Lámpsakos | Nº 15 | pp. 11-21 | enero-junio | 2016 | ISSN: 2145-4086 | Medellín–Colombia

Implementación del Algoritmo Trace Alignment Empleando Técnicas de Programación Paralela

Trace Alignment Algorithm Implementation Using Parallel Programming Techniques

Marlis Fulgueira-Camilo, MSc.
Centro de Investigación Tecnológica Integrada
La Habana, Cuba
mfulgueirac@citi.cu

Ernesto Insúa-Suárez, Ing.
Centro de Investigación Tecnológica Integrada
La Habana, Cuba
einsuas@citi.cu

Humberto Díaz-Pando, PhD.
Instituto Superior José Antonio Echeverría
La Habana, Cuba
hdiazp@ceis.cujae.edu.cu

(Recibido el 20-10-2015. Aprobado el 16-12-2015)

Estilo de Citación de Artículo:
M. Fulgueira-Camilo, E. Insúa-Suárez, H, Díaz-Pando, "Implementación del Algoritmo Trace Alignment Empleando Técnicas de Programación Paralela", *Lámpsakos*, no. 15, pp 11 - 21, 2016

Resumen. En este artículo se refiere un algoritmo del campo de la minería de procesos, Trace Alignment, cuyo objetivo es detectar anomalías en una secuencia de patrones e identificar patrones comunes. El análisis de las trazas generadas por los procesos de negocios puede tardar un tiempo considerable, teniendo en cuenta que una gran parte de los procesos, hoy en día, se encuentran informatizados. El algoritmo en cuestión es paralelizado empleando el paradigma de memoria compartida, específicamente OpenMP, CUDA y OpenCL. El diseño paralelo propuesto cuenta de dos etapas: una primera donde se paraleliza la construcción de la matriz de similitud y una segunda donde se alinean pares o conjuntos de trazas a la misma vez. Los resultados obtenidos indican que, con el diseño propuesto, los mejores tiempos se obtienen empleando OpenMP para todos los juego de datos propuestos.

Palabras clave: CUDA, Minería de Procesos, OpenCL, OpenMP, Trace Alignment.

Abstract. The article refers to an algorithm in the field of mining processes, Trace Alignment, aimed at detecting anomalies in a sequence of patterns and identify common patterns. The analysis of the traces generated by the business process may take considerable time, given that a large part of the process, today, are computerized. The algorithm in question is parallelized using the shared memory paradigm, specifically OpenMP, CUDA and OpenCL. The parallel proposed design has two stages: a first construction where the similarity matrix and a second where pairs or sets of traces at the same time is aligned. The results indicate that, with the proposed design, the best times are obtained using OpenMP.

Keywords: CUDA, OpenCL, OpenMP, Process Mining, Trace Alignment.

1. INTRODUCCIÓN

La Minería de Procesos (MP) es una técnica de administración de procesos que permite analizar registros de eventos, extraer información y convertirla de forma explícita en conocimiento. A su vez se divide en tres técnicas fundamentales: Descubrimiento de Procesos, Chequeo de Conformidad y Mejoramiento de Modelos. La técnica de Chequeo requiere comparar el registro de trazas almacenado contra un modelo para comprobar si se ha ejecutado el proceso conforme a lo esperado [1], [2]. El Trace Alignment [3] clasifica dentro de la MP como algoritmo para el Chequeo de Conformidad. Las pruebas realizadas, teniendo en cuenta el número de trazas a procesar por el algoritmo, demostraron que el tiempo de ejecución aumenta conforme a dicho número.

Otro factor que influye en el tiempo de procesamiento es la marcada diferencia que en ocasiones pueden tener las trazas. Una de las posibles soluciones al problema descrito es la computación paralela y distribuida, la cual hace uso de técnicas y herramientas que pueden disminuir, en el mayor número de los casos, el tiempo de ejecución de los algoritmos [4]. El objetivo general de la investigación es disminuir el tiempo del algoritmo Trace Alignment, para lo cual se realiza un diseño paralelo del algoritmo y se implementa empleando OpenMP, OpenCL y CUDA.

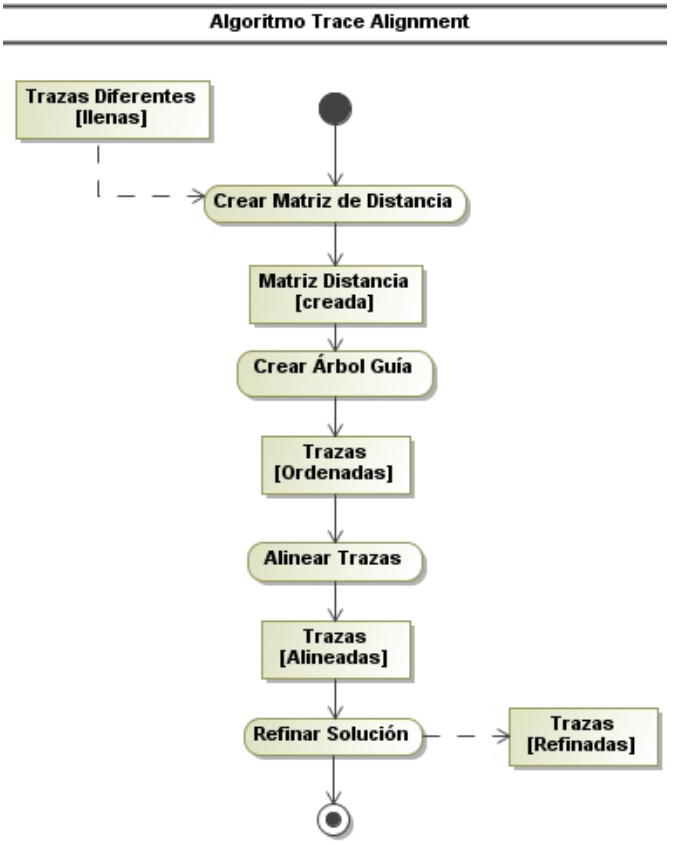

Fig. 1 Diagrama de Actividad del algoritmo Trace Alignment.

