HLSTL Biblioteca para programación genérica sobre FPGA con HLS

Manuel J. Abaldea, [1] Jesús Barba Romero, Julián Caba Jiménez, Fernando Rincón Calle, Julo Dondo Gazzano y Juan Carlos López López

Resumen— **La evolución de los flujos de diseño sobre FPGAs lleva hoy en día a la utilización de técnicas y herramientas basadas en HLS (*High Level Synthesis*). A través de la elevación del nivel de abstracción de las especificaciones del hardware mediante el uso de lenguajes de programación populares como ANSI C o C++, se consigue una reducción en tiempo y complejidad del ciclo de desarrollo en dispositivos de lógica reconfigurable.**

Sin embargo, la reutilización de código heredado, como base para los modelos de referencia, es aún un proceso manual que requiere de ciertos conocimientos de la arquitectura destino y habilidades que permitan aprovechar al máximo el potencial de las herramientas HLS. Con el objetivo de reducir esa brecha y facilitar la reutilización de código no escrito específicamente para HLS, se plantea en este trabajo una biblioteca para HLS que permita trabajar de manera transparente con estructuras de datos complejas. Basándose en los principios de programación genérica, la interfaz de alto nivel abstrae al desarrollador de los detalles de implementación y optimizaciones de bajo nivel, permitiéndole centrarse en el desarrollo de la solución.

Palabras clave— **FPGA, HLS, estructura de datos, grafo, hardware reconfigurable, reutilización.**

I. Introducción

Si hablamos de aportar soluciones hardware a medida para cualquier problema de computación las ventajas son claras: mayor rendimiento y/o menor consumo si las comparamos con su equivalente software. Típicamente, estas soluciones se basan en la programación de un sistema basado en un solo microprocesador, un conjunto de elementos de procesamiento (por ejemplo, sistemas multiprocesador o arquitecturas masivamente paralelas como las GPU *Graphics Processing Unit*). Pero los hándicaps de un desarrollo hardware también son bien conocidos; el elevado coste de su desarrollo y la difícil reutilización de dicha solución en otros sistemas. Estas características, por lo tanto, hacen que este tipo de diseños sólo son atractivos cuando el volumen de ventas sea elevado.

Como respuesta a los principales inconvenientes del diseño y desarrollo de sistemas hardware a medida surgen las FPGAs (*Field Programmable Gate Array*). Una FPGA es un dispositivo lógico de puertas reconfigurables que hacen posible el desarrollo hardware en una plataforma reprogramable. Una solución basada en FPGA proporciona un rendimiento próximo al hardware hecho a medida, pero sin los desorbitados costes de fabricación e ingeniería. Sin embargo, no hasta hace mucho, el diseño para FPGAs utilizaba unos lenguajes y flujos de trabajo casi inaccesibles para alguien sin conocimientos profundos de la arquitectura y los modelos de computación normalmente asociados al diseño de circuitos lógicos.

Sin duda, los flujos de trabajo basados en técnicas de HLS (*High Level Synthesis*) supusieron un gran avance en el diseño de componentes para FPGAs respecto a las metodologías y lenguajes de descripción de hardware (HDL) existentes en los años 80 y 90. Actualmente, el uso de herramientas HLS permite al desarrollador elevar el nivel de abstracción de la especificación de la funcionalidad del hardware mediante el uso de lenguajes de programación de alto nivel como ANSI C o C++. Este paso, permite reducir significativamente la complejidad del desarrollo para FPGA y aumenta la reutilización de componentes gracias a la posibilidad de portar el código de las especificaciones. Todo esto se traduce en: (a) un menor coste del desarrollo (ciclos de trabajo más cortos por la automatización de múltiples procesos) y; (b) apertura del uso de FPGAs en nuevos ámbitos.

No obstante, a pesar de la mejora que supone la adopción de flujos de trabajo basados en HLS, su uso no es tan directo y eficiente como cabría esperar. Por un lado, el desarrollador HLS aún debe conocer y dominar conceptos clave del lenguaje de especificación, comportamiento de la herramienta de síntesis y la arquitectura de plataforma destino para poder alcanzar las restricciones impuestas por el diseño. Por otro lado, en proyectos complejos donde lo habitual es utilizar código heredado, nos encontramos con que dicho código no es directamente sintetizable por la herramienta. Aquí entra en juego la experiencia del desarrollador HLS para aplicar las transformaciones necesarias en el modelo de referencia usado, con el fin de hacerlo compatible con la herramienta de síntesis. Aquí se aplicarían diferentes estilos de programación y optimizaciones (en forma normalmente de directivas) con el fin de obtener la mejor solución con un uso racional de recursos.

En el ámbito de la ingeniería del software, es habitual la utilización de bibliotecas de rutinas que proporcionan una manera estandarizada de acceder a cierta funcionalidad recurrente en determinados dominios mediante un API (Application Programming Interface) bien definida. El nivel de abstracción de las primitivas ofrecidas depende del dominio de aplicación, pudiéndose construir bibliotecas sobre otras soluciones ya existentes. Mediante el uso de este tipo de bibliotecas, se favorece que el diseñador se centre en la especificación de la solución de su problema, sin necesidad de desarrollar nuevas soluciones (una más) a problemas recurrentes.

[1] Grupo ArCo, Escuela Superior de Informática, Universidad de Castilla la Mancha, e-mail: manueljose.abaldea@uclm.es

La aplicación del concepto de "componentización" del desarrollo hardware no es nuevo, pero sí que adquiere una nueva dimensión en el ámbito de diseño HLS. Con este paradigma, se abre la posibilidad de desarrollar bibliotecas de especificaciones de componentes alto nivel para facilitar su reutilización en futuros diseños.

En este trabajo, se presenta una prueba de concepto de una biblioteca para HLS que permite incorporar, de una manera sencilla, rápida y transparente, estructuras de datos básicas en multitud de tipos de problemas. En concreto, el caso de uso descrito se centra en la implementación en HLS de una interfaz para trabajar con grafos, que abstrae al desarrollador de los detalles concernientes a la gestión de la información y elementos de la macroarquitectura.

Esta prueba de concepto es el primer paso de una propuesta de mayor alcance que pretende ofrecer soluciones en el campo de HLS que faciliten la portabilidad de código heredado y la implementación en FPGAs de algoritmos que hagan uso de estructuras de datos dinámicas; como los existentes en aplicaciones de IA (Inteligencia Artificial), por ejemplo.

Para ello, esta propuesta se inspira en soluciones suficientemente probadas y maduras en el mundo del software, que aportan las siguientes ventajas:

- Reutilización y portabilidad: Interfaz genérica para trabajar con las estructuras de datos más comunes y sus principales algoritmos asociados.
- Implementación óptima: Abstracción sin pérdida de eficiencia.

De este modo, podemos encontrar bibliotecas para diferentes lenguajes y entornos que materializan estas características mediante la utilización de una serie de artefactos y patrones de diseño: contenedores, algoritmos, iteradores y funciones. Ejemplos de bibliotecas para programación genérica son la *Standard Template Library* para C++ o *Java Collection Framework* para Java y la parte correspondiente a la definición y gestión de los tipos de datos presente en la *GNOME Library* para lenguaje C.

En la sección II expondremos algunos artículos que comparten varias de las ideas que presentaremos así como la necesidad de una solución de la misma índole de la que se pretende aportar en este artículo. Continuaremos en la sección III exponiendo el concepto más detallado de nuestra propuesta y en IV explicaremos el modelo implementado como prototipo para realizar la prueba de concepto. Finalmente en la sección V expondremos las conclusiones obtenidas y los trabajos futuros.

II. Estado del Arte

A continuación, se recogen diferentes trabajos que desarrollan diferentes enfoques acerca del diseño, implementación y uso de estructuras de datos en FPGAs.

La utilidad y el alto rendimiento que se puede alcanzar con una implementación sobre una plataforma FPGA de algoritmos paralelizables que hacen uso de estructuras de datos complejas, es evidente. Se identifican las aplicaciones y usos de los mismos, así como las técnicas de mejora y optimización (tanto en rendimiento como utilización de recursos de memoria, fundamentalmente).

Por ejemplo, Li y Bermak [1] utilizan un árbol de decisión como base para la implementación de un modelo de clasificación. Dicho árbol es implementado en un lenguaje de descripción de hardware de bajo nivel como Verilog HDL. Obtienen un ahorro de hasta el 80% en recursos con respecto a otras implementaciones. Un gran trabajo difícilmente reutilizable en otro ámbito fuera de la identificación de gases.

Narayanan en [2] presenta también la implementación de árboles de decisión sobre lógica reconfigurable. En esta ocasión con el objetivo de mejorar la implementación de un algoritmo común de minería de datos para el cálculo del coeficiente de Gini. La implementación llega a alcanzar resultados 5 veces más rápidos que una implementación de referencia trabajando con un índice 16 de Gini.

El desarrollo de este tipo de soporte en FPGA para la implementación de estos algoritmos en hardware, no solo ofrece un mayor rendimiento sino que también, como bien referencia [1], permite crear un sistema más eficiente en cuanto a consumo. En [3] podemos contrastar el rendimiento y la eficiencia energética del desarrollo de un estimador de función matricial estocástica en hardware. En este caso, la arquitectura y recursos necesarios para su implementación son modelados con Altera OpenCL HLS.

Lo que tienen en común estos tres trabajos, es que ofrecen una solución muy efectiva, pero hecha a medida para el dominio del problema en el que se ubican, lo que las hace complicadas de portar a otras plataformas.

Con el objetivo de favorecer la reutilización de diseños y, de esta manera, aprovechar el esfuerzo previo en futuros proyectos, en [4] se presenta un entorno de trabajo de bajo nivel para el desarrollo de grafos en FPGA. Esta propuesta, si bien reduce el esfuerzo de implementación en el caso concreto de manejo de grafos, necesita de la implementación de dos núcleos de comunicación específicos cada vez que queramos utilizarlo en una nueva plataforma.

Otro ejemplo de mejora de la reusabilidad en el campo que nos compete es [5], que propone un entorno de trabajo centrado en la aceleración de algoritmos que usen grafos de gran tamaño. Al igual que en el caso anterior sería necesaria una tarea de implementación a mano de la lógica de comunicación para poder reutilizar componentes en un nuevo proyecto.

En cuanto a la utilización de herramientas y técnicas de HLS para el diseño e implementación de aplicaciones que hagan uso de estructuras de datos complejas, cabe destacar la propuesta de [6]. En ella se puede comprobar la utilidad e importancia de facilitar soporte para estructuras de datos dinámicas en desarrollos hardware HLS. En este trabajo los autores implementan la lógica que se encarga de ges-

tionar los punteros en hardware, emulando el comportamiento del API de los sistemas software para el trabajo con memoria dinámica. Esta solución, aún cuando ofrece unos buenos resultados, ofrece una interfaz de bajo nivel y limitada a un modelo en concreto de gestión de memoria, lo que la hace poco flexible para ser adaptada y para optimizar su uso en determinadas circunstancias.

Otra propuesta interesante es la planteada en [7], donde se introduce una biblioteca de código abierto escrita en HLS para enlazar estructuras como árboles binarios, tablas hash y vectores. Junto a esta biblioteca, se ofrece una herramienta de análisis con la que evaluar el rendimiento de dichas estructuras en hardware y comparar los resultados para diferentes plataformas. No obstante, la propuesta de [7] adolece de la misma debilidad que [6], ambas se centran en ofrecer un conjunto de primitivas que emulan una interfaz software para la gestión de memoria dinámica. Si bien no se pone en duda su necesidad y utilidad de estos artefactos para la implementación de estructuras de datos dinámicas y complejas, sería deseable tener una herramienta que ayudara al diseñador a elevar el nivel de abstracción.

De esta manera, el planteamiento que se presenta en este artículo se basa en un enfoque generalista y de más alto nivel. Se ofrece al desarrollador las herramientas necesarias para trabajar directamente a nivel de la estructuras de datos, aislándole de la problemática de una gestión óptima de los recursos. Otra característica de este trabajo es su flexibilidad y portabilidad, ofreciendo soluciones que no están ligadas a ningún problema o tecnología concreta. Para ello, hace uso de un C básico y de elementos del lenguaje sencillos que aseguran la sintetizabilidad del modelo y su reutilización en diferentes entornos y herramientas de HLS, ayudando al desarrollador a no preocuparse de las conocidas limitaciones de las herramientas HLS [8].

III. Concepto

Es habitual, para cualquier tipo de desarrollo (tanto software como hardware), que partes del sistema sean reutilizadas, provenientes de otros proyectos.

En el ámbito del software es muy común utilizar bibliotecas con funciones matemáticas, con algoritmos básicos (por ejemplo, ordenación o clasificación) o para trabajar con estructuras de datos, como el caso tratado en este artículo, son algunos ejemplos de los diversos usos. En cambio, en el desarrollo hardware la reutilización que se puede hacer de componentes heredados de terceros no es tan directa. En muchas ocasiones es necesario llevar a cabo un sobreesfuerzo con el fin de adaptar la especificación del IP (*Intellectual Property*) a la plataforma y/o tecnología destino. Además, en muchas ocasiones las especificaciones del componente no se adaptan a los requisitos del nuevo sistema, por lo que es necesario un trabajo de re-ingeniería.

Con la utilización de entornos de desarrollo HLS para componentes hardware, se reduce la cantidad de esfuerzo necesario para hacer usable en un nuevo proyecto un desarrollo anterior. Sin embargo, los flujos de trabajo HLS obligan en muchas ocasiones a partir de especificaciones del problema y/o algoritmos de referencia que no provienen de proyectos HLS anteriores. Esa es una de las claves del éxito y crecimiento de la utilización de este tipo de herramientas: la posibilidad de partir de código C/C++ escrito por desarrolladores no necesariamente de HLS.

Llegados a este punto, es habitual encontrarse con una serie de inconvenientes derivados de las restricciones y estilos de programación que imponen los entornos de desarrollo HLS para dispositivos de lógica reconfigurable.

La finalidad última de esta propuesta es la de proveer a los desarrolladores HLS de una serie de herramientas que les faciliten el desarrollo de aplicaciones/componentes, que hacen uso intensivo de estructuras de datos dinámicas, minimizando el esfuerzo a realizar para adoptar código de terceros no necesariamente escrito para HLS. Un efecto secundario de esta propuesta es que también, para desarrolladores de código que no dispongan necesariamente de la formación y conocimiento de plataformas FPGA y HLS, se relajan las barreras de entrada a este tipo de herramientas, abriendo una puerta para la generalización del uso de la síntesis de alto nivel en múltiples disciplinas.

Las razones detrás de la decisión de trabajar con estructuras de datos dinámicas se fundamentan en el gran auge y popularidad que recientemente ha adquirido todo aquello que tiene que ver con el uso de técnicas de inteligencia artificial. Así, modelos de clasificación, procesos estocásticos, minería de datos o sistemas de aprendizaje, son familias de algoritmos que hacen uso intensivo de estos artefactos.

Es fácil comprobar cómo, en el desarrollo de todos estos sistemas, una parte clave es la elección de las estructura de datos idóneas para abordar el algoritmo en cuestión; en la mayoría de los casos grafos o árboles.

Con esta idea arranca nuestra propuesta, en este trabajo se propone el desarrollo de una biblioteca sintetizable para HLS que proporcione soporte para las estructuras de datos dinámicas más utilizadas y las funcionalidades necesaria para su manejo. Se tiene siempre en cuenta que el resultado obtenido sea lo más genérico posible para maximizar su reutilización y optimización desde el punto de vista del rendimiento y uso de recursos.

Según la información de que disponemos, actualmente no se dispone de facilidad alguna como la que se plantea en este trabajo. La propuesta comprende el desarrollo e implementación de módulos parametrizables que implementen el comportamiento de estructuras de datos complejas y dinámicas (listas, árboles y grafos) inspirados en soluciones probadas provenientes del mundo software. Se busca la compatibilidad con bibliotecas existentes para programación genérica, ampliamente utilizadas en desarro-

llos C/C++ para los campos de aplicaciones anteriormente mencionados.

Con todo ello los desarrolladores dispondrán de una biblioteca con componentes configurables que, mediante la utilización de herramientas de síntesis de alto nivel, generen una infraestructura de soporte para la implementación de algoritmos más complejos.

A. STL como referente

Son varias las bibliotecas de las que disponemos con soporte para el manejo simplificado y eficiente de estructuras de datos dinámicas. Y sería estupendo si, las herramientas HLS pudieran incorporar directamente dicha funcionalidad a los proyectos de desarrollo de componentes hardware. Sin embargo, esta migración no es posible directamente ya que en dichas bibliotecas el código asociado expone una serie de características generalmente no soportadas por las actuales herramientas HLS derivadas de la naturaleza de los recursos de lógica reconfigurable, a saber:

- Utilización de memoria dinámica y la pila.
- Utilización de las llamadas al sistema (necesidad de soporte por parte del sistema operativo).
- Uso de operaciones de casting de puntero "arbitrario".
- Implementaciones que usan recursividad "arbitraria".

Como referente, hemos tomado la biblioteca STL ya que es la colección de estructuras de datos genéricas y algoritmos escritos en C++ más utilizada por la comunidad de desarrolladores y sobre todo por ser la adoptada de forma oficial por el comité ANSI de estandarización de C++.

La biblioteca STL es un buen referente dado su buen rendimiento para manejo de estructura de datos y de algoritmos complejos. Si nos fijamos en cómo está implementada, en su código fuente es fácil comprobar la complejidad del desarrollo, lo cual contrasta con la la simplicidad de manejo que ofrece gracias al alto nivel de abstracción de las interfaces disponibles. STL proporciona al desarrollador un conjunto de clases que permiten dar soporte a diferentes características, como la gestión de memoria, de un modo completamente transparente desde el punto de vista del desarrollador. También mejora aspectos del propio lenguaje C++ definidos por el estándar como, por ejemplo, la mayor precisión en aritmética de punto flotante. Además, STL proporciona las que quizás sean su características más visible; una serie de artefactos que hacen realidad un modelo de programación generando: (1) un conjunto de contenedores que son estructuras de datos que almacenan y permiten manipular otras estructuras de datos; (2) iteradores para acceder a los elementos de los contenedores y; (3) algoritmos para llevar a cabo operaciones sobre estos.

Uno de estos tres pilares sobre el que se construye el modelo de la STL, los iteradores, se podría considerar el elementos clave alrededor del cual se puede construir cualquier tipo de algoritmo genérico. Los iteradores nos permiten retornar la referencia a memoria de un elemento de una estructura de datos. Funcionalmente, un iterador es equivalente a un puntero y podemos utilizar cualquier operador estándar (por ejemplo it++, *it o it[n]). Ambos son totalmente compatibles y, por lo tanto, cualquier algoritmo que utilice punteros también se puede desarrollar utilizando iteradores. Pero los iteradores van más allá y nos pueden proporcionar mayor eficiencia que los punteros ya que están divididos en categorías. Para cada categoría se proporciona solamente las operaciones necesarias, de modo que se puede realizar una implementación más segura y eficiente, dichas categorías son: sólo lectura (*input*); sólo escritura (*output*); recorrido sólo hacia adelante (*forward*), bidireccional (*bidirectional*) o aleatorio (*random*).

En la siguiente sección, se presenta la prueba de concepto que pretende incorporar los conceptos de la programación genérica y las características más interesantes tanto de la STL (como de otras bibliotecas de referencia como son Glib o *Java Collection Framework*) al desarrollo con HLS de aplicaciones que hacen uso intensivo de estructuras de datos dinámicas.

IV. Biblioteca para desarrollo genérico en HLS

Como prueba de concepto se ha realizado una implementación de un modelo genérico de una estructura de datos de tipo grafo para Vivado HLS. Los grafos son un tipo de datos ampliamente utilizados en multitud de dominios de aplicación, especialmente en los de interés para este trabajo tal y como se ha mencionado anteriormente. Además, los grafos son una estructura genérica sobre la cual se pueden derivar otras más específicas como son los árboles de diferentes tipos. Esta prueba permite comprender y evaluar la aplicación del concepto de programación genérica en el ámbito del desarrollo para FPGAs, siendo el germen de lo que sería una futura HLSTL (*HLS Template Library*).

El objetivo es desarrollar los conceptos de contenedor e iterador para un caso concreto, que permita obtener unas conclusiones que sirvan de base sobre la que asentar la implementación completa de la biblioteca.

Como se ha mencionado arriba, la estructura de datos elegida es un grafo $G = (V, E)$, siendo V el conjunto de vértices y E un conjunto de aristas que relacionan dichos nodos. Las características de los grafos soportados por esta primera implementación son las siguientes:

- Simple: Para todo $a, b \in V$ solo existe una arista $(a, b) \in E$ que una dichos vértices.
- No dirigido: las aristas son no orientadas.
- Conexo: para cualquier par de vértices $a, b \in V$ existe al menos un camino posible.
- No completo: existen vértices $a, b \in V$ que no están conectados entre sí por ninguna arista.
- No ponderado: las aristas no tienen peso.

Sobre este conjunto de tipos de grafos, las primeras funciones de acceso que se han implementado son de recorrido en anchura y profundidad. A través del proceso de desarrollo y análisis de los resultados de síntesis, se han obtenido datos que permitirán optimizar la implementación de futuras versiones de este módulo en cuanto a optimización de las necesidades de almacenamiento y rendimiento. Las lecciones aprendidas podrán ser aplicadas también al desarrollo de otros tipos de estructuras de datos. En esta primera etapa se ha hecho especial énfasis en el diseño del API, para que existiera un equilibrio entre sencillez y eficiencia, y para que tenga un bajo acoplamiento respecto a los detalles de implementación de bajo nivel (representación de la información del grafo en memoria, sobre todo). Por el momento, no se ha abordado las funciones que tienen que ver con la modificación del grafo en tiempo de ejecución (inserción/eliminación de vértices y aristas) y todos los aspectos relativos a la gestión dinámica de la estructura de datos.

La implementación desarrollada a día de hoy se puede dividir en:

- Elementos base: donde se definen los tipos y estructuras necesarias para almacenar los datos (equivalentes a los contenedores).
- Funciones de acceso básicas: con las cuales se accede a la información almacenada (equivalente a los iteradores).

Sobre estas dos capas, se pueden empezar a programar algoritmos y funciones más avanzadas como las presentadas en la sección V.

A. Elementos base

Dentro de lo que se define como capa de elementos base, se exponen los dos criterios claves que se han tomado a la hora del desarrollo de esta parte que equivaldría a la implementación del contenedor para el grafo.

A.1 Almacenamiento de la información.

Uno de las primeras decisiones a la hora de abordar el problema es el modo en el cual se representará y almacenará la información asociada al tipo de datos concreto. Se establece como requisito que la biblioteca sea capaz de trabajar con grafos de gran tamaño y, por lo tanto, es necesario optimizar la gestión de la memoria necesaria.

Las dos principales maneras de almacenar la información del grafo son:

- Matriz de adyacencia. Se utiliza una matriz donde filas y columnas hacen referencia a los vértices y se almacena en cada posición la información booleana si están conectados o no.
- Lista de adyacencia. Para cada vértice se almacena una lista enlazada con los vértices adyacentes a él, por lo tanto, el tamaño será variable ya que depende del número de aristas.

Las dos opciones presentan sus ventajas. Respecto a acceso, la matriz de adyacencia proporciona una recuperación de la información más rápida y sencilla. Dado un par de nodos a, b de nuestro grafo, el acceso sería una lectura directa a dicha posición de la matriz. Mientras que en una lista de adyacencia tendríamos que buscar para el vértice a, en su lista de vértices adyacentes, si se encuentra el vértice b. El tiempo de acceso, en el peor de los casos sería de $\Theta(gv)$ siendo gv el grado o valencia del vértice, es decir, el número de aristas incidentes a dicho vértice. En relación al espacio requerido, la matriz de adyacencia utiliza un tamaño de $n*n$ siendo n el número de nodos de nuestro grafo. En comparación, la lista de adyacencia ahorra todos los espacios de los nodos no conectados, utilizando un tamaño de $\sum_{a=0}^{n} a+b_a$. En el peor de los casos, trabajando con un grafo completo en el cual todos los vértices estén conectados entre sí, el espacio sería el mismo con la matriz $n*n$.

Como se explica más adelante en la sección V, la interfaz propuesta permite utilizar estas variantes en la implementación de bajo nivel, según las necesidades del desarrollador y/o aplicación, sin tener que cambiar el API de acceso. Se consigue así total transparencia respecto a la tecnología y tipo de componentes (BRAMs, DDR, etc.) utilizada para el almacenamiento de la información.

A.2 Optimizaciones en el almacenamiento.

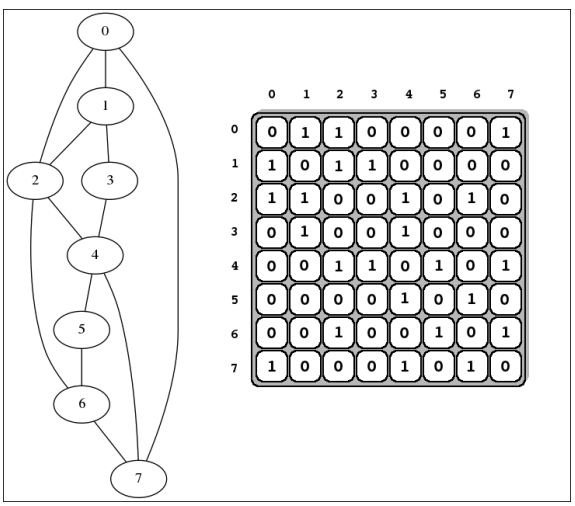

Fig. 1. Grafo y almacenamiento en memoria en matriz de adyacencia.

Existen dos casos en los que el uso de una matriz de adyacencia completa supone una sobrecarga en cuanto a las necesidades de memoria reales para almacenar la información relativa al grafo. Para el caso del que estamos hablando, grafo simple no dirigido, podemos comprobar en la figura 1 como la parte triangular superior y triangular inferior de la matriz nos proporcionan información redundante, de modo que es posible trabajar solamente usando algo más de la mitad de las posiciones de la matriz, concretamente en $(n*(n+1))/2$ posiciones que corresponde con el $\sum_{a=1}^{n} a$. Teniendo esto en cuenta, se puede almacenar toda la información en un vector tal y como

se muestra en la figura 2, al cual llamaremos VMT (vector de matriz triangular), de este modo es posible ahorrar casi la mitad de la memoria inicialmente utilizada en el anterior planteamiento.

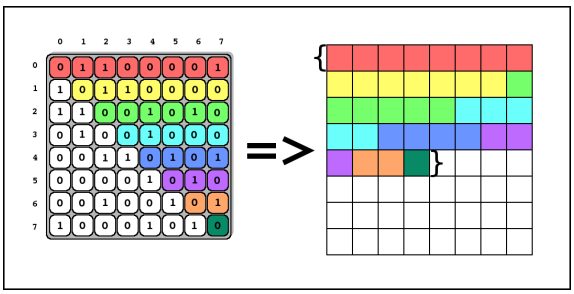

Fig. 2. Almacenamiento en memoria en VMT (Vector de Matriz Triangular).

Esta optimización no implica un aumento en el tiempo de acceso, ya que el cálculo de la dirección de acceso a la memoria puede completarse en un ciclo con la siguiente fórmula:

$$a_v = (a_m*(n-(a_m-1))+((a_m-1)*a_m) >> 1+b_m-a_m)$$

siendo a_m, b_m la posición correspondiente a la matriz de adyacencia y a_v el desplazamiento correspondiente para identificar al dicho elemento dentro del VMT.

La otra optimización posible es el uso de una lista enlazada (lista de adyacencia) para guardar la información de los vértices conectados con otro vértice. Se trata de una de las configuraciones soportadas por la biblioteca para la implementación del contenedor. En este caso, dicha lista es implementada almacenando en la memoria del grafo una serie de tripletas v_o, v_d, w_{ij} donde v_o es el identificador del vértice origen, v_d el identificador del vértice destino y, opcionalmente, w_{ij} que representa el coste de moverse desde v_o a v_d. Así, sólo se guarda la información de las aristas del grafo, lo cual supone un ahorro de recursos especialmente en aquellos casos donde la correspondiente matriz de adyacencia sea dispersa. En el caso de una aplicación que sólo necesite realizar consultas sobre la información del grafo (no modifica su estructura) se podría prescindir del campo v_o y obtener así un ahorro adicional. En cualquier caso, para acceder al primer elemento de la lista, se necesita almacenar la dirección base de comienzo de los nodos hijos en una tabla adicional. Se ha implementado un modelo simplificado de esta lista donde las n primeras posiciones (siendo n el número de vértices) indexan a la primera posición de memoria del propio vector donde se almacenan los identificadores de los vértices conexos como muestra la figura 3.

De este modo solamente se almacena la información de las aristas del grafo, lo cual supone un ahorro de recursos especialmente en aquellos casos donde la correspondiente matriz de adyacencia sea dispersa. No obstante, la tener que almacenar los identificadores de los vértices y las posiciones de memoria no nos basta con 1 bit por posición, en el desarrollo se han utilizado 16 bits.

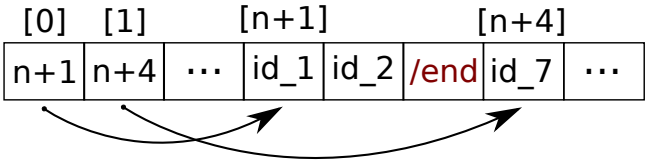

Fig. 3. Almacenamiento en memoria en vector de adyacencia.

Por ejemplo, para una configuración con 1024 vértices, considerando que necesitamos solamente 1 bit para saber si los nodos son conexos o no tendríamos (16 bits en el caso de la lista enlazada):

- Para la matriz de adyacencia $1024*1024$ bits, es decir 128kB.
- Para el VMT $1024*1025/2$ bits, que serían algo más de 64KB.
- Para la lista enlazada, recordamos que el tamaño depende del número de aristas, necesitaríamos menos de 64KB para grafos con menos de 16000 aristas y más de 128KB para grafos con más de 32000 aristas.

Sin embargo, para algoritmos que necesiten modificar la morfología del grado, es necesario plantear otras alternativas que permitan gestionar la (potencial) fragmentación de la información almacenada en memoria por las operaciones de eliminación y inserción de elementos. No es objetivo de este artículo entrar en los detalles de implementación para dar soporte a dicha funcionalidad que se plantean en un trabajo futuro.

B. Funciones de acceso básicas

Para el caso de uso seleccionado, estas funciones son las responsables de recuperar la información correspondiente a un determinado nodo del grafo e implementan el iterador correspondiente para recorrer la lista de nodos conectados.

La figura 4 muestra el código C para HLS de la función que recupera la lista de nodos adyacentes. Dado un identificador de nodo "p", accede a la memoria con toda la información del grafo e inicializa una variable local (iterador de la lista de *hijos*) para su posterior acceso.

Debido a que la forma en la que se ordena la información del grafo "G" depende de su tipo y estrategia de optimización de espacio utilizada, se puede observar en el listado la adaptación de la implementación para cada caso. El desarrollador sólo debe declarar las directivas que desee en un fichero de configuración de cabecera. Es necesario destacar que, en el ejemplo, la interfaz de la función no sufre cambio alguno y es independiente del tipo de grafo instanciado.

Una vez recuperada la lista de nodos adyacentes, el desarrollador obtiene como resultado una estructura (*childs_t*) que contiene tanto el contenedor como la información necesaria para implementar el iterador sobre esa lista. La figura 5 muestra las tres funciones básicas del iterador: *begin* y *end* para definir los límites de acceso al contenedor (vector *childs->list[]*) y *next* para posicionar en el siguiente nodo de la lista.

```
void hlsnode_get_children(graph_t G,nodeid_t p, childs_t *cit)
{
  ...
#ifdef UDIRECTED_GRAPH
  int aux_b = p*(N_NODES-(p-1))+((p-1)*p)>>1;
  int aux_e = (p!=N_NODES-1)?
              ((p+1)*(N_NODES-p)+(p*(p+1))>>1)-1:N_NODES-1;
  L_GET_CHILDS:for (b=p,e=N_NODES-1;
                    b<=e;b++,aux_b++,e--,aux_e--)
  {
    cit->list[b] = G.vmt[aux_b];
    cit->list[e] = G.vmt[aux_e];
#elif DIRECTED_GRAPH
  L_GET_CHILDS:for (b=0,e=N_NODES-1;b<=e;b++,e--)
  {
    cit->list[b] = G.mtx[p][b];
    cit->list[e] = G.mtx[p][e];
#elif ADJACENCY_LIST
 L_GET_CHILDS: for (b=G.nodeinfo[p].firstentry,
                    e=G.nodeinfo[p].lastentry;b<e;b++,e--)
 {
    cit->list[G.adj_list[b].child_id] = 1;
    cit->list[G.adj_list[e].child_id] = 1;
#endif

#ifndef ADJACENCY_LIST
    cit->first = ((cit->list[b]==1) &&
                  (cit->first == NULL_NODE_IT))?b:cit->first;
    cit->last = ((cit->list[e]==1) &&
                 (cit->last == NULL_NODE_IT))?e:cit->last;
  }
#else
  cit->first = G.adj_list[G.nodeinfo[p].firstentry].parent_id;
  cit->last = G.adj_list[G.nodeinfo[p].lastentry].parent_id;
#endif

  ...
}
```

Fig. 4. Función parametrizada para recuperar los nodos adyacentes.

```
nodeit_t hlschildren_begin(childs_t *childs) {
  return childs->first;
}

nodeit_t hlschildren_end(childs_t *childs) {
  if (childs->last == N_NODES-1 ||
      childs->last == NULL_NODE_IT)
    return NULL_NODE_IT;
  else
    return childs->last+1;
}

nodeit_t hlschildren_next(nodeit_t curr,childs_t *childs) {
  nodeit_t next;

  if (curr == childs->last)
    return hlschildren_end(childs);
  L_NEXT_CHILD:for (next = curr+1;(childs->list[next]==0) &&
                                  (next<=childs->last);next++);
  return next;
}
```

Fig. 5. Funciones para el manejo del iterador.

B.1 Optimizaciones en el acceso.

Nótese que en el bucle principal de la función *hlsnode_get_children* se utilizan dos índices para acceder a la memoria. Uno incremental que recorre el vector de adyacencia desde la posición inicial y otro decremental, que lo hace desde la posición final. Esta estrategia pretende aprovecharse de la utilización de memorias de doble puerto o tecnologías de memoria que permiten realizar múltiples accesos simultáneos o casi simultáneos. Estas características suelen ser expuestas a las herramientas de HLS (incluso permiten especificar la tecnología de memoria a utilizar para instanciar una determinada variable), lo que permiten planificar lecturas de dichas memorias de forma concurrente; de este modo, se reducirá el tiempo de recorrido a la mitad.

V. Validación y Conclusiones

Con el objetivo de validar la propuesta de biblioteca para desarrollo genérico en HLS de aplicaciones que utilicen estructuras de datos complejas como son los grafos, se han implementado dos sencillas funciones que recorren la estructura del grafo tanto en profundidad como en anchura. A través de dicha implementación se han podido concretar aspectos clave de diseño de la biblioteca, así como demostrar las capacidades de la misma.

La figura 6 muestra la firma de ambas funciones, así como el código sintetizable.

```
uint32 hlsgraph_traverse_width(graph_t G,nodeid_t start_node,
                               nodeid_t node_list[N_NODES]) {
...
}

uint32 hlsgraph_traverse_depth(graph_t G,nodeid_t start_node,
                               nodeid_t node_list[N_NODES])
{
  static nodeid_t n2visit[N2_VISIT_LIFO_DEPTH];
  nodeid_t *first,*top,current,hijo;
  nodeit_t nit_begin,nit_end,nit;
  static uint1 nvisited[N_NODES];
  childs_t childs;
  nodeid_t i;

  /* Clear temp variables */

  first = top = n2visit;
  *(top++) = start_node;
  i = 0;

  TRAVERSE_W:while (first < top) {
    current = *(--top);

    //Check if already visited
    if (nvisited[current] == 0) {
      node_list[i++] = current;
      //Include in the list of visited nodes
      nvisited[current] = 1;

      hlsnode_get_children(G,current,&childs);
      nit_begin = hlschildren_begin(&childs);
      nit_end = hlschildren_end(&childs);

      for (nit = nit_begin;nit != nit_end;
           nit = hlschildren_next(nit,&childs)) {
          hijo = hlsnodeit_get(nit,&childs);
          //Check if already visited
          if (nvisited[hijo] == 0)
                  *(top++) = hijo;
      }
    }
  }
  return i;
}
```

Fig. 6. Funciones para recorrer el grafo en anchura y en profundidad.

El código hace uso de los artefactos de la biblioteca (contenedores, funciones de acceso e iteradores) de manera que es totalmente independiente de la implementación concreta del grafo que el diseñador haya elegido en función de las necesidades de la aplicación.

Esta primera versión de la biblioteca se ha escrito para ser sintetizable con Vivado HLS de Xilinx, y se han generado diferentes versiones de un com-

ponente IP para su integración y pruebas en la plataforma destino. En este trabajo se ha utilizado una placa ZYNQ ZC702 de Xilinx, sobre la cual se ha construido una infraestructura que permite almacenar la información del grafo indistintamente en Block RAMs o en memoria DDR. La herramienta de síntesis permite generar bien una interfaz genérica de memoria (*ap_memory*) o una interfaz AXI que conecte el IP con un puerto HPC/ACP del PS (*Processor System*) de la plataforma.

En las pruebas realizadas hasta el momento y sobre esta configuración, se han utilizado grafos de hasta 1024 vértices que es el tamaño máximo del grafo que se puede ubicar en un componente de memoria (generado mediante la herramienta Vivado Integrator) al utiliza una matriz de adyacencia. Cuando se utiliza vectores de adyacencia (grafo dirigidos), el número máximo de nodos soportados asciende a 1447. En ambos casos los recursos utilizados de la placa se mantenían por debajo del 3%.

En este trabajo preliminar, las pruebas se han centrado en comprobar cómo desde el punto de vista de un desarrollador hardware, al utilizar bajo Vivado HLS la biblioteca HLSTL nos es posible centrarnos en la aplicación a desarrollar y obviar la gestión de las estructuras necesarias. También cómo la biblioteca optimiza el almacenamiento y acceso a la información sin que haya que reescribir el código de la función a sintetizar. Por lo tanto, puede concluirse que gracias al diseño modular de la biblioteca y el uso de contenedores e iteradores se consigue un API de desarrollo genérico que permite al desarrollador explorar diferentes alternativas y optimizaciones de una manera transparente.

Esta primera versión supone una base sobre la cual desarrollar nuevas características avanzadas, ampliar la colección de estructuras de datos soportadas (colas, listas, árboles, etc.) e introducir la gestión dinámica de las mismas (añadir, modificar y eliminar elementos).

VI. Agradecimientos

Este trabajo ha sido financiado por la Junta de Comunidades de Castilla-La Mancha (proyecto SAND, PEII-2014-046-P) y por el *Ministerio de Economía y Competitividad del Gobierno de España* bajo el proyecto `REBECCA (TEC2014-58036-C4-1-R)`.

Referencias

[1] Qingzheng Li and Amine Bermak, "A low-power hardware-friendly binary decision tree classifier for gas identification," *Journal of Low Power Electronics and Applications*, vol. 1, no. 1, pp. 45–58, 2011.

[2] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An fpga implementation of decision tree classification," in *2007 Design, Automation Test in Europe Conference Exhibition*, April 2007, pp. 1–6.

[3] H. Giefers, P. Staar, and R. Polig, "Energy-efficient stochastic matrix function estimator for graph analytics on fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[4] N. Engelhardt and H. K. H. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[5] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for fpga acceleration of large graph problems: Graphlet counting case study," in *2011 International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–8.

[6] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using vivado hls," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 362–365.

[7] Z. Xue and D. B. Thomas, "Synadt: Dynamic data structures in high level synthesis," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 64–71.

[8] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.