Enfoque UTE, V.8-N.2, Mar.2017, pp.78 - 93
http://ingenieria.ute.edu.ec/enfoqueute/
e-ISSN: 1390-6542 / p-ISSN: 1390-9363
Recibido (Received): 2017/01/19
Aceptado (Accepted): 2017/03/27

Aplicación del modelo de programación CUDA en la simulación de la evolución de secuencias genéticas

(Application of the CUDA programming model in the simulation of genetic sequences evolution)

Fredy Yasmany Chávez[1], Daniel Gálvez Lio[1,2]

Resumen:
La simulación resulta un poderoso enfoque en el estudio de la evolución molecular de secuencias genéticas y su divergencia a lo largo del tiempo; existen diferentes procedimientos de simulación de la evolución molecular, pero todos ellos poseen alta complejidad computacional, y en la mayoría de los casos las secuencias genéticas poseen gran tamaño, aumentando los tiempos de ejecución de las implementaciones de estos procedimientos. A partir de esta problemática, en este trabajo se describe una propuesta de modelo de paralelización utilizando la tecnología CUDA y los resultados de esta propuesta se comparan con su equivalente secuencial.

Palabras clave: simulación; evolución molecular; Markov; programación paralela; CUDA

Abstract:
Simulation is a powerful approach in the study of the molecular evolution of genetic sequences and their divergence over time; there are different procedures for the simulation of molecular evolution, but all of them have high computational complexity, and in most cases the genetic sequences have large size, increasing the execution time of the implementations of those procedures. Based on this problem, this paper describes a proposal of parallelization model using CUDA technology and the results of this proposal are compared with its sequential equivalent.

Keywords: simulation; evolution model; Markov; parallel programming; CUDA

1. Introducción

La simulación es un enfoque sumamente útil en la realización de estudios y en la validación de teorías o programas cuando los métodos de análisis conocidos son complejos o los modelos son analíticamente intratables; el desarrollo explosivo de los medios de cómputo en los últimos años ha permitido la aplicación de este poderoso enfoque de estudio a diferentes áreas de conocimiento. Una de estas áreas favorecidas es la biología molecular, en especial los estudios sobre evolución molecular y procesos evolutivos, posibilitando la realización de experimentos virtuales que imitan estos procesos biológicos con el objetivo de estudiar sus propiedades, centrándose fundamentalmente en la reconstrucción de las relaciones evolutivas entre las especies, y en investigaciones sobre los mecanismos y fuerzas del proceso evolutivo. El proceso de simular la evolución molecular no solo posee alta complejidad computacional, en la gran

[1] Universidad Central “Marta Abreu” de Las Villas, Santa Clara – Cuba (yasmany@uclv.cu)
[2] Universidad Metropolitana del Ecuador, Quito – Ecuador (d.galvez@umet.edu.ec)

mayoría de los casos, las bases de datos de secuencias genéticas constituyen grandes volúmenes de datos, por lo cual se impone la necesidad de buscar nuevos métodos o tecnologías que permitan obtener buenos resultados, pero en tiempos de ejecución razonables.

Una solución a este problema es el uso de la programación paralela, específicamente se aborda el uso de la tecnología CUDA, que dado su alto desempeño se hace cada vez más interesante para acelerar aplicaciones científicas en diversas ramas, entre ellas la Biología Computacional y la Bioinformática. CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de programación inventado por NVIDIA ("CUDA Zone | NVIDIA Developer," 2011; Nvidia, 2011; NVIDIA, 2015) permite aumentos impresionantes en el rendimiento del computador al aprovechar las unidades de procesamiento gráfico (GPU) alojadas en la tarjeta gráfica. Actualmente se reportan varias aplicaciones de esta tecnología en el área de la Bioinformática: en el análisis de genes en microarreglos y alineación de secuencias (Yongchao Liu, Schmidt, & Maskell, 2012), simulación de sistemas biológicos y búsqueda de secuencias en bases de datos (Y Liu, Wirawan, & Schmidt, 2013), entre otras aplicaciones disponibles en el sitio para desarrolladores de NVIDIA ("CUDA Zone | NVIDIA Developer," 2011), (Hwu, 2012), (Schatz, Trapnell, Delcher, & Varshney, 2007), (Manavski & Valle, 2008).

El uso de CUDA en cualquier ordenador puede convertirlo en un "superordenador" ya que potencia de forma importante el trabajo de los numerosos procesadores que están alojados en el núcleo de su tarjeta gráfica. El hecho de que el procesamiento de los datos se realice en paralelo hace que la velocidad de ejecución sea 400 veces superior a la de un ordenador sin este software (Claver, Sanjuan, & Arnau, 2007).

En este artículo, a partir de la problemática presentada, y con el objetivo de reducir los tiempos de ejecución del proceso de simulación de la evolución molecular de secuencias genéticas, se plantea un modelo de paralelización utilizando la tecnología CUDA y se analiza la efectividad de su implementación computacional.

2. Materiales y Métodos

Procedimientos de simulación basados en la técnica de Markov

Para la realización de simulaciones de la evolución molecular de secuencias genéticas han sido utilizados diferentes métodos de Markov (Weber, 2012), (Link & Eaton, 2012), dadas las capacidades de este tipo de simulación para representar procesos estocásticos y la divergencia genética a lo largo del tiempo. Todos estos métodos clásicos (Yang, 2006) requieren como entradas un árbol filogenético con una longitud de una rama específica, una secuencia genética de raíz, y un modelo Markoviano de la evolución molecular. La simulación es realizada por cada rama del árbol filogenético, empezando por la raíz y terminando en las puntas de las hojas.

Tiempo de espera evolutivo

En (Counsell, 2005) se plantea que un proceso de Markov de tiempo continuo, determina el tiempo de espera evolutivo entre las sustituciones, o sea, el período que el sitio se mantiene en el estado i, y que este es distribuido exponencialmente con el parámetro |q*ii|. Dado que este último es la tasa de sustitución de un nucleótido determinado i, es fácil construir una variable aleatoria de distribución exponencial a partir de la otra uniformemente distribuida entre 0 y 1 (u) utilizando la ecuación 1:

$$t = -\frac{1}{|q^*_{ii}|}\log_e(u) \quad [1]$$

Sea T el conjunto representado por los tiempos transcurridos para cada sitio de la secuencia, durante cada paso de la simulación, y t el tiempo de espera generado utilizando dicho número aleatorio. Si se cumple que t > T entonces se mantendrá el sitio en su estado actual, de lo contrario, ocurre una sustitución nucleotídica. Cuando t < T, o sea, que va a ocurrir una mutación, el nucleótido de remplazo es seleccionado de acuerdo con la cadena de Markov de tiempo discreto $D^* = \{d^*_{ij}\}$ definida por la ecuación 2:

$$d^*_{ij} = \begin{Bmatrix} 0, & i = j \\ \frac{q^*_{ij}}{q^*_{ii}}, & i \neq j \end{Bmatrix} \quad [2]$$

El nucleótido sustituto, se obtiene igualmente generando un número aleatorio, distribuido uniformemente entre 0 y 1. Si este número aleatorio es menor que d, el nucleótido 1 sustituye al nucleótido i. De lo contrario, si el número aleatorio es menor que d*i1 + d*i2, el nucleótido 2 sustituye al i. Este proceso se repite hasta que se encuentre el nucleótido de sustitución. Por el hecho que se cumple que: $\sum_j d^*_{ij} = 1$, siempre se encontrará un nucleótido de sustitución, y dado que $d^*_{ii} = 0$, el nucleótido sustituido no será el mismo.

Cadena de salto

En este método se ignoran los tiempos de espera (exponenciales con media 1/|q*ii|) entre transiciones, la secuencia de los estados constituirá una cadena de Markov de tiempo discreto, con matriz de transición que se muestra en la ecuación 3:

$$M = \begin{bmatrix} 0 & \frac{q_{AC}}{q_A} & \frac{q_{AG}}{q_A} & \frac{q_{AT}}{q_A} \\ \frac{q_{CA}}{q_C} & 0 & \frac{q_{CG}}{q_C} & \frac{q_{CT}}{q_C} \\ \frac{q_{GA}}{q_G} & \frac{q_{GC}}{q_G} & 0 & \frac{q_{GT}}{q_G} \\ \frac{q_{TA}}{q_T} & \frac{q_{TC}}{q_T} & \frac{q_{TG}}{q_T} & 0 \end{bmatrix} \quad [3]$$

Tanto el método de los tiempos de espera evolutivo, como la cadena de salto, pueden ser aplicados a toda la secuencia (todos los sitios en lugar de uno), con tasa de sustitución igual a la sumatoria de las tasas en todos los sitios, y con un tiempo de espera hasta que una sustitución ocurra en cualquier sitio de toda la secuencia, que sigue una distribución exponencial de media igual a la inversa de la tasa total, para cada uno de los sitios de la secuencia.

Metropolis-Hasting (MH)

Este último método es "conocido como el más popular de la familia de métodos Markov Chain Monte Carlo, y comúnmente usado para la evolución de secuencias" (Andrieu & Doucet, 2010; Cornebise & Peters, 2009). MH consiste básicamente en generar un nuevo valor candidato x* dado el valor actual x, de acuerdo con la probabilidad de distribución propuesta q(x*|x). La probabilidad de aceptación que la cadena de Markov se mueva hacia x* está dada por la ecuación 4, en el cual p(x) es la distribución invariante definida en x:

$$p(x, x^*) = \min\left\{1, \frac{p(x^*)q(x\,|x^*)}{p(x)q(x^*\,|x)}\right\} \quad [4]$$

Existen varios algoritmos prácticos interpretados como casos especiales o extensiones de este método, cuya principal diferencia es la especificación de la probabilidad de distribución propuesta q(x*|x). Es necesario destacar, que las probabilidades de distribución y mutación deben ser determinadas para un nuevo estado, siempre que ocurra una mutación de nucleótido, o que el valor candidato x* sea aceptado acorde con la probabilidad previamente definida para el nuevo nucleótido. Por el contrario, si la cadena de Markov se mantiene en el estado actual con el mismo valor nucleotídico, no existirá la necesidad de cambiar las probabilidades de distribución y mutación de la base en ese sitio.

Los métodos de tiempo de espera evolutivo y de cadena de salto poseen la ventaja de no requerir el cálculo de la matriz de probabilidades P(t) sobre el tiempo t requerida por MH, puesto que, tanto la matriz de transición para la cadena de salto como los tiempos de espera, se especifican en su totalidad por la matriz de tasas instantáneas Q* definida por el modelo evolutivo Markoviano seleccionado para la simulación de la evolución molecular de las secuencias. Los tres métodos descritos con anterioridad no son los únicos posibles, de hecho, estos tres procedimientos en la práctica son comúnmente combinados dando lugar a nuevos.

Simulación basada en el método de MH y el tiempo de espera

Mediante la estimación del tiempo de espera evolutivo se evalúa la ocurrencia o no de una mutación en un sitio determinado. Dado el tiempo evolutivo T transcurrido en cada paso de la simulación, si t < T, o sea, que hay un cambio de nucleótido en el sitio dado, se aplica el método de MH de forma repetitiva hasta que un nuevo nucleótido de sustitución sea encontrado.

Simulación basada en el tiempo de espera y la matriz de transición

Mediante el proceso de simulación del tiempo de espera se determinan los nucleótidos que cambiarán, de la misma forma que en el procedimiento anterior. En caso que haya cambio de nucleótido, se buscará el nucleótido de remplazo mediante la matriz de transición de la cadena de salto, de la siguiente manera: al suponer que i es el nucleótido de partida ocupado en el sitio dado (el estado actual del proceso es i), a partir de la matriz de tasas de sustitución Q*, se obtiene la fila correspondiente a ese nucleótido. Los valores en esta fila constituyen una cadena de Markov de tiempo discreto $D^* = \{d^*_{ij}\}$, cuya definición fue descrita con anterioridad.

Simulación basada en el método de MH con repetición

Mediante el uso de MH se determinará el cambio de nucleótido en cada sitio de la secuencia. Posteriormente, mediante la matriz de probabilidades de sustitución P(t), se obtendrá la fila correspondiente al nucleótido ocupado en el sitio dado, esta fila es el vector de distribución de probabilidades de ese nucleótido y se denota por p. Para saber si ocurre la mutación del nucleótido actual i, se genera un número aleatorio con distribución uniforme entre 0 y 1 llamado u. Si se cumple que u < p(i), no ocurrirá cambio en el sitio dado, en caso contrario, se buscará un nucleótido de sustitución diferente al nucleótido actual mediante la aplicación repetitiva del método MH, garantizando el cambio del proceso a un nuevo estado.

Simulación basada en el método de MH sin repetición

Este procedimiento no examina si una mutación ocurre o no en un sitio determinado. El método de MH se aplica una sola vez para cada sitio de la secuencia, y el nucleótido puede cambiar su valor a otro diferente o mantenerse en su estado actual, según el valor devuelto por MH. Si MH retorna una base igual que la actual en el sitio dado, se dice que no hay mutación. De otro modo, el nucleótido ancestral se remplaza por el nuevo.

Modelos Markovianos de la evolución molecular

Para estimar el número real de sustituciones entre secuencias relativamente alejadas en el proceso evolutivo, se necesita un modelo probabilístico que describa los cambios entre nucleótidos. Las cadenas continuas de Markov son la herramienta más usada con este propósito (Yang, 2006), donde normalmente se asume que cada sitio de la secuencia evoluciona de forma independiente. Las sustituciones en cada sitio en particular son descritas mediante una cadena de Markov, que tiene como estados las bases nucleotídicas. La imposición de algunas restricciones en las tasas de sustitución entre nucleótidos lleva a diferentes modelos. En la literatura (Yang & Rodríguez, 2013), han sido diversos los modelos de Markov propuestos para la evolución de las secuencias, y se han desarrollado también, varias técnicas para estimar experimentalmente los parámetros de dichos modelos de Markov mediante el análisis de secuencias reales observadas.

Según Yang (Yang, 2006), de forma general, los modelos evolutivos describen el modo y la probabilidad que una secuencia de nucleótidos cambie a otra secuencia de nucleótidos homóloga a lo largo del tiempo. Estos modelos describen para cada uno de los sitios de la matriz, la probabilidad que se produzca el cambio de un nucleótido a otro a lo largo de las ramas de un árbol filogenético dado.

Los modelos de evolución de nucleótidos se definen matemáticamente mediante dos clases de parámetros que determinan el cambio:

1) Frecuencia de cada nucleótido: parámetro que mide la frecuencia de cada nucleótido en la matriz de datos. En los modelos evolutivos más sencillos, toma una misma frecuencia para los cuatro nucleótidos ($\pi A = \pi C = \pi G = \pi T = 0.25$), sin tener en cuenta la frecuencia de aparición de los mismos en la matriz de datos; En el caso de los modelos más complejos asume que las frecuencias asociadas a cada uno de los nucleótidos son diferentes y son calculadas a partir de los datos ($\pi A \neq \pi C \neq \pi G \neq \pi T$).
2) Tipos de sustituciones y sus correspondientes tasas de sustitución (rate parameters): las tasas de sustitución se representan con las tasas relativas de cambio de un nucleótido a otro para una posición de un tiempo t0 a un tiempo t1. Así, cada posición de la matriz de tasas de sustitución tendrá una probabilidad asociada de cambio para cada unidad de tiempo (unidad de distancia evolutiva). Los modelos más sencillos asumen una misma tasa relativa para todas las sustituciones posibles, mientras que los más complicados asumen una tasa relativa diferente para cada tipo de sustitución. A partir de estas tasas relativas se calcula la tasa media de sustitución (μ). La matriz de tasas más general de un modelo de Markov tiene solo el requisito de que los elementos no diagonales son positivos y la suma de cada fila es cero.

Modelo de programación en CUDA

La programación en CUDA está basada en tres abstracciones básicas: una jerarquía de grupos de hilos, otra de tipos de memoria y barreras de sincronización. La estructura que conforma la jerarquía de hilos que se ejecutan en el dispositivo NVIDIA (device) se agrupan, en tres elementos: mallas, bloques e hilos. Una malla está conformada por múltiples; un bloque está formado por un grupo de hilos, y el hilo es la unidad elemental, a continuación, en la *Figura 1* se muestra esta jerarquía.

Como el dispositivo NVIDIA (tarjeta gráfica) está colocado en una computadora se establece un modelo “híbrido” para la programación en el que se combina el código del programa principal que se ejecuta en la CPU de la computadora anfitrión (host) y las llamadas a funciones que se ejecutan en el dispositivo NVIDIA (device). El proceso comienza en un programa que se ejecuta en la CPU anfitriona y este programa a su vez invoca funciones que se ejecutarán paralelamente por varios hilos en la GPU, conocidas como funciones kernel, especificando para cada una de

ellas la configuración de ejecución que define la dimensión de la malla de bloques (dimGrid) y la cantidad de hilos por bloques (dimBlock) con que será ejecutada la función kernel en la GPU.

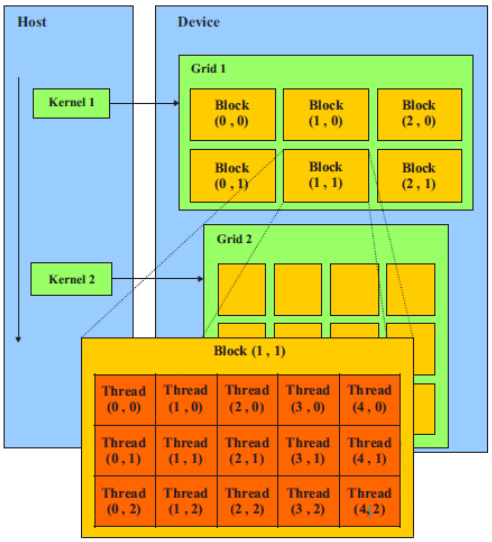

Figura 1. Jerarquía del lote de hilos (Tomado de (Nvidia, 2011))

Los hilos que se ejecutan en un dispositivo CUDA tienen acceso a múltiples espacios de memoria: global, local, compartida, textura y registros. Cada hilo tiene un espacio de memoria local privado; a su vez, cada bloque de hilos posee memoria compartida visible solo a todos los hilos del bloque y con la misma duración de vida que el bloque. Todos los hilos ejecutados por una función kernel tienen acceso a la misma memoria global. Los espacios de memoria global, constante y de textura son persistentes a diferentes activaciones de funciones kernels en la misma aplicación. El trabajo de estos hilos puede ser sincronizado mediante directivas de bloqueo, lo cual posibilita la coