Hybrid Sorting Algorithm Implemented by High Level Synthesis

L. De Micco, M. Acosta, and M. Antonelli

***Abstract*—This paper proposes a hybrid data ordering algorithm which executes serial and parallel instructions. The implementation of the system is presented in the Zedboard development board of Xilinx that includes a SoC (System on Chip). The design was done in high level language HLS (High Level Synthesis). It receives a vector of N elements and delivers the set of indexes of the L major elements ordered. The complexity of the algorithm is analyzed in a generic way. The required times and resources are evaluated and compared with well known sorting algorithms.**

***Index Terms*—Sort algorithm, FPGA implementation, Zedboard, Hardware Accelerator, Compressed Sensing.**

I. Introducción

El ordenamiento de datos es una herramienta básica que actualmente es empleada por una amplia gama de algoritmos, como por ejemplo algoritmos de búsqueda y de optimización. Por ello, disponer de un método de ordenamiento eficiente es muy preciado ya que permite aumentar la velocidad de ejecución de los procesos [1]. La mayoría de los algoritmos de ordenamiento existentes trabajan en forma secuencial, como el método de Burbujeo [2], Árbol cartesiano [3], ordenamiento por Casilleros [4], etc. Por otro lado, existen las llamadas redes de ordenamiento [1], [5], [6] las cuales minimizan el tiempo de ejecución a expensas de una utilización mayor de recursos, en general estas redes están limitadas a pequeños sets de datos a ordenar. El avance de la tecnología dio lugar a dispositivos que permiten ejecutar instrucciones en forma secuencial y concurrente, lo que abre nuevas posibilidades y permite desarrollar algoritmos que combinen estos procesamientos a partir de los algoritmos existentes. Por supuesto, existe siempre un compromiso entre velocidad y consumo de recursos.

La implementación de algoritmos de ordenamiento en FPGA (Field-Programmable Gate Array) ha tenido un creciente interés en los últimos años [7]–[12]. En [7] los autores proponen un método de ordenamiento paralelo de datos basado en redes de ordenamiento implementado en FPGA mediante VHDL (Very High Speed Integrated Circuits Hardware Description Language) que presenta buenos resultados en implementaciones con respecto a tiempos de ejecución para pequeños sets de datos (hasta 2^{10}) y simulaciones para sets más grandes (del órden de 2^{20}). Los trabajos [8] y [9] proponen esquemas de ordenamiento paralelo basados en FPGA, en el primer caso optimizado para cuando se dispone de pocos recursos lógicos y en el segundo caso proporcionan una técnica que mantiene constante el retardo del circuito, independientemente de la longitud del set a ordenar. El trabajo [10] presenta un algoritmo de ordenamiento paralelo recursivo mediante árboles binarios, para lograr la recursividad con lenguaje de descripción de hardware utilizan un modelo de máquina de estado finito jerárquico. Un muy interesante análisis del empleo y ventajas del uso de FPGAs para la implementación de redes de ordenamiento puede encontrarse en [11]. Allí los autores destacan que no siempre una implementación en FPGA supera a la implementación en otros dispositivos como GPUs (Graphics Processing Unit) o microprocesadores multinúcleo, para lograrlo se debe realizar un minucioso diseño y tener en cuenta ciertas consideraciones con respecto a la arquitectura del dispositivo que en el que se realiza la implementación.

En este trabajo se presenta un algoritmo de ordenamiento y los resultados de su implementación. Este algoritmo combina procesamiento concurrente con secuencial a fin de optimizar el ordenamiento de datos, el diseño es parametrizable y permite buscar la mejor relación velocidad-recurso empleados para cada aplicación. El diseño se desarrolla en lenguaje de alto nivel, luego, mediante el uso de directivas, se le indica a la herramienta de síntesis cómo obtener el circuito final.

Este documento está organizado de la siguiente manera: la Sección II presenta el problema y el contexto en el que surge la necesidad del desarrollo. En la Sección III se describe y analiza el algoritmo propuesto. La Sección IV presenta la implementación, los resultados y la validación del sistema. Finalmente en la Sección V se comentan las conclusiones.

II. Planteamiento del Problema

En este caso particular, se requiere un algoritmo de ordenamiento para ser empleado en un sistema de reconstrucción de señales con sensado compresivo caótico [13], [14]. Este sistema permite la reconstrucción de señales a una frecuencia de muestreo 16 veces menor que la de Nyquist, con una degradación controlada bajo parámetros establecidos. En [14] se evalúan distintos mapas caóticos para la generación de secuencias utilizadas por el adquisidor en el proceso de mezclado; con el fin de reemplazar el uso de generadores pseudoaleatorios. En [13], [15] se presenta el sistema completo y se muestran los resultados experimentales obtenidos.

Más específicamente, el sistema de ordenamiento requerido será utilizado por el algoritmo AIHT (Accelerated Iterative Hard Thresholding) que es el encargado recuperar vectores dispersos con una precisión parametrizable [16]. Para acelerar la ejecución de este algoritmo se requiere un acelerador en hardware que efectúe un ordenamiento de datos.

Instituto de Investigaciones Científicas y Tecnológicas en Electrónica (ICYTE), Facultad de Ingeniería, Universidad Nacional de Mar del Plata (FIUNMdP), Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Buenos Aires, Argentina e-mail: ldemicco@fi.mdp.edu.ar (Luciana).