2. MARCO TEÓRICO

El crecimiento de la capacidad de almacenamiento de los sistemas de cómputo ha hecho posible que las organizaciones puedan registrar y analizar los eventos ocurridos de los procesos de negocios que en ellas realicen. A través de la MP se puede identificar violaciones en los procesos de negocios, definir cuellos de botella y recomendar medidas para el mejoramiento del proceso [1].

Uno de los métodos que pueden apoyar la técnica de Chequeo es la alineación de las trazas. El algoritmo Trace Alignment tiene como antecedente a la alineación de secuencias en el campo de la bioinformática [3]. En este ambiente se alinean pares o múltiples secuencias. La característica principal de la alineación de secuencias biológicas, es que son secuencias muy largas, por el orden de los miles [5]-[8]. En cambio en la Minería de Procesos, específicamente en el caso de estudio de este trabajo, la traza más larga contiene solo 50 caracteres, pero la cantidad de trazas a alinear si está en el orden de los miles.

El alineamiento de las trazas permite identificar el comportamiento común o más probable del proceso ejecutado, la ocurrencia de desviaciones, patrones comunes de ejecución, los contextos en que una o varias actividades son ejecutadas y patrones especiales, como ocurrencia de actividades simultáneas en el proceso .

2.1. Análisis del algoritmo Trace Alignment

El algoritmo cuenta con cuatro pasos [3] representados en la fig. 1, cuyos pasos se describen posteriormente.

Paso 1. Calcular la matriz de distancia: las alineaciones son sensitivas al tipo de métrica empleada. La métrica empleada en este trabajo está determinada por la distancia de Levenshtein [9], la cual permite

Actividad: **Crear Matriz Distancia**

```
Require: All traces different, T
1: Let m be the distance matrix
2: Let n the length of T
3: Let count the size of the row matrix to fill
4: Let d_temp the distance between two traces
5: for all traces T_i in T do
6:   for all traces T_j in T < count do
7:       d_temp=Levenshtein(T_i,T_j,i,j)
8:       m[i][j] = d_temp
9:       m[j][i] = d_temp
10:    end for
11: end for
```

Fig. 2 Seudocódigo de la actividad crear Matriz de Distancia.

Actividad: **Crear Árbol Guía**

```
Require: All traces different, T
1: Let m be the distance matrix
2: Let n the length of T
3: Let min _dist the distance minimum of all traces
4: Let col the column of matrix who have the minimum distance
5: Let row the row of matrix who have the minimum distance
6: let temp_T all traces order by distance
7: let count the control variable who start in 0
8: while count<n-1 do
9:   for all traces T_i in T < count do
10:      if m[col][i] < min_dist && col! = i
11:          min_dist = m[col][i]
12:          row = i
13:      end if
14:   end for
15:   temp_T = T[col]
16:   col = row
17: end while
```

Fig. 3 Seudocódigo de la actividad crear Árbol Guía.

Actividad: **Alinear Trazas**

```
Require: ∑j set of traces to align, T
1: Let m be the length of the alignment A
2: Let n be the length of the alignment B
2: Let MScore the matrix who have the score of alignment
3: for i = 2 to m do
4:    CalculateFrequency(i − 1, trace)
5:    MScore[i][0] = MScore[i − 1][0] + OperationIndel(trace[i], trace[i − 1])
6: end for
7: for j = 2 to m do
8:    CalculateFrequency(j − 1, trace)
9:    MScore[0][j] = MScore[j − 1][0] + OperationIndel(trace[j], trace[j − 1])
10: end for
11: for i = 1 to m do
12:    for j = 1 to n do
13:       MScore[i][j] = FillMScore(MScore[i − 1][j − 1], MScore[i − 1][j],  MScore[i][j − 1], T)
14:    end for
15: end for
16: TraceBack(MScore, T)
```

Fig. 4 Seudocódigo de la actividad crear Alinear Trazas.

Actividad: **Refinar solución. Algoritmo Block Shift Refinement**

```
Require: An alignment, A
1: Let m be the length of the alignment
2: Let ∑j denote the set of activities in column j of A
3: for all aligned traces T̄_i in A do
4:  for j = 1 to m do
5:   if there exist a block of gap of length p (p ≥ 1) starting at j in T̄_i
6:      then
7:       if there exist a k, such that j ≤ k < j + p and A(i, j + p) ∈ ∑k
8:            then
9:              swap A(i, k) and A(i, j + p). Set j = k
10:           else
11:             set j = j + p − 1
12:         end if
13:      end if
14:  end for
15: end for
16: Remove any column from A that contains only the gap symbol
```

Fig. 5 Seudocódigo de la actividad refinar solución.

identificar cuáles son las trazas más cercanas entre sí. Tiene la característica principal de ser una matriz simétrica. La fig. 2 muestra el procedimiento descrito.

Paso 2. Crear un árbol guía: la matriz de distancia previamente calculada es usada para construir el árbol guía, el cual determina el orden en que se alinean las trazas. La fig. 3 muestra el seudocódigo del paso.

Paso 3. Alinear progresivamente las trazas: Este paso se caracteriza por alinear en una primera instancia los pares de trazas, donde se crea una matriz por cada par o conjunto de trazas a alinear, llamada matriz de puntuación. Rara vez las trazas a alinear tienen el mismo tamaño. Una vez alineados los pares, se alinean los conjuntos más cercanos.

La matriz creada, por cada conjunto a alinear, es la matriz de alineamiento. Primeramente es necesario calcular la primera fila y columna de la matriz, luego se calcula uno a uno los elementos restantes comenzando por la segunda fila. Cada elemento de la matriz necesita los elementos $(i-1,j-1)$,$(i-1,j)$ y $(i,j-1)$ para obtener su valor.

Una vez calculada, para encontrar el alineamiento, es necesario buscar el camino que mejor puntuación tenga, para lo cual de la posición (i,j) se transita hacia la posición $(i-1,j-1)$,$(i-1,j)$ o $(i,j-1)$ siempre buscando el valor mayor que contenga cada celda. El movimiento hacia la posición $(i-1,j-1)$ significa $T_1(i)$ y $T_2(j)$; hacia la posición $(i-1,j)$ significa $T_1(i)$ y el símbolo de hueco $(-)$; y hacia la posición $(i-1,j)$ símbolo de hueco $(-)$ y $T_2(j)$. El recorrido termina cuando se llega a la posición $(0,0)$. El procedimiento descrito se llama *TraceBack* [3]. La fig.4 muestra el seudocódigo de la actividad Alinear Trazas.

Paso 4. Refinar el resultado final, que puede contener columnas completas de guiones o actividades muy distantes unas de otras, lo que dificulta la compresión del proceso. El algoritmo Block Shift Refine-

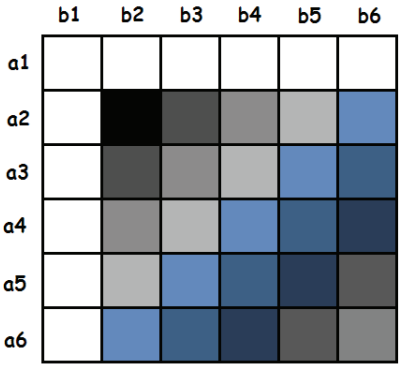

Fig. 6 Método de onda

ment [3] es el encargado de remover estas columnas y mover las actividades sin violar el orden en que fueron realizadas. La idea básica es considerar cada traza del alineamiento de izquierda a derecha e intercambiar cada actividad que se encuentre precedida por un bloque de "-" lo más a la izquierda posible en esa misma traza, coincidiendo con la misma actividad en otra traza del alineamiento final. La fig. 5 muestra el procedimiento descrito.

El algoritmo se encuentra implementado en C/C++ empleando el entorno de desarrollo Visual Studio 2012. El compilador que se emplea es Visual C++ 11.0.

2.2. Antecedes del algoritmo Trace Alignment

El algoritmo para alinear trazas tiene como antecedentes la alineación de secuencias biológicas [3]. El siguiente acápite realiza un estudio del estado del arte de los algoritmos de este tipo que emplean técnicas de programación paralela.

Los artículos [5], [6], [10] presentan varios algoritmos paralelizados para alinear pares de secuencias biológicas. La técnica frente de onda (ver fig. 6) es ampliamente empleada en varios documentos como son [5]-[7]. Hace referencia a la matriz de alineamiento donde cada elemento a calcular depende de sus vecinos (ver paso 3). Básicamente lo que plantea la técnica, es calcular los elementos que se encuentren en la misma diagonal a la misma vez, puesto que ya debieron calcularse los elementos necesarios para el nuevo cálculo. La fig. 6 muestra dos secuencias $S_1 = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ y $S_2 = \{b_1, b_2, b_3, b_4, b_5, b_6\}$, los elementos representados con el mismo color,

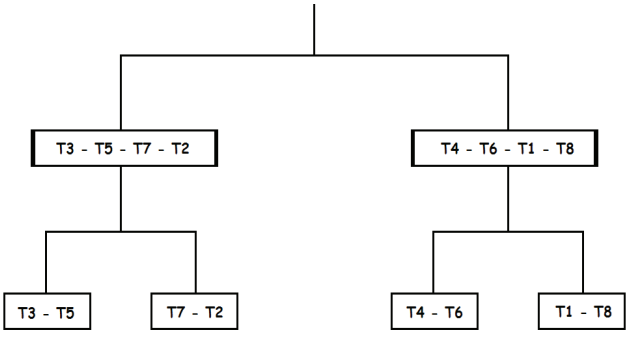

Fig. 7 Árbol guía para alinear trazas.

que pertenecen a cada diagonal, son los únicos que pueden calcularse paralelamente. Solo es posible calcular una diagonal en cada iteración.

Los algoritmos para alinear múltiples secuencias generalmente paralelizan la matriz de distancia empleando un ambiente distribuido y localmente cada nodo calcula los alineamientos de secuencias biológicas que le fueron asignados [11], [12].

2.3. Diseño paralelo del algoritmo Trace Alignment

El caso de estudio para el cual se analiza el diseño paralelo que se presentará, se enmarca en las trazas generadas por un proceso de negocio. Las trazas se guardan en bases de datos, de las cuales solo se alinean las trazas que sean diferentes. Las trazas más largas contienen solo 49 caracteres y las más cortas tienen 1 solo carácter. A pesar de que las bases de datos contienen miles de trazas solo diferentes tienen 4350.

Se propone en el presente artículo paralelizar la construcción de la matriz de distancia y alinear los pares o conjuntos de trazas a la misma vez; a continuación se describe el proceso seguido:

Matriz de distancia. La matriz S o matriz de distancia es una matriz de orden $N \times N$, donde N es el total de trazas a alinear. Cada uno de los elementos $S(i,j)$ de la matriz puede ser calculado independientemente. Se pretende calcular la distancia de Levenshtein por cada elemento $S(i,j)$ (la distancia entre T_1 y T_2 es la misma que entre T_2 y T_1) paralelamente.

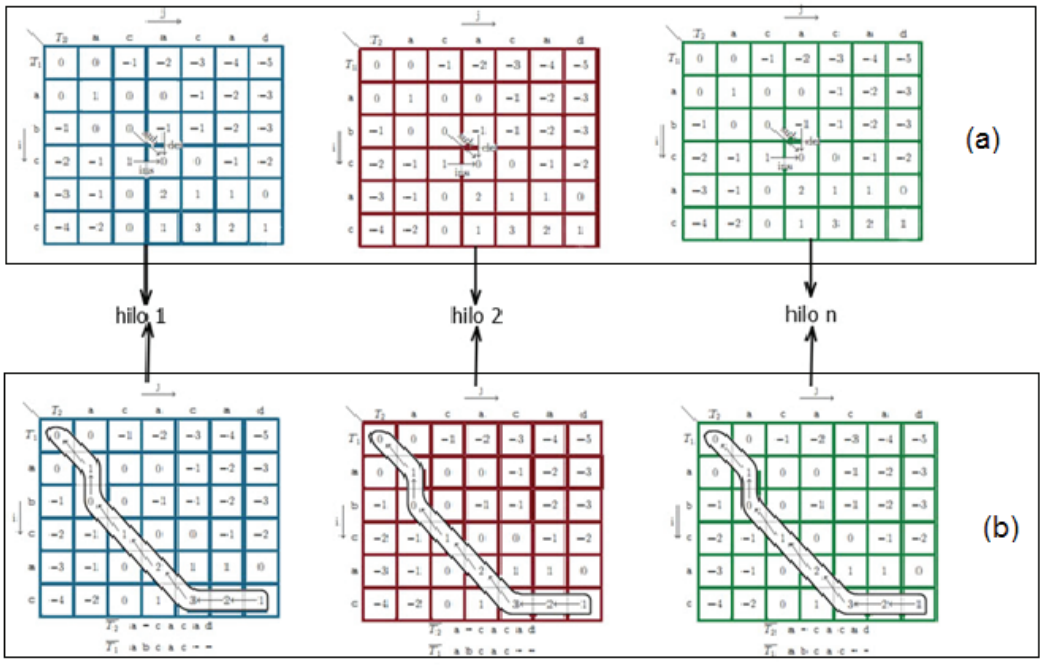

Fig. 8 Diseño propuesto para implementar paralelamente la alineación de las trazas.

Alinear pares o conjuntos de trazas. Los pares de trazas en el mismo nivel del árbol son independientes entre sí, por lo que pueden ser alineados paralelamente. La Fig. 7 muestra que las trazas T3-T5, T7-T2, T4-T6 y T1-T8 se pueden alinear a la misma vez. Una vez calculadas, se pueden alinear paralelamente también los conjuntos T3-T5-T7-T2 y T4-T6-T1-T8.

El principal problema es que a medida que se vayan alineando conjuntos, menor cantidad de elementos de procesamiento son necesarios. Tanto es así que la alineación del último conjunto se hará de forma secuencial. Es en este último paso donde más debe demorar la alineación, puesto que las trazas son más largas por la introducción de guiones y porque en este es donde se realiza el alineamiento final de solución.

Tabla 1.
Bibliotecas de programación

Memoria Compartida	Memoria Distribuida
OpenMP (CPU)	MPI (CPU)
Cuda (GPU)	PVM (CPU)
OpenCL (CPU-GPU)	

3. IMPLEMENTACIÓN DEL ALGORITMO PARALELO

3.1. Modelos de Programación Paralela

Existen dos modelos principales de programación paralela, aunque es posible emplear un híbrido entre los dos. Ambos modelos se especifican a continuación [13], [14]:

Modelo de memoria compartida: todos los procesadores comparten la misma zona de memoria.

Modelo de memoria distribuida o paso de mensajes: cada procesador tiene su zona de memoria que no puede ser accedida por el resto de los procesadores.

Tabla 2.
Hardware empleado para la realización de las pruebas

Hardware	Tipo	CPU/GPU			Memoria				
		Unidad de cómputo	Elemento de cómputo	Reloj (MHZ)	Tipo	Tamaño (MB)	Reloj (MHZ)	Ancho de BUS (bits)	Ancho de banda (GB/s)
Intel Core 2 Duo E7300	CPU	2	2	2600	DDR2	2048	800	64	12.4
Intel Core 2 Quad Q9300	CPU	4	4	2500	DDR2	4096	800	64	12.4
Intel Core i7 920	CPU	4	8	2670	DDR3	6144	1066	64	25.5
nVidia Geforce GTX 260	GPU	27	216	1242	DDR3	896	1998	448	111.9
nVidia Geforce GTX 550 Ti	GPU	4	192	1800	DDR5	1024	4104	192	98.5

Las bibliotecas de programación paralela clasifican dentro de estos modelos, algunas de las cuales se especifican en la Tabla 1. La selección de estas bibliotecas se realiza teniendo en cuenta el lenguaje de programación C++.

Una primera instancia de implementación del algoritmo se realiza empleando las bibliotecas descritas en la Tabla 1 clasificadas como modelo de memoria compartida.

La implementación del algoritmo empleando OpenMP [15-18], CUDA [19-21] y OpenCL[22-24] se caracteriza a continuación:

OpenMP: se emplean Visual C++ que brinda soporte para el estándar OpenMP 2.0.

CUDA: controlador gráfico de NVIDIA 334.89, CUDA GPU Computing SDK y Toolkit 5.5.20.

OpenCL: controlador gráfico de NVIDIA 334.89 e Intel SDK para aplicaciones OpenCL 2013 v3.0.

Debido al tamaño que alcanza la matriz de distancia (2.85 GB) se decide paralelizar el cálculo de los elementos de la matriz con OpenMP. Cada hilo del procesador disponible ejecutará [n(n-1)2]/p, siendo n la cantidad de trazas que se desea alinear y p la cantidad de hilos disponibles en la arquitectura de hardware.

La implementación de los pares o conjuntos de trazas a alinear se implementa con OpenMP, Cuda y OpenCL. La fig. 8 muestra el procedimiento seguido por cada hilo. Cada hilo, 1,2,.. n (Referente a la Fig. 8) debe calcular la matriz de alineamiento y luego obtener el resultado de la alineación.

La primera iteración requiere que cada hilo ejecute $(\frac{n}{2})/p$ pares de alineaciones, la segunda iteración $(\frac{n}{4})/p$, así sucesivamente hasta llegar a la última iteración. Si en cada una de las iteraciones, siendo $x = iteración + 1, (n/x) >= p$ todos los hilos trabajarían; si por el contario no se cumple esa condición en cada iteración solo se estaría aprovechando n/x hilos disponibles, aun cuando la arquitectura de hardware provea más.

El hecho de que los dispositivos Cuda y OpenCL no permitan la creación de memoria dinámica dentro de ellos requiere que se hagan modificaciones al código.

Teniendo en cuenta la fig. 8 (a) se necesita combinar las n matrices de alineamiento como un único arreglo lineal, conociendo siempre que parte del arreglo pertenece a cada matriz para que cada hilo pueda trabajar con ella. Esta matriz, en forma de arreglo, es la que se transfiere al dispositivo.

Dentro de los dispositivos también se obtiene el resultado del nuevo conjunto alineado (procedimiento *TraceBack* explicado en el paso 3), representado en la fig. 8 (b).

Cómo cada nivel $i + 1$ del árbol guía (ver fig. 7) requiere los resultados obtenidos en los niveles i del árbol, por cada nivel de él, se requieren transferencias entre el dispositivo y la memoria RAM de envío y recibo.

3.2. Resultados y discusión

El acápite valida la implementación realizada, para lo cual se realizan tres experimentos. La métrica empleada para evaluar los experimentos es la aceleración (Speed UP) obtenida en los escenarios de prueba. La métrica indica cuántas veces es más rápido el

Tabla 3.
Tiempos de ejecución obtenidos (segundos) para cada una de las implementaciones realizadas con 350 trazas

	Tiempo de ejecución(segundos)					
	Arquitectura 1		Arquitectura 2		Arquitectura 3	
	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas
Secuencial	0.0970	0.2014	0.0981	0.2570	0.0512	0.1410
OpenMP	0.0530	**0.1504**	0.0404	0.2686	**0.0375**	0.1540
OpenCL CPU	0.0530	1.3095	0.0404	1.4026	0.0375	1.2722
OpenCL GPU	0.0530	1.5560	0.0404	6.5064	0.0375	6.1552
CUDA	0.0530	1.4100	0.0404	1.6422	0.0375	5.0500

programa paralelo comparado con el programa secuencial. La fórmula de la aceleración es la siguiente [25]-[28]: $S = \frac{Ts}{Tp}$

Siendo Ts el tiempo de ejecución secuencial y Tp el tiempo de ejecución paralelo.

3.2.1. *Escenarios de pruebas*

Además de las métricas descritas se realizaron varios escenarios de pruebas los cuales se detallan a continuación:

Arquitectura 1: Intel Core 2 Duo y una tarjeta de video nVidia Geforce GTX 550 Ti.

Arquitectura 2: Intel Core 2 Quad y una tarjeta de video nVidia Geforce GTX 260.

Arquitectura 3: Intel Core i7 y una tarjeta de video nVidia Geforce GTX 260.

Para la realización de las pruebas se emplean las arquitecturas descritas, las cuales se detallan en la Tabla 2. Los dispositivos pertenecen a distintos fabricantes, con generaciones y arquitecturas que surgieron entre el 2007 y el 2012.Los resultados obtenidos se refieren solo al procesamiento. Los tiempos de ejecución que se exponen son el promedio de 10 ejecuciones del algoritmo para cada escenario descrito. El sistema operativo donde se ejecutaron las pruebas es Windows 7 de 64 bits.

3.2.2. *Análisis de los resultados obtenidos*

El objetivo fundamental de las pruebas realizadas es comprobar el objetivo de investigación planteado: disminuir el tiempo de ejecución del algoritmo implementado respecto al secuencial.

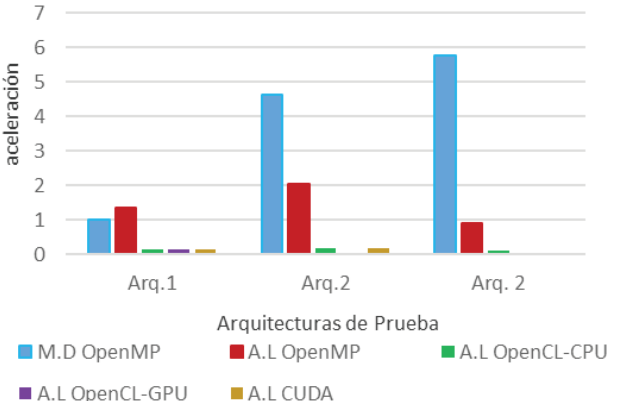

Fig. 9 Aceleración obtenida en las arquitecturas de prueba con 350 trazas.

Experimento 1:

A continuación se procede a realizar los experimentos con 350 trazas. Las trazas más largas tienen 10 caracteres mientras que las más cortas solo tienen 1 carácter, con promedio de 5 caracteres. Los tiempos que se exponen en la Tabla 3 exponen los tiempos obtenidos en las dos secciones paralelizadas, la matriz de distancias (T. Matriz D) y la alineación de las trazas (T. AL trazas).

Al analizar los resultados descritos en la Tabla 3 se observa que para todas las arquitecturas de prueba empleada, el tiempo de ejecución de la matriz de distancia es inferior a su tiempo secuencial. La alineación de las trazas solo disminuye cuando se emplea OpenMP en la arquitectura 1 y 2. Se concluye que para esta cantidad de patrones la paralelización de la alineación de las trazas empelando OpenCL y CUDA el tiempo de ejecución no disminuye. Los tiempos logrados con GPU superan en gran medida al tiempo

Tabla 4.

Tiempos de ejecución obtenidos (segundos) para cada una de las implementaciones realizadas con 2350 trazas

	Tiempo de ejecución(segundos)					
	Arquitectura 1		Arquitectura 2		Arquitectura 3	
	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas
Secuencial	4.3502	148.7538	5.7206	212.4726	3.9956	200.7250
OpenMP	2.5944	**147.4612**	1.9157	152.7680	**0.9724**	200.2720
OpenCL CPU	2.5944	215.0320	1.9157	237.5774	0.9724	230.5660
OpenCL GPU	2.5944	588.7847	1.9157	-	0.9724	-
CUDA	2.5944	585.5305	1.9157	-	0.9724	-

secuencial. Uno de los factores que influye en esto, es la cantidad de datos transferidos a través de la PCI entre el dispositivo y la memoria RAM.

La fig. 9 muestra la aceleración obtenida en cada arquitectura de prueba. La barra azul representa la aceleración obtenida con la paralelización de matriz de distancia con OpenMP (M.D), y las restantes barras son la aceleración obtenida con las implementaciones OpenMP, OpenCL-CPU, OpenCL-GPU y CUDA de la alineación de las trazas (A.L). La arquitectura que mejor aceleración logra disminuir el tiempo con la matriz de distancia es la 3. Las implementaciones realizadas a la alineación de las trazas no logran disminuir el tiempo de ejecución secuencial en la mayoría de los casos. La única implementación que logra aceleración es OpenMP para la arquitectura 1 y 2, en las restantes arquitectura la aceleración obtenida es inferior a 1.

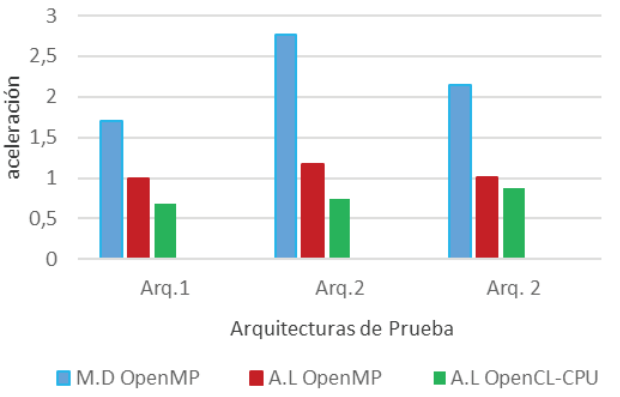

Fig. 10 Aceleración obtenida en las arquitecturas de prueba con 2350 trazas.

Experimento 2:

A continuación se procede a realizar los experimentos utilizando 2350 trazas. Las trazas más largas tienen 26 caracteres mientras que las más cortas solo tienen 1 carácter, con promedio de 24 caracteres. Los tiempos que se exponen en la Tabla 4 exponen los tiempos obtenidos en las dos secciones paralelizadas, la matriz de distancias (T. Matriz D) y la alineación de las trazas (T. AL trazas).

En la Tabla 4 no se muestran resultados que la implementación realiza respecto a la alineación de las trazas para la arquitectura 2 y 3. La GPU instalada en ambas arquitecturas es la misma (ver Tabla 2). Al ejecutar el algoritmo en estas arquitecturas, la GPU no fue capaz de ejecutar tantas alineaciones de trazas a la misma vez. La cantidad de trazas a alinear es superior al experimento anterior, por lo que se esperaba que la GPU aumentara el tiempo de ejecución del algoritmo como se evidencia para la arquitectura 1 (OpenCL y CUDA).

OpenMP es la única técnica que logra disminuir el tiempo de ejecución de esta implementación. Respecto a la paralelización de la matriz de distancia, para todas las arquitecturas se disminuye el tiempo de ejecución.

Se concluye que el proceso de alinear varios conjuntos de trazas a la misma vez dentro de la GPU, debido a las características descritas del algoritmo no es recomendable para este algoritmo. Muchas bifurcaciones afectan el rendimiento de las GPUs, la ejecución se serializa entre los hilos de un mismo grupo, por lo que se ejecutan pocos hilos de forma simultánea, esto provoca que los accesos a memoria sean muy dispersos.

Tabla 5.
Tiempos de ejecución obtenidos (segundos) para cada una de las implementaciones realizadas con 4350

	Tiempo de ejecución(segundos)					
	Arquitectura 1		Arquitectura 2		Arquitectura 3	
	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas	T. Matriz D	T. Al de trazas
Secuencial	16.8490	2262.5640	15.9205	2417.6380	13.4015	3359.9725
OpenMP	8.7395	**2202.6020**	4.6360	2312.6980	**3.0100**	3349.7100
OpenCL CPU	8.7395	2753.4120	4.6360	2904.8920	3.0100	3195.4410
OpenCL GPU	8.7395	-	4.6360	-	3.0100	-
CUDA	8.7395	-	4.6360	-	3.0100	-

La fig. 10 solo muestra la aceleración obtenida para las implementaciones OpenMP y OpenCL CPU. La barra azul muestra la aceleración lograda con la paralelización de la matriz de distancia. Las restantes hacen alusión a la implementación realizada a la alineación de las trazas (A.L). La arquitectura 2 es la que más logra disminuir el tiempo de ejecución de la matriz de distancia. No se obtiene, para ninguna de las arquitecturas de prueba, aceleración respecto a la alineación de las trazas con OpenCL (CPU). La aceleración obtenida con OpenMP disminuye relativamente poco el tiempo de ejecución de la implementación secuencial de la alineación de las trazas. A pesar de ello la arquitectura 2 es la que mejores resultados logra respecto a esta implementación.

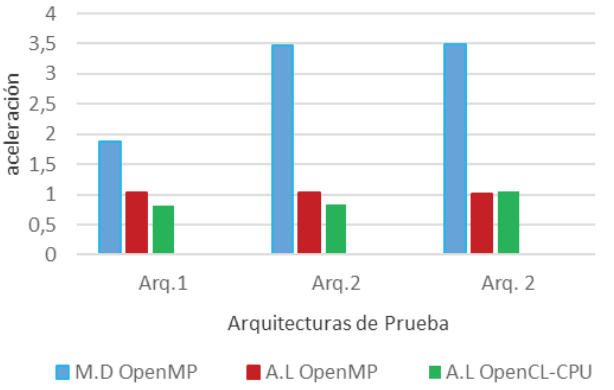

Fig. 11 Aceleración obtenida en las arquitecturas de prueba con 4350 trazas.

Experimento 3:

A continuación se procede a realizar los experimentos utilizando 4350 trazas. Las trazas más largas tienen 49 caracteres mientras que las más cortas solo tienen 1 carácter. Los tiempos que se exponen en la Tabla 5 exponen los tiempos obtenidos en las dos secciones paralelizadas, la matriz de distancias (T. Matriz D) y la alineación de las trazas (T. AL trazas).

La Tabla 5 no muestra resultados para ninguna de las ejecuciones empleando GPU (OpenCL y CUDA). El controlador de video falla al realizar las alineaciones con esta cantidad de datos. Al recuperarse el algoritmo no termina de forma abrupta pero los resultados obtenidos no son correctos. En todas las arquitecturas de prueba se logra disminuir el tiempo de ejecución a la matriz de distancia. OpenCL CPU solo disminuye tiempo en la arquitectura 3. OpenMP a pesar de que en todas las arquitecturas logra disminuir el tiempo de la alineación de trazas, el tiempo no es significativo.

La fig. 11 muestra la aceleración obtenida para la implementación realizada a la matriz de distancia. Solo para OpenMP y OpenCL-CPU se muestra la aceleración obtenida para la alineación de las trazas. La barra azul representa la aceleración obtenida con la matriz de distancia. Todas las arquitecturas logran disminuir el tiempo de ejecución.

La aceleración obtenida con las implementaciones OpenMP y OpenCL-CPU para la alineación de las trazas, es casi imperceptible y con OpenCL-CPU solo se logra aceleración con la arquitectura 3.

Se concluye que a medida que la arquitectura ofrece mejores componentes de hardware, la implementación realizada con OpenMP respecto a la matriz de distancia, obtiene mejores resultados. Lo cual demuestra ser un algoritmo escalable.

Respecto a la alineación de las trazas, las implementaciones realizadas no logran disminuir en la mayoría de ellas, el tiempo de ejecución del algorit-

mo secuencial. Solo la implementación con OpenMP obtiene una aceleración, para todas las arquitecturas de prueba, aproximadamente 1.

4. CONCLUSIONES

Teniendo en cuenta que el objetivo de la investigación es disminuir el tiempo del algoritmo Trace Alignment se concluye lo siguiente:

- El diseño propuesto implementado con OpenMP, OpenCL y CUDA solo logra disminuir tiempo de ejecución con OpenMP.
- El tiempo de ejecución del algoritmo Trace Alignment paralelizado (OpenMP) disminuye aproximadamente 5 minutos respecto al tiempo secuencial.
- De las secciones implementadas con técnicas paralelas, la sección que más disminuye tiempo de ejecución es la matriz de distancia, a pesar de no ser la región que más consume tiempo dentro del algoritmo.
- El tiempo alcanzado empleando las GPUs disponibles es siempre superior al algoritmo secuencial. A medida que se aumenta la cantidad de trazas a alinear, las GPUs tienen muy bajo rendimiento.
- Los flujos complejos en las operaciones presentes en el kernel influyen negativamente en el rendimiento de este tipo de dispositivos.

REFERENCIAS

[1] W. Van Der Aalst, “Process mining: discovery, conformance and enhancement of business processes: Springer Science & Business Media,” 2011. ISBN: 978-3-642-19344-6.

[2] A. Weijters, W. M. Van Der Aalst, and A. A. De Medeiros, “Process mining with the heuristics miner-algorithm,” *Technische Universiteit Eindhoven, Tech. Rep. WP,* vol. 166, pp. 1-34, 2006. Available: doi: 10.1.1.118.8288

[3] R. J. C. Bose and W. M. van der Aalst, “Process diagnostics using trace alignment: opportunities, issues, and challenges,” *Information Systems,* vol. 37, pp. 117-141, 2011. Available: doi: 0.1016/j.is.2011.08.003

[4] N. Carriero and D. Gelernter, “How to Write Parallel Programs: A First Course 1st ed.: MIT Press,” 1992. ISBN: 0-262-03171-X

[5] Z. Du, Z. Yin, and D. Bader, “A tile-based parallel Viterbi algorithm for biological sequence alignment on GPU with CUDA,” presented at the Proc. of the *IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW)*, Atlanta, GA, 2010. Available: doi: 10.1109/IPDPSW.2010.5470903

[6] T. Siriwardena and D. Ranasinghe, “Accelerating global sequence alignment using CUDA compatible multi-core GPU,” in *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on, Colombo,* 2010, pp. 201-206. Available: doi: 10.1109/ICIAFS.2010.5715660

[7] L. Hasan, M. Kentie, and Z. Al-Ars, “GPU-accelerated protein sequence alignment,” in *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE,* 2011, pp. 2442-2446. ISBN: 978-1-4244-4121-1

[8] M. R. Babu, “Parallelized hierarchical expected matching probability for multiple sequence alignment,” *Journal of Theoretical & Applied Information Technology,* vol. 64, 2014. Available: http://www.jatit.org/volumes/Vol64No2/10Vol64No2.pdf

[9] A. E. C. González, “La métrica de Levenshtein,” *Revista de Ciencias Básicas UJAT,* vol. 7, pp. 35-43, 2008. Available: www.publicaciones.ujat.mx/publicaciones/revista_dacb/Acervo/v7n2OL/v7n2.pdf#page=37

[10] D. Shrimankar and S. Sathe, “Performance Analysis of OpenMP and MPI for NW algorithm on multicore architecture,” *International Journal of Advanced Studies in Computers, Science and Engineering,* vol. 3, p. 23, 2014. Available: http://www.ijascse.org/volume-3-issue-6/OpenMp.pdf

[11] Sharma C and V. A. K., Parallel Approaches in Multiple Sequence Alignments. *International Journal of Advanced Research in Computer Science and Software Engineering 4(2),* 2014. Available: http://www.ijarcsse.com/docs/papers/Special_Issue/icadet2014/Lord_35.pdf

[12] A. Y. Zomaya, Parallel computing for bioinformatics and computational biology: models, *enabling technologies, and case studies* vol. 55: John Wiley & Sons, 2006. ISBN: 978-0-471-71848-2 ISBN: 978-1-4613-6601-0

[13] D. E. Lenoski and W.-D. Weber, *Scalable shared-memory multiprocessing: Elsevier*, 2014. ISBN: 978-1-4613-6601-0

[14] N. Matloff, "Programming on Parallel Machines University of California: Davis," 2012. Available: http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf

[15] A. OpenMP, "The OpenMP API specification for parallel programming," 2010. Available: http://openmp.org, 2010.

[16] B. Chapman, G. Jost, and R. Van Der Pas, "Using OpenMP: portable shared memory parallel programming," vol. 10: MIT press, 2008. ISBN: 978-0-262-53302-7

[17] OpenMP and A. R. Board, "OpenMP Application Program Interface", 3.0 ed., 2008. Available: http://www.openmp.org/mp-cuments/spec30.pdf

[18] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *Proceedings of the 15th international symposium on System Synthesis,* 2002, pp. 109-111. ISBN: 1-58113-576-9

[19] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs: Newnes*, 2012. ISBN: 978-0124159334

[20] R. Farber, *CUDA application design and development: Elsevier*, 2011. ISBN: 978-0-12-388426-8

[21] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*, 2010. ISBN: 978-0131387683

[22] AMD, "Accelerated Parallel Processing OpenCL Programming Guide: AMD," 2012. Available: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[23] B. e. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 2nd ed.: Morgan Kaufmann, 2013. ISBN: 970-0-12-405894-1

[24] M. Scarpino, "Opencl in Action: How to Accelerate Graphics and Computation. NY," ed: USA: Manning, 2012. ISBN: 9781617290176

[25] P. S. Pacheco, *An Introduction to Parallel Programming,* 1st ed., Morgan Kaufmann, 2011. ISBN: 0080921442

[26] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of Computer Organization and Architecture,* 1st ed. New Jersey, USA: Wiley-Interscience, 2004. ISBN: 0-471-4674-1-3

[27] F. Gebali, *Algorithms and Parallel Computing,* 1st ed., Wiley, 2011. ISBN: 978-0-470-90210-3

[28] A. Grama, A. Gupta, G. Karyspis, and V. Kumar, *Introduction to Parallel Computing,* 2nd ed., Addison Wesley, 2003. ISBN: 978-0201648652