  // Bit #23
    out |= ( x7                  ) <<24;  // Bit #24
    out |= ( x8                  ) <<25;  // Bit #25
    out |= ( x9                  ) <<26;  // Bit #26
    out |= (x10                  ) <<27;  // Bit #27
    out |= (x11                  ) <<28;  // Bit #28
    out |= (x12                  ) <<29;  // Bit #29
    out |= (x13                  ) <<30;  // Bit #30
    out |= (x14                  ) <<31;  // Bit #31

    return out;
}
```

Bibliografía

[1] Rafaél Gadea Gironés and Jorge Fe, Acceleration of optimization of neural networks throught on SoC with OpenCL, 2016

[2] Jorge D. Fe, Ramon J. Aliaga, and Rafael Gadea-Gironés, Evolutionary Optimization of Neural Networks with Heterogeneous Computation: Study and Implementation, 2015

[3] W. S. McCulloch and W. Pitts, A logical calculus of ideas immanent in nervous activity, 1943

[4] S. Haykin, Neural Networks: A Comprehensive Foundation, 1995

[5] T. Kohonen, Self-organization and associative memory. , 1989

[6] J. R. Hilera and V. j. Martinez, Redes Neuronales Artificiales: Fundamentos, Modelos y Aplicaciones, 1995

[7] Geoffrey Hinton et al., Deep Neural Networks for Acoustic Modelingin Speech Recognition, 2012

[8] THE NEURAL NETWORK ZOO, http://www.asimovinstitute.org/neural-network-zoo/

[9] JARMO ILONEN, JONI-KRISTIAN KAMARAINEN and JOUNI LAMPINEN, Differential Evolution Training Algorithmfor Feed-Forward Neural Networks, 2003

[10] Adam Slowik and Michal Bialko, Training of artificial neural networks using differential evolution algorithm, 2008

[11] Musrrat Ali, Millie Pant and Ajith Abraham, Simplex Differential Evolution, 2009

[12] Kenneth Price, An Introduction to DE, 1999

[13] Deepak Dawar, Evolutionary Computation and Applications, 2015

[14] Differential Evolution (DE) for Continuous Function Optimization, http://www1.icsi.berkeley.edu/~storn/code.html

[15] Página Web de Xilinx, www.xilinx.com

[16] Página Web de Altera, www.altera.com

[17] Khronos OpenCL Working Group, The OpenCL Specification, 2009

[18] DE1-SoC Board on Terasic web page, https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=836&PartNo=5

[19] S. Curteanu and H. Cartwright, Neural networks applied in chemistry. I. Determination of the optimal topology of multilayer perceptron neural networks, 2011

[20] Md Monirul Islam, Md Abdus Sattar, Md Faijul Amin, Xin Yao, and Kazuyuki Murase, A New Adaptive Merging and Growing Algorithm for Designing Artificial Neural Networks, 2009

[21] K. H. Han and J. H. Kim, Quantum-inspired evolutionary algorithms with a new termination criterion,H-epsilon gate, and two-phase scheme, 2004

[22] X. Yao, Evolving artificial neural networks, 1999

[23] X. Yao and Y. Liu, A new evolutionary system for evolving artificial neural networks, 1997

[24] Fernando Mateo, Dusan Sovilj, and Rafael Gadea-Girones, Approximate k-NN delta test minimization ´method using genetic algorithms: Application to time series, 2010

[25] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter, Outlier Detection

Using Replicator Neural Networks. In In Proc. of the Fifth Int, 2002

[26] Claudio M. Gonzalez, Carlos A. Gayoso, Leonardo J. Arnone y Miguel R. Rabini, Implementacion de generadores de números pseudoaleatorios utilizando registros dedesplazamiento realimentados leap ahead, 2012

[27] Altera, Altera SDK for OpenCL - Programming Guide, 2016

[28] Mauricio Raúl Palavecino Nicotra, Implementación de OpenCL en DE1-SoC, 2016

[29] Altera, Altera SDK for OpenCL - Best Practices Guide,