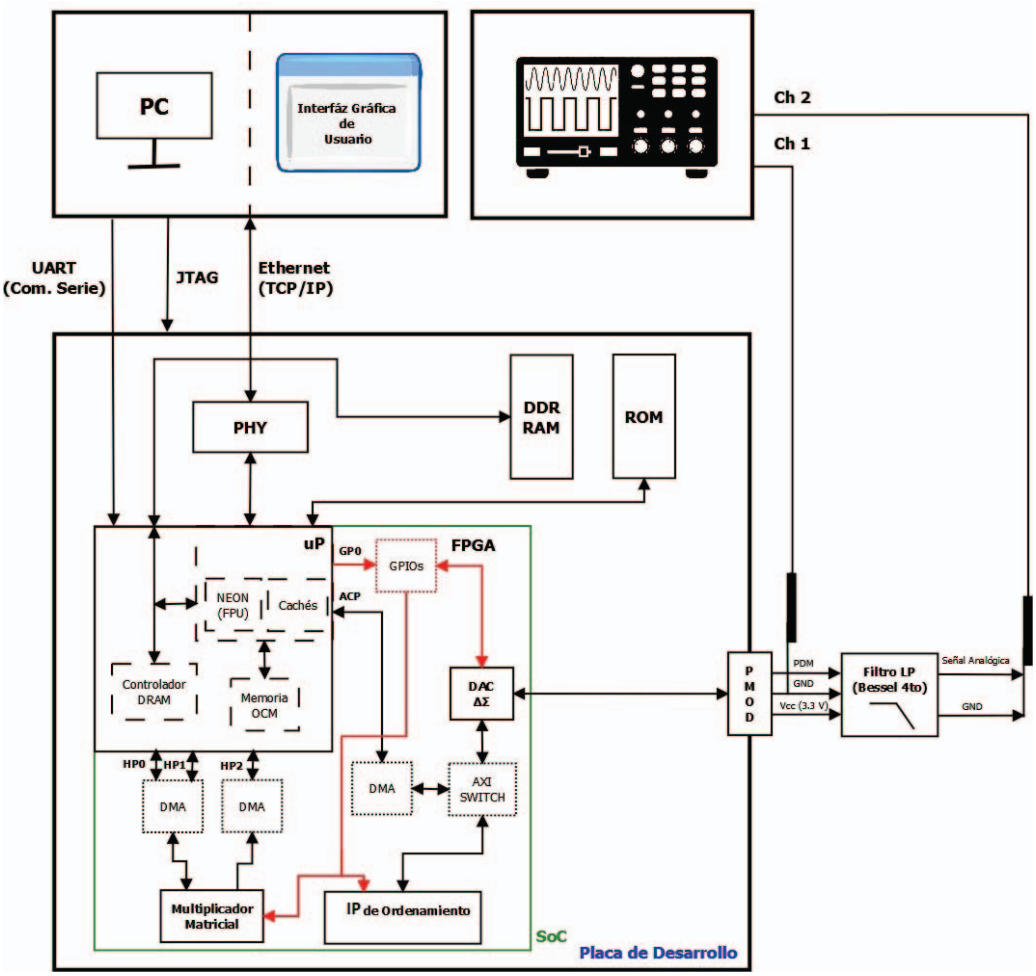


Fig. 1. Sistema de muestreo sub-Nyquist. Implementado en el entorno Vivado de Xilinx, el bloque *IP de ordenamiento* es el acelerador en hardware desarrollado en este trabajo.

La Fig. 1 muestra el sistema de reconstrucción de señales con sensado compresivo caótico completo. Allí puede verse el diagrama en bloques implementado junto con el banco de medición empleado. El sistema se implementa en la placa de desarrollo Zedboard de Xilinx. Esta placa incluye un SoC, el cual permite ejecución secuencial mediante un doble núcleo ARM Cortex-A9 (PS, Processor System) y ejecución concurrente mediante FPGA de Xilinx (PL, Programmable Logic). El bloque *IP de ordenamiento* es el IP (Intellectual Property) desarrollado en este proyecto y se implementa en la zona PL del SoC.

El bloque a desarrollar debe recibir un vector de 1.024 elementos enteros sin signo de 32 bits mediante una interfaz AXI (Advanced eXtensible Interface) Stream [17] y devolver el conjunto de índices de los 32 elementos mayores. La latencia (medida en ciclos de reloj) no debe superar los 2.000 ciclos y la frecuencia de operación del sistema es de $125\ MHz$, además es deseable que emplee la mínima cantidad posible de recursos.

III. Algoritmo Propuesto

El núcleo del algoritmo propuesto emplea la red de ordenamiento en paralelo que se puede ver en la Fig. 2. En la misma se puede apreciar que los elementos se almacenan en un registro de tamaño M y en cada ciclo de reloj se efectúan comparaciones de manera concurrente. En esta red cada elemento comparador devuelve el dato de mayor magnitud por su terminal $\geq$. Si ambos son iguales no realiza el intercambio. De esta forma la red se compone de dos etapas de comparadores: $\frac{M}{2}$ comparadores sucedidos por otros $\frac{M}{2}-2$. A su vez, el sistema está realimentado, por lo que luego de cada etapa de ordenamiento el resultado se vuelve a guardar en el registro. Luego de $\frac{M}{2}$ ciclos de reloj, el segmento siempre queda ordenado de mayor a menor. En este caso particular sólo los L elementos mayores son utilizados, descartándose los $M-L$ restantes. De esta forma el valor de M determina la cantidad de ciclos de reloj necesarios y el tamaño del registro de la red. Si tenemos N datos a ordenar, se realizarán inicialmente $\frac{N}{M}$ comparaciones, y se obtendrán los $\frac{N}{M} \times L$ elementos mayores ordenados, que corresponderán a los grupos de L elementos mayores de cada comparación. Luego se realizan $\frac{\frac{N}{M} \times L}{M}$ nuevas comparaciones, obteniéndose $\frac{\frac{N}{M} \times L}{M} \times L$ elementos mayores. Así siguiendo hasta que se cumpla la desigualdad:

$$N \times \left(\frac{L}{M}\right)^{\alpha} \leq M, \tag{1}$$

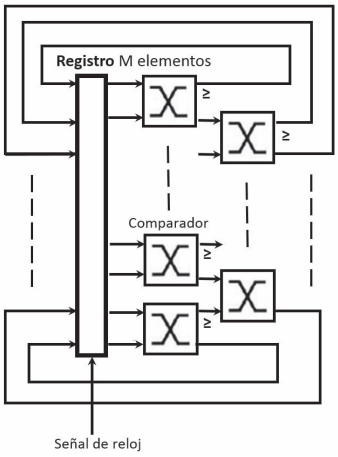


Fig. 2. Interior del bloque Red de Ordenamiento (Paralelo).

donde se debe cumplir que $L < M$ para que el algoritmo converja. Gráficamente, este procedimiento se muestra en la Fig. 3.

Del mínimo valor de α que satisface la ec. 1 obtenemos la cantidad de etapas de comparación, $\alpha = \lceil \frac{log(\frac{M}{N})}{log(\frac{L}{M})} \rceil$.

La primer etapa itera sobre la red paralela de ordenamiento $\frac{N}{M}$ veces, la segunda etapa $\frac{N \times L}{M^2}$ veces, y así sucesivamente. Si M, N y L tienen la misma base, la cantidad de iteraciones en la red resulta ser la ec. 2:

$$N_{iter} = \sum_{i=1}^{\alpha} \frac{N \times L^{(i-1)}}{M^i} \tag{2}$$

Mediante la serie geométrica $\sum_{k=1}^{n} r^k = \frac{r - r^{n+1}}{1-r}$ se llega a:

$$N_{iter} = \frac{N}{L} \times \frac{L - L^{\alpha+1} \times M^{-\alpha}}{M - L} \tag{3}$$

El algoritmo 1 muestra el pseudocódigo propuesto en este trabajo. Allí puede verse un esquema del algoritmo desarrollado en lenguaje C++. Se distinguen dos secciones principales, una función llamada *Comp* que realiza la comparación de dos elementos y la función principal llamada *Ordenamiento*. Esta función se encargada de, calcular las etapas e iteraciones según los valores de sus parámetros y, por medio de la función *Comp*, realizar el ordenamiento de los datos, finalmente los índices a entregar son las posiciones iniciales del vector desordenado. Cabe destacar que este algoritmo está diseñado con el propósito de implementar sectores de su ejecución en forma concurrente, y así conseguir una optimización con respecto a la complejidad computacional que presenta.

IV. Resultados

Se analizan primero los criterios de selección de los parámetros del algoritmo con respecto a la complejidad y los recursos. Luego, se elige la arquitectura que mejor se ajusta a los requisitos explorando opciones de síntesis. Finalmente se compara el IP de ordenamiento final con implementaciones de algoritmos existentes.

Algorithm 1: Algoritmo propuesto que ordena un vector de N datos y devuelve los L índices mayores ordenados.

función $Comp$ (A, B);
Input : Elementos a comparar A, B
Output: Elemento mayor C
if $A > B$ **then**
  $C \leftarrow A$;
else
  $C \leftarrow B$;
end
función $Ordenamiento$ (buf_in, M, L);
Input : Vector de datos a ordenar buf_in, tamaño de secciones M, número de salidas L
Output: Indices ordenados $Sorted\_Index$
$N \leftarrow length(buf\_in)$;
$d[M]$;
for $\alpha \leftarrow 1$ ***to*** $\lceil \frac{log(\frac{M}{N})}{log(\frac{L}{M})} \rceil$ **do**
  for $N_{iter} \leftarrow 1$ ***to*** $\frac{L^{\alpha-1}}{M^{\alpha}}$ **do**
    for $h \leftarrow 0$ ***to*** M **do**
      $d[h] \leftarrow buf\_in[h + N_{iter} \times M]$;
    end
    for $m \leftarrow 0$ ***to*** $M/2$ **do**
      $sort\_1$: **for** $i \leftarrow 0$ ***to*** $M/2$ **do**
        $Comp(d[i \times 2], d[i \times 2 + 1], \&buf[i \times 2 + 1], \&buf[i \times 2])$;
      end
      $sort\_2$: **for** $i \leftarrow 1$ ***to*** $M/2 - 2$ **do**
        $Comp(buf[i], buf[i + 1], \&d[i + 1], \&d[i])$;
      end
      $d[0] \leftarrow buf[0]$;
      $d[M - 1] \leftarrow buf[M - 1]$;
    end
    for $h \leftarrow 0$ ***to*** L **do**
      $buf\_in[h + N_{iter} \times L] \leftarrow d[h]$;
    end
  end
end
$p_A \leftarrow 0$;
$p_B \leftarrow 0$;
for $i \leftarrow 0$ ***to*** L **do**
  if $buf\_in[p_A] > buf\_in[p_B + L]$ **then**
    $Sorted\_Index[i] \leftarrow p_A$;
    $inc\ p_A$;
  else
    $Sorted\_Index[i] \leftarrow p_B + L$;
    $inc\ p_B$;
  end
end

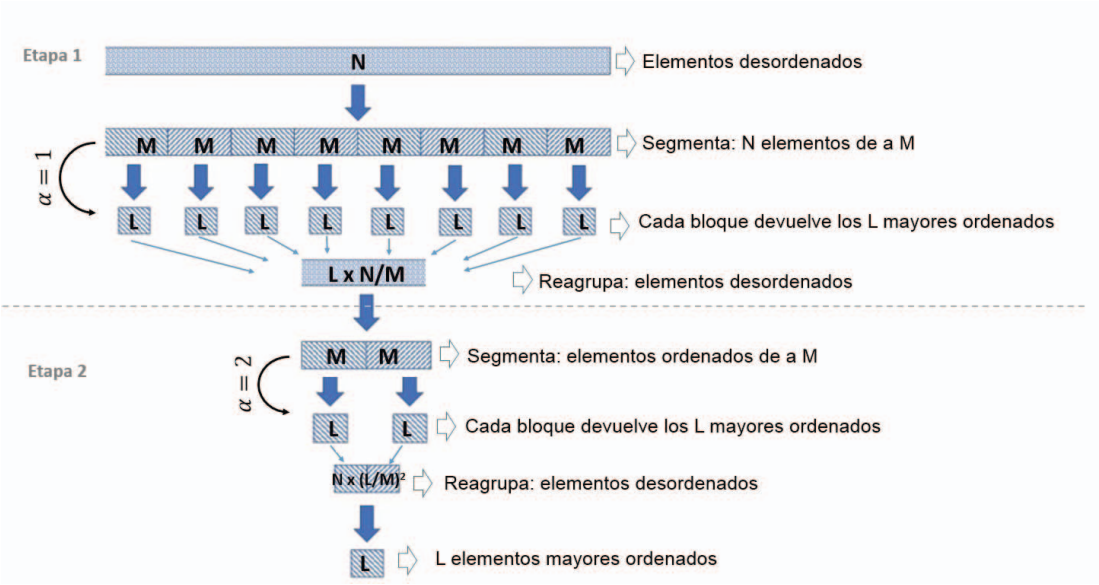


Fig. 3. Procedimiento de ordenamiento para N datos de entrada, L de salida y red de ordenamiento de tamaño M.

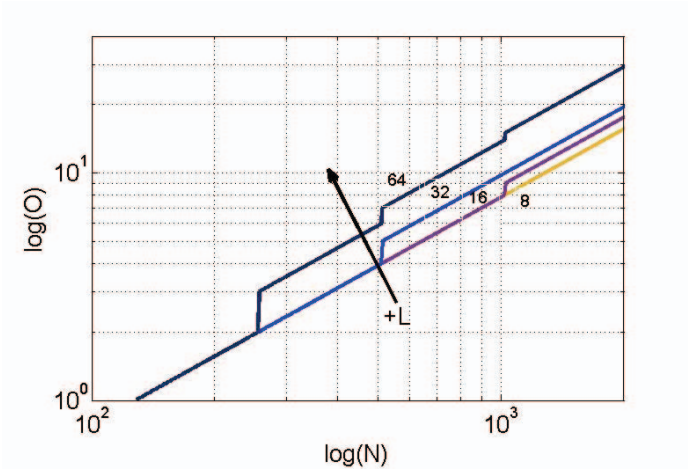


Fig. 4. Complejidad del algoritmo propuesto con $M = 128$ a medida que varía L.

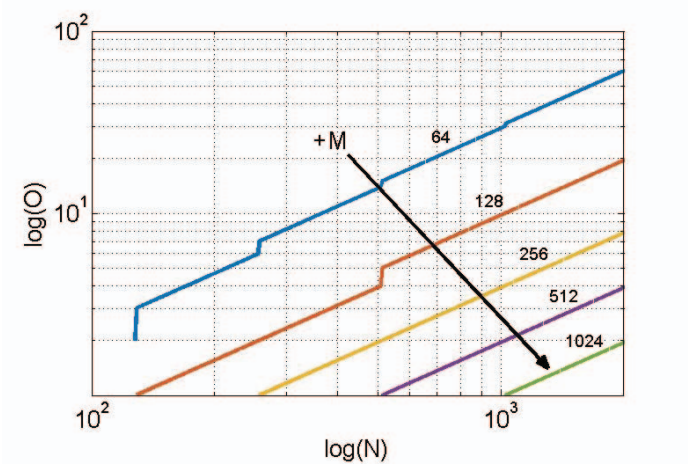


Fig. 5. Complejidad del algoritmo propuesto con $L = 32$ a medida que varía M.

A. *Selección de Parámetros*

Las Figuras 4 y 5 muestran cómo afecta a la complejidad del algoritmo la variación de los valores de M y L, éstas fueron calculadas mediante la ec. 3. En la Fig. 4 se muestra la dependencia de la complejidad del algoritmo frente a la variación de L. Se establece un valor de M (en este caso 128) y se varía L, manteniendo el valor de L menor a M para que el algoritmo converja. Puede observarse que, a medida que L disminuye, disminuye la complejidad, saturando para valores menores a 8. Para $L = 8$, 4 y 2 la complejidad colapsa sobre la misma recta. Este comportamiento se repite para todos los valores de M, donde M mayores generan que las curvas colapsen a partir de valores de L mayores. En nuestro caso el valor de L está fijado por los requisitos de la implementación ($L = 32$), esto también establece que $M > 32$. La Fig. 5 muestra que la complejidad disminuye al incrementarse M, luego, según esta figura el mejor M sería el mayor posible. Hasta aquí se consideró únicamente la complejidad que presenta el algoritmo, sin embargo la máxima frecuencia de operación, la latencia y los recursos requeridos también juegan un papel importante en el diseño. Cuanto mayor es la entrada a la red de ordenamiento, la red requerirá de más ciclos de reloj para entregar la salida ordenada y a su vez se requerirán más elementos combinacionales y un registro de mayor tamaño (Fig. 2). Para analizar la dependencia en recursos consumidos al variar M se implementa la arquitectura optimizada del algoritmo (se detalla en la Subsección IV-B) en la zona PL de la Zedboard para distintos valores de M. Pueden verse en la Fig. 6 los resultados de estas implementaciones, para el caso particular en el que se requiere obtener los 32 índices de los valores mayores ($L = 32$) de los 1.024 datos recibidos ($N = 1.024$). En la figura se ve que a partir de $M = 128$ la latencia aumenta significativamente, y el consumo de recursos en cuanto a FFs (Flip Flops) y LUTs (Look Up Tables) crece de forma lineal. Por lo que en este caso se eligió

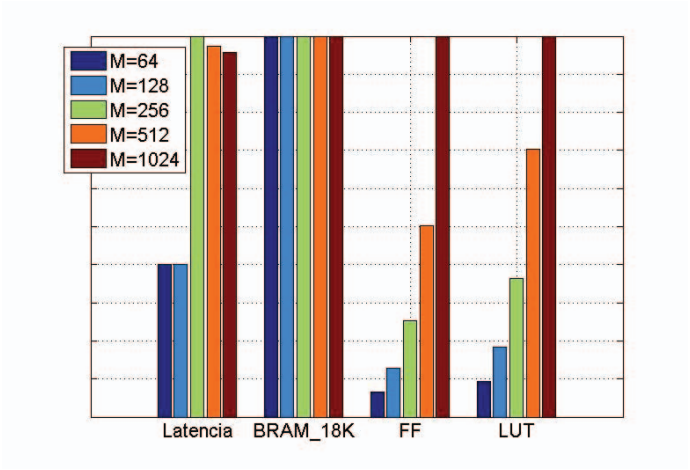


Fig. 6. Consumo de recursos y latencia del algoritmo para distintos valores de M, con $L = 32$ y $N = 1.024$.

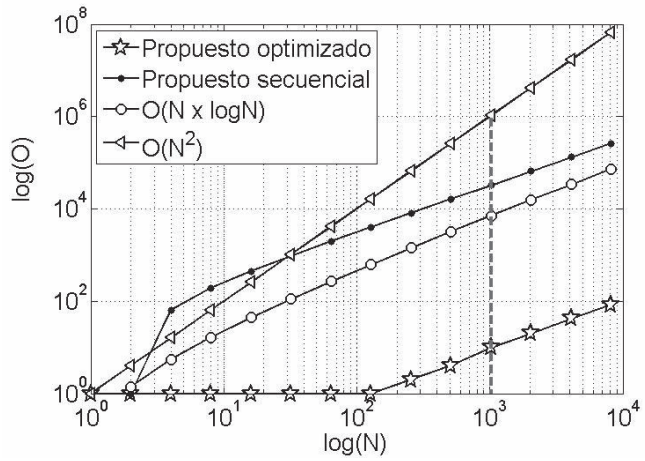


Fig. 7. Complejidad del algoritmo propuesto totalmente secuencial y optimizado frente a complejidades tradicionales.

$M = 128$.

En la Fig. 7 puede verse la complejidad del algoritmo propuesto frente a complejidades de algoritmos de ordenamiento tradicionales, $O(N^2)$ (Inserción, Burbujeo, etc) y $O(N \times logN)$ (Mezcla, etc), [18]. Se calculó la complejidad del algoritmo propuesto (utilizando la ec. 3) considerando su ejecución en forma secuencial ($L = 1$ y $M = 2$), y para el caso optimizado ($L = 32$ y $M = 128$). Puede verse allí que la complejidad del algoritmo propuesto ejecutado en forma secuencial resulta ser mayor que la complejidad $O(N^2)$ y menor que $O(N \times logN)$ sólo para sets de datos mayores a 50. Esto se debe a que el algoritmo propuesto no está diseñado para ser ejecutado de forma totalmente secuencial. Para el caso de nuestra implementación en el que se deben ordenar 1.024 datos (línea cortada en Fig. 7), la complejidad del algoritmo propuesto en forma secuencial se reduce en un $96,88\%$ con respecto a una complejidad de tipo $O(N^2)$. Sin embargo es un $78,29\%$ mayor que para una complejidad del tipo $O(N \times logN)$. Se ve una mejora significativa en cuanto a reducción de complejidad cuando se considera el algoritmo optimizado, la complejidad se reduce de 32.704 ciclos (algoritmo secuencial) a 10 ciclos. Siendo varios órdenes de magnitud menor que los tres casos presentados. El algoritmo optimizado presenta una mejora del $99,85\%$ con respecto a $O(N \times logN)$ y de $99,99\%$ con respecto a una complejidad $O(N^2)$ y al algoritmo ejecutado en forma secuencial.

Para sets de menos de 128 datos el algoritmo propuesto requiere sólo una iteración, ya que la red paralela de ordenamiento en este caso es de $M = 128$. A partir de ahí crece en forma menor a lineal, presentando escalones que se corresponden con los valores enteros de las etapas requeridas α.

B. Síntesis

Para la síntesis del circuito, se exploran distintas configuraciones híbridas para elegir una arquitectura que minimice los recursos y la latencia, y que pueda operar a la frecuencia requerida. Esto es, principalmente determinar cuáles sectores del código se ejecutarán en forma secuencial y cuáles en forma concurrente, como así también ajustes más finos como partición de circuitos combinacionales mediante pipeline, sectores de memoria, entre otros. El sistema se diseñó en el entorno Vivado HLS de la empresa Xilinx [19], para luego ser implementado en una placa Zedboard. Esta herramienta permite programar mediante lenguaje de alto nivel, en este caso C++, el algoritmo y el sintetizador se encarga de "traducir" el código a RTL (Register-transfer level) para ser implementado en la FPGA. Este sintentizador se encarga también de optimizar el diseño.

Para elegir la arquitectura óptima se utilizan directivas de síntesis de alto nivel las que permiten dar indicaciones al sintetizador. Se generan distintos circuitos (soluciones) a partir del mismo código fuente hasta encontrar la solución óptima que cumpla con los requisitos del sistema. Se emplea la directiva *PIPELINE* para indicar al sintetizador dónde segmentar las funciones, bucles y/u operaciones. También se utilizan las directivas *UNROLL* para implementar los bucles en forma concurrente y *MEMORY PARTITION* que particiona la memoria y permite su lectura en forma paralelo. También se utiliza la directiva *HLS INTERFACE axis register* para indicar que se quiere realizar una comunicación de datos mediante el estándar de comunicación AXI Stream.

En la Tabla I pueden verse las soluciones implementadas donde se indica en qué línea del algoritmo 1 se insertó cada directiva. Cada una de estas soluciones se corresponde con una arquitectura diferente. En la Tabla. II pueden verse los recursos necesarios y latencias para las distintas soluciones generadas. La *Solución 1* se corresponde al algoritmo sin ninguna directiva, en este caso la herramienta de síntesis tuvo libertad absoluta para realizar el circuito final. En este caso, se obtiene un circuito con bajo uso de FFs y LUTs, pero alta latencia y gran consumo de bloques RAM. El agregado de la directiva *HLS UNROLL* permite bajar la latencia (*Solución 3* y *Solución 4*) a expensas de un aumento en el uso de FFs y LUTs. En las soluciones generadas el máximo retardo (que determina la frecuencia máxima de operación) varió entre

$6,508\ ns$ (*Solución 5* y *Solución 7*) y $11,152\ ns$ (*Solución 3*). Las *Soluciones 5* y *7* reducen significativamente el retardo (admitiendo una frecuencia de operación mayor), sin embargo en ambos casos la latencia es muy alta, como así también el consumo de FFs y LUTs sobre todo en la *Solución 7*.

Como puede verse en la Tabla II, la herramienta de síntesis optimiza e implementa el algoritmo de forma compleja, por lo cual no siempre se cumple que la inserción de pipelines disminuya el retardo, y el desglose de bucles disminuya la latencia. Los efectos de las directivas en algunos casos se compensan y/o se potencian por lo que elegir la mejor solución no es una tarea trivial.

En este caso se optó por la *Solución 6* la cual presenta un retardo que cumple con la frecuencia de operación establecida. Además, presenta la mínima latencia y una utilización de FFs y LUTs relativamante baja. En esta solución se hace uso de la sentencia $\#pragma$ *HLS INLINE recursive* para que la función que realiza las comparaciones no se sintetice como un bloque aparte sino como varios elementos dentro de la red. Los lazos *for* principales tienen incluidos $\#pragma$ *HLS UNROLL* lo que indica al sintetizador que implemente todas las comparaciones en forma paralela.

La herramienta de síntesis genera a partir del código en C++ de 130 líneas, un archivo VHDL de más de 8.000 líneas. La estructura final del código puede verse en el diagrama en bloques en la Fig. 8. El sistema diseñado recibe un vector de $N = 1.024$ elementos enteros sin signo de 32 bits mediante una interfaz AXI Stream. Luego, se procede a agregar el índice correspondiente (posición actual del elemento en el vector desordenado) como cabecera de 10 bits a cada dato. Seguidamente, el vector ahora de 42 bits por 1.024 se almacena en una memoria RAM. La máquina de estado, que controla la secuencia de pasos desde la recepción, procesamiento y transmisión, envía en bloques de $M = 128$ datos a la red de ordenamiento en paralelo (Fig. 2).

Sin embargo, en cada proceso de ordenamiento de un bloque de 128 datos, sólo se almacenan los $L = 32$ mayores y se descarta el resto. Es decir, después de utilizar la red 8 veces, se tendrán 256 elementos restantes. Posteriormente, se vuelve a dividir al remanente en dos partes iguales para ordenarlas. Como resultado, se obtienen los 64 elementos mayores divididos en dos grupos ordenados de 32.

Una vez que se obtienen los dos conjuntos de 32 datos, se procede a cargar los registros A y B con el primer dato de cada grupo. Ambos números son comparados y se transmite a la salida el índice (los primeros 10 bits) del elemento mayor o igual.

El registro que contenía el valor cuyo índice fue transmitido es reemplazado por uno nuevo perteneciente al mismo conjunto que el anterior. De esta manera, luego de repetir este proceso 32 veces, se envía la ubicación de los 32 números mayores.

Los resultados del análisis temporal se muestran en la Fig. 9. El período mínimo de señal de reloj es $7,78\ ns$ lo que corresponde a una frecuencia máxima de $129\ MHz$ y la latencia, medida en ciclos, varía entre 3.108 ($31,08\ \mu s$) hasta 5.580 ($55,80\ \mu s$). Si el vector esta completamente ordenado (mejor caso) se tarda el tiempo mínimo. En caso contrario, el

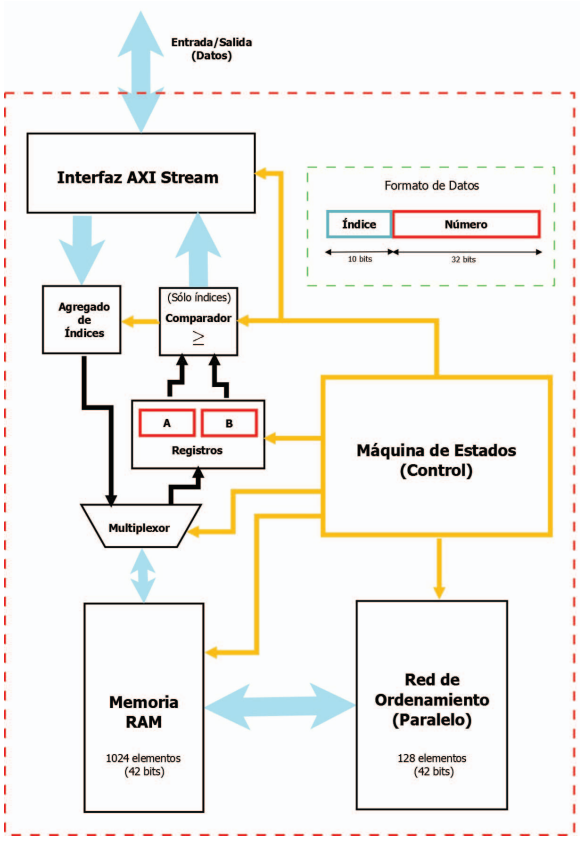


Fig. 8. Diagrama en Bloques para el IP de ordenamiento.

máximo. La red de ordenamiento utiliza el 55% de las LUTs, el 15% de los FFs disponibles, 3 bloques de memoria BRAM y ningún elemento DSP.

C. Comparación Frente a Otros Algoritmos

Con el fin de comparar el algoritmo propuesto con otros algoritmos de ordenamiento conocidos se implementaron en HLS los algoritmos de ordenamiento por Mezcla (Merge Sort) [5], ordenamiento por Inserción (Insertion Sort) [5], ordenamiento de Burbujeo (Buble Sort) [2] y una variación del ordenamiento por Burbujeo que sólo ordena los primeros 32 elementos mayores del set (aquí llamado Burujeo32). Esto se realizó ya que los requisitos sólo pedían los primeros 32 elementos mayores, en el caso del algoritmo por Inserción no fue posible ya que este algoritmo no permite realizar una cota, inevitablemente se deben ordenar todos los elementos para luego extraer los 32 mayores.

El algoritmo de Mezcla está definido recursivamente, en el caso de su implementación mediante HLS esta herramienta (al igual que VHDL) no soporta la recursividad. Por esta razón se lo implementó de forma no recursiva, lo que requirió un espacio adicional de memoria para almacenamiento de la pila.

En la Tabla III pueden verse una comparación de los períodos mínimos (o frecuencias máximas de operación) para los algoritmos implementados, incluyendo el propuesto. Se observa que el algoritmo propuesto es el que presenta el menor retardo, además es el único que cumple con el requisito frecuencia de reloj solicitado en las especificaciones

TABLA I
UBICACIÓN DE DIRECTIVAS EN EL ALGORITMO 1 PARA CADA SOLUCIÓN SINTETIZADA

Directiva	Sol. 1	Sol. 2	Sol. 3	Sol. 4	Sol. 5	Sol. 6	Sol. 7
INTERFACE	7	7	7	7	7	7	7
PIPELINE	-	12	12	12	-	12;25	-
UNROLL	-	10;11	10;11;16;19	25		16;19	10;11;16;19;25
ARRAY_PARTITION	-	-	-	-	9	7;9	9
INLINE	-	-	-	-	1	1;7	1;7

TABLA II
COMPARACIÓN DE DISTINTAS SOLUCIONES MEDIANTE LA UTILIZACIÓN DE DIRECTIVAS EN EL MISMO CÓDIGO FUENTE

Recursos	Sol. 1	Sol. 2	Sol. 3	Sol. 4	Sol. 5	Sol. 6	Sol. 7
Retardo [ns]	6,720	8,758	11,152	9,957	6,508	7,780	6,508
Latencia	530.599	249.482	85.002	85.014	859.527	5.879	8.332
FFs	489	4.711	8.387	6.153	40.209	16.517	108.414
LUTs	1.106	1.595	8.017	4.781	17.931	29.264	181.104
BRAM_18k	9	9	6	6	3	3	3

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.78	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3107	5879	3108	5880	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	12217
FIFO	-	-	-	-
Instance	-	-	0	340
Memory	3	-	0	0
Multiplexer	-	-	-	16707
Register	-	-	16517	-
Total	3	0	16517	29264
Available	280	220	106400	53200
Utilization (%)	1	0	15	55

Fig. 9. Resultado de tiempo y recursos utilizados.

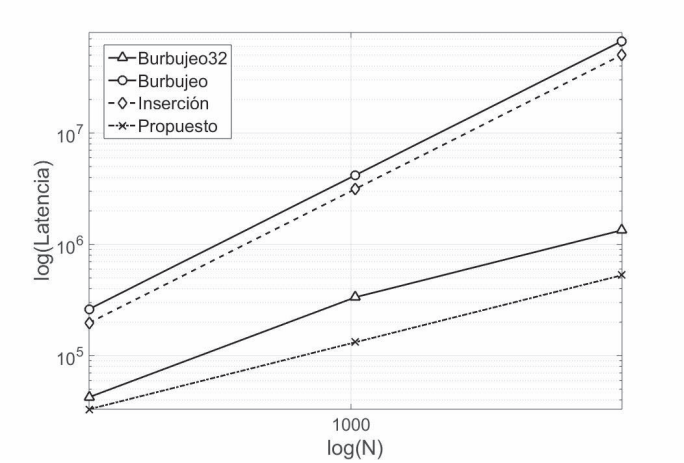


Fig. 10. Comparación de latencia en función del tamaño del set de elementos a ordenar N, latencia en ciclos de reloj

(125 MHz) ya que su frecuencia máxima de operación es de 129 MHz. También se muestran la latencia y los recursos empleados por cada uno de los algoritmos de ordenamiento implementados, puede verse que el algoritmo Mezcla es el que presenta peor comportamiento (más consumo de recursos), esto se debe a lo ya comentado que no se pudo utilizar recurrencia por lo que se implementó de forma alternativa a costa de recursos. El circuito desarrollado requiere de un 52% menos de FFs y un 68% menos de LUTs con respecto al mejor del resto de los algoritmos implementados, Burbujeo32 en este caso. Además, se ve claramente que el IP de ordenamiento desarrollado presenta una latencia significativamente inferior a la que presentan el resto de los algoritmos.

La Fig. 10 muestra los resultados de latencia, entregados por el sintetizador, para cada algoritmo implementado. Puede verse que los algoritmos Burbujeo e Inserción son los peores en términos de latencia, el algoritmo propuesto es el que presenta menor latencia, seguido por el Burbujeo32.

V. CONCLUSIONES

Se diseñó e implementó un algoritmo de ordenamiento híbrido que combina procesamiento secuencial y concurrente en un dispositivo SoC. Se utilizó síntesis de alto nivel como técnica de descripción de hardware, mediante el uso de directivas se logró que el compilador sintetice el circuito óptimo y que cumpla con los requisitos. El circuito final implementado redujo la complejidad del algoritmo un $99,99\%$ con respecto al algoritmo propuesto ejecutado en forma secuencial y en

TABLA III
COMPARACIÓN ENTRE IMPLEMENTACIONES DE DISTINTOS ALGORITMOS

Recursos	Mezcla	Inserción	Burbujeo	Burbujeo32	IP ordenamiento
Retardo [ns]	8,148	8,408	8,685	8,685	7,780
Latencia	4.197.484	3.146.770	3.144.722	3.144.722	5.879
FFs	52.814	35.702	38.412	35.111	16.517
LUTs	161.328	143.202	95.120	88.312	29.264
BRAM_18k	11	4	3	3	3

$99,85\%$ con respecto a la mínima complejidad de los algoritmos clásicos ($O(NlogN)$). Se reportaron los recursos requeridos y el análisis de tiempo del sistema. Se realizaron comparaciones con otros algoritmos de ordenamiento implementados en la misma plataforma, se analizaron recursos empleados, latencia y tiempo de ejecución. Se encontró que el algoritmo propuesto emplea menos de la mitad de los recursos (LUTs y FFs), admite una frecuencia de reloj mayor y presenta una latencia menor que los algoritmos clásicos, inclusive presenta mejor comportamiento que el algoritmo Burbujeo optimizado para esta aplicación particular. El algoritmo propuesto demostró ser el más eficiente en términos de retardo, latencia y recursos y muy superior con respecto a la complejidad presentada. El bloque IP desarrollado fue empleado en un sistema de sensado compresivo caótico cumpliendo los requerimientos solicitados y permitiendo acelerar el proceso de reconstrucción de la señal muestreada mediante el algoritmo AIHT.

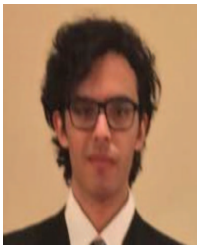
AGRADECIMIENTOS

Este trabajo fue parcialmente financiado por el Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), UNMDP Argentina e ICTP.

REFERENCIAS

[1] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.
[2] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 1–5.
[3] E. D. Demaine, G. M. Landau, and O. Weimann, "On cartesian trees and range minimum queries," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2009, pp. 341–353.
[4] B. S. Chlebus, "A parallel bucket sort," *Information processing letters*, vol. 27, no. 2, pp. 57–61, 1988.
[5] D. E. Knuth, *Art of Computer Programming, Volumes 1-4A Boxed Set*. Addison-Wesley Professional, 2011.
[6] M. Dowd, Y. Perl, L. Rudolph, and M. Saks, "The periodic balanced sorting network," *Journal of the ACM (JACM)*, vol. 36, no. 4, pp. 738–757, 1989.
[7] V. Sklyarov, I. Skliarova, and A. Sudnitson, "Fpga-based accelerators for parallel data sort," *Applied Computer Systems*, vol. 16, no. 1, pp. 53–63, 2014.
[8] S. Dong, X. Wang, and X. Wang, "A novel high-speed parallel scheme for data sorting algorithm based on fpga," in *2009 2nd International Congress on Image and Signal Processing*. IEEE, 2009, pp. 1–4.
[9] F. A. Alquaied, A. I. Almudaifer, and M. A. AlShaya, "A novel high-speed parallel sorting algorithm based on fpga," in *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*. IEEE, 2011, pp. 1–4.
[10] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Parallel fpga-based implementation of recursive sorting algorithms," in *2010 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2010, pp. 121–126.
[11] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal?The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, 2012.
[12] A. Széll, "Parallel sorting algorithms in fpga," in *13TH PHD MINI-SYMPOSIUM*, 2006, p. 8.
[13] M. L. Acosta, M. Antonelli, and L. De Micco, "Chaotic compressed sensing system for 16x sub-nyquist signal reconstruction," in *2019 Argentine Conference on Electronics (CAE)*. IEEE, 2019, pp. 31–36.
[14] M. L. Acosta and L. De Micco, "Xampling and chaotic compressive sensing signal acquisition and reconstruction system," in *2017 XVII Workshop on Information Processing and Control (RPIC)*. IEEE, 2017, pp. 1–6.
[15] M. L. Acosta, "System on chip para la reconstrucción de señales con sensado compresivo," Ph.D. dissertation, Universidad Nacional de Mar del Plata. Facultad de Ingeniería. Argentina, 2018.
[16] T. Blumensath, "Accelerated iterative hard thresholding," *Signal Processing*, vol. 92, no. 3, pp. 752–756, 2012.
[17] A. Xilinx, "Reference guide, ug761 (v13. 1)," *URL http://www. xilinx. com/support/documentation/ip documentation/ug761 axi reference guide. pdf*, 2011.
[18] C. E. Leiserson, R. L. Rivest, T. H. Cormen, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, MA, 2001, vol. 6.
[19] V.-H. Xilinx, "Vivado design suite user guide-high-level synthesis," 2014.

Luciana De Micco Recibió su título de Ingeniera Electrónica de la Universidad Nacional de Mar del Plata en 2003 y su Doctorado en Ingeniería con orientación electrónica en la misma universidad en 2009. Actualmente es directora Laboratorio de Sistemas Caóticos de ICYTE e Investigadora Adjunta de CONICET, también es Profesora Adjunta del Departamento de Electrónica de la Facultad de Ingeniería de la Universidad Nacional de Mar del Plata. Investigadora Junior Asociada a Abdus Salam International Centre for Theoretical Physics (ICTP).



Mariano L. Acosta Es graduado de la carrera de Ingeniería Electrónica de la Universidad Nacional de Mar del Plata (2018). Su tema de tesis de fin de carrera de grado es "System on Chip para reconstrucción de señales mediante Compressed Sensing".

Maximiliano Antonelli Es graduado de la carrera de Ingeniería Electrónica de la Universidad Nacional de Mar del Plata (2012) y Doctor en Ingeniería con orientación electrónica (2018) de la misma universidad. Es Jefe de Trabajos Prácticos del Departamento de Física e investigador del Laboratorio de Sistemas Caóticos de ICYTE. Sus principales temas de interés son el estudio de la dinámica no lineal, sistemas caóticos, digitalización e implementación de hardware.