Implementación de algoritmos de ordenamiento para el detector Near-ML en FPGA

Aarón Escoboza Villegas
Instituto Tecnológico de Sonora
Cd. Obregón, México
aaron.villegas@hotmail.com

Eduardo Romero Aguirre
Instituto Tecnológico de Sonora
Cd. Obregón, México
Eduardo.romero@itson.edu.mx

***Resumen*—En este artículo se presentan la evaluación de los algoritmos de ordenamiento en términos de su tiempo de ejecución y los recursos de hardware consumidos del algoritmo con mejor rendimiento. El algoritmo de selección promete ser el indicado para incorporarlo a detectores cuyo parte esencial es el ordenamiento de una cantidad de datos relativamente pequeña como lo es en el detector Near-ML.**

Palabras clave*—*FPGAs, Algoritmos de ordenamiento,Vidado HLS

I. Introducción

Actualmente, la mayoría de las aplicaciones que realizan cálculos computacionales necesitan los datos en un orden especifico [1]. El procedimiento de colocar los datos en un orden dado es llamado ordenamiento, tal procedimiento se ha convertido en una de las operaciones fundamentales en la computación [2],[3]. Dado una secuencia de n números $< a_1, a_2, \dots, a_n >$ el problema de ordenamiento se define como la permutación $< a'_1, a'_2, \dots, a'_n >$ tal que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ [4]. La operación de ordenamiento se puede encontrar en gran variedad de aplicaciones como: procesamiento de señales, compresión de datos, sistemas de adquisición de datos [5], [6],[7].

En las últimas décadas la industria semiconductora ha crecido de manera exponencial a pesar de su complejidad y el rendimiento del hardware. FPGAs ha sido la tecnología con mayor desarrollo en comparación con el resto de industria [8]. Dada las aplicaciones del ordenamiento y el crecimiento de los FPGAs, el trabajo en conjunto de estas dos se ha convertido en un tema interesante de investigación.

Por otra parte, existen algunos retos, los programadores competentes en lenguajes de descripción de hardware como VHDL y Verilog son muy pocos, por lo tanto, los no expertos deben alcanzar un alto nivel de especialización para el desarrollo de modelos eficientes a nivel de registro de transferencia (RTL). Además, VHDL y Verilog no son tan poderosos como los lenguajes de alto nivel lo cual conlleva a códigos significativamente largos incrementando la probabilidad de errores de codificación y tiempo en realizar modificaciones una vez el diseño está hecho.

La síntesis de alto nivel (HLS) puede ayudar a superar los retos anteriormente mencionados [9]. HLS ha estado creciendo rápidamente en los últimos 30 años. Esta tendencia es debido principalmente por la necesidad de una rápida y confiable de una plataforma que, empezando de un modelo en alto nivel descrito en C o C++ genere su contraparte de bajo nivel, que luego se pueda implementar en diferentes tecnologías como lo es el FPGA [10].Entonces, las herramientas HLS han experimentado un aumento de popularidad debido a que se especializan en generar modelos RTL de alta calidad de producción basados en modelos de alto nivel. Uno de los HLS más populares es Vivado HLS [11].

El ordenamiento es crucial para el rendimiento general de algunos detectores, uno de ellos es el algoritmo de baja complejidad cerca de la máxima verosimilitud (Near-ML) cuya eficiencia depende completamente del ordenar de 48 y 64 datos [12].

Este trabajo se organiza de la siguiente manera: sección II presenta los algoritmos más conocidos para el ordenamiento de datos; en la sección III , simulación en Matlab e implementación en Vivado son mostrados; finalmente, en la sección IV se realizan conclusiones del trabajo y discusiones sobre futuras investigaciones en el área de estudio.

II. Algoritmos de ordenamiento

En esta sección se presentarán los algoritmos mas conocidos para el ordenamiento de datos y su respectivo seudocódigo.

A. Algoritmo de inserción

El algoritmo de inserción incorpora un nuevo elemento en cada iteración y compara los valores de los elementos en la lista. Si el valor de un elemento es menor que el valor del presente elemento, entonces una operación de intercambio es realiza. Esto se repite hasta $n - 1$ elementos [13].

INSERTION-SORT (A)

```
1       for j = 1 to n – 1
2               min = j
3               for i = j + 1 to n
4                       if A[i] < A[min]
5                               min = i
6               swap A[j], A[min])
```

B. Algoritmo de selección

El algoritmo de selección es parte de la familia de ordenamiento por comparación en el lugar. Su nombre se debe a que selecciona un elemento mínimo en cada paso de la ordenación. El método funciona de la manera siguiente: para ordenar en orden creciente, el primer elemento se compara con todos los elementos. Si el primer elemento es mayor que el elemento más pequeño se intercambia la posición de los elementos. Entonces, después de una primera pasada ,el elemento mas pequeño se coloca en la primera posición. El procedimiento se repite para el segundo elemento y se repite hasta tener todos los elementos ordenados [14].

SELECTION-SORT (A)

```
1       for j = 2 to A.length
2       key = A[J]
3               i = j - 1
4               while i > 0 and A[i] > key
5                       A[i + 1] = A[i]
6                       i = i - 1
7               A[i + 1] = key
```

C. Algoritmo de ordenamiento por montículos

Ordenamiento por montículos es un árbol binario completo que satisface el orden del montón. Hay dos tipos de montón. El montón mínimo que contiene el valor mas pequeño en el nodo raíz y el montón máximo que contiene el valor mas grande en la raíz. El algoritmo funciona de la siguiente manera. Se construye un montón máximo con un arreglo A[1...n].Se extrae el valor más grande del montón repetidamente. Cuando se elimina el elemento mas grande se deja un árbol secundario como nueva raíz. Se repite el mismo procedimiento hasta llegar a la última hoja [15].

HEAPSORT (A)

```
1 BUILD-MAX-HEAP (A)
2       for i = A.length downto 2
3               exchange A[1] with A[i]
4               A.heapSize = A.heapSize - 1
5               MAX-HEAPIFY (A,1)
```

BUILD-MAX-HEAP (A)

```
1       A.heapSize = A.length
2       for i = A.length/2 downto 1
3               MAX-HEAPIFY(A,i)
```

MAX-HEAPIFY(A,i)

```
1       l = 2i
2       r = 2i + 1
3       if l ≤ A.heapSize and A[l] > A[i]
4               largest = l
5       else largest = i
6       if r ≤ A.heapSize and A[r] > A[largest]
7               largest = r
8       if largest ≠ i
9               exchange A[i] with A[largest]
10              MAX-HEAPIFY(A,largest)
```

D. Algoritmo de ordenamiento por mezcla

John von Neumann fue el inventor de la ordenación por mezcla la cual pertenece a la familia de ordenación basada en comparación. El algoritmo también se basa en el enfoque dividir y conquistar. Dada una secuencia de n elementos $A[1], \dots A[N]$, la idea general es vividir en dos conjuntos de datos $A[1], \dots A[N/2]$ y $A\left[\frac{n}{2}+1\right], \dots A[N]$. En la etapa de conquistar se ordena cada conjunto individualmente de forma recursiva. Finalmente, la secuencia resultante se fusiona para producir solo una secuencia ordenada de N elementos [16].

MERGE(A,p,q,r)

```
1       n1 = q - p + 1
2       n2 = r - q
3       for i = 1 to n1
4               L[i] = A[p + i - 1]
5       for j = 1 to n2
6               R[j] = A[q + j]
7       L[n1 + 1] = infinito
8       R[n2 + 1] = infinito
9       i = 1
10      j = 1
11      for k = p to r
12              if L[i] ≤ R[j]
13                      A[k] = L[i]
14                      i = i + 1
15              else A[k] = R[j]
16                      j = j + 1
```

MERGE-SORT(A,p,r)

```
1       if p < r
2               q = lowest((p + r)/2)
3               MERGE-SORT(A,p,q)
4               MERGE-SORT(A,q + 1,r)
5               MERGE(A,p,q,r)
```

III. Simulación

Con el fin de identificar el algoritmo de ordenamiento con menor tiempo de ejecución se realizó un script en MATLAB. En cada iteración los algoritmos mencionados en la sección II son ejecutados para ordenar un arreglo de números aleatorios. En la figura 1 se observa que los mejores algoritmos para ordenar 48 datos son el algoritmo de selección e inserción. Entre los dos mejores, el algoritmo de selección es el que consigue un menor tiempo de ejecución y por lo tanto es el algoritmo elegido a implementar en Vivado. En la figura 2 se muestra los recursos consumidos.

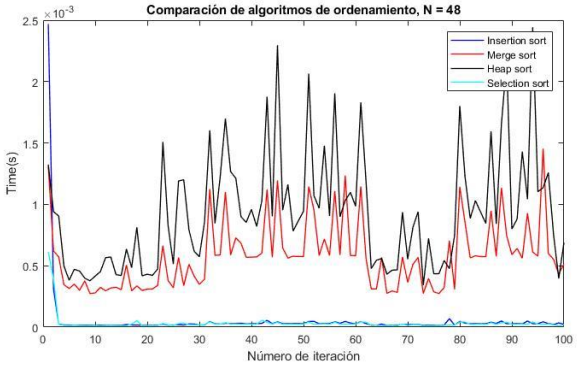

Figura 1. Comparación de algoritmos de ordenamiento

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	193
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	198
Register	-	-	366	-
Total	0	0	366	391
Available	730	740	269200	129000
Utilization (%)	0	0	~0	~0

Figura 2. Resultado de síntesis del algoritmo de selección

IV. Conclusiones

El ordenamiento es una importante operación para una gran cantidad de aplicaciones y puede ser la parte crucial para el rendimiento general de un sistema. En este artículo se evaluó tiempo de ejecución de los algoritmos de ordenamiento más populares hasta el momento. Se encontró que el algoritmo de selección es el mejor para ser incorporado al detector Near-ML y se obtuvo los recursos consumidos de hardware para ser implementado en un FPGA.

Referencias

[1] Lipu, A. R., Amin, R., Mondal, M. N. I.,& Al Mamun, M. (2016, December). Exploiting parallelism for faster implementat ion of Bubble sort algorithm using FPGA. In 2016 2nd Internat ional Conference on Elect rical, Computer & Telecommunication Engineering .

[2] D.E Knuth. The Art of computer programming, Sort ing and Searching volume II, Addison-Wesley, 2011.

[3] G. Goetz, 'Implementing sorting in database systems,' ACM Comput. Surv., vol. 38, pp. 10, 2006.

[4] S. Radhakrishnan, D. Kolippakkam, and V. S. Mathura, *Introduction to algorithms*. 2007.

[5] S. Lukáš, 'Evolutionary Design Space Exploration for Median Circuits', Lecture Notes in Computer Science, Vol. 2004, No. 3005, DE, pp. 240-249, 2004.

[6] E. Jamro, M. Wielgosz, and K. Wiatr, "FPGA Implementation of the Dynamic Huffman Encoder," Proc. Workshop of Programmable Devices and Embedded Systems, pages 60-65, February 2006.

[7] C. C. W. Robson and C. Bohm, 'A high speed data acquisition collector for merging and sorting data,' Nuclear Science Symposium Conference Record, 2008. NSS '08, 2008.

[8] D. Chen, J. Cong, and P. Pan, 'FPGA Design Automation: A Survey,' Foundations and Trends in Electronic Design Automation, vol. 1, no. 3, pp. 139–169, 2006.

[9] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, Algorithmic and Register-Transfer Level Synthesis: The System Architects Workbench. The Kluwer International Series in Engineering and Computer Science 85, Springer.

[10] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, 'An empirical evaluation of high-level synthesis languages and tools for database acceleration,' in 24th International Conference on.

[11] Xilinx Inc., Vivado Design Suite User Guide v2015.1, 2015."

[12] Hector Eduardo Aldrete Vidrio, "Sistema de comunicación multiportadora para el estándar 802.11p utilizando precodificación frecuencial y cancelación no lineal de interferencia," 2019.

[13] Y. Ben Jmaa, R. Ben Atitallah, D. Duvivier, and M. Ben Jemaa, "A comparative study of sorting algorithms with FPGA acceleration by high level synthesis," in *Computacion y Sistemas*, 2019, vol. 23, no. 1, pp. 213–230, doi: 10.13053/CyS-23-1-2999.

[14] Jadoon, S., Solehria, S. F., Rehman, S., & Jan, H. (2011). Design and analysis of optimized selection sort algorithm. International Journal of Electric & Computer Sciences (IJECS-IJENS), pp. 16–22.

[15] Al-Jaloud, E., Al-Aqel, H., & Badr, G. (2014). Comparative performance evaluation of heap-sort and quick-sort algorithms. International Journal of Computing Academic Research, pp. 39–57.

[16] Jmaa, Y. B., Ali, K., Duvivier, D., Jemaa, M. B., & Atitallah, R. B. (2017). An efficient hardware implementation of timsort and mergesort algorithms using high level synthesis. IEEE International Conference on High Performance Computing & Simulation (HPC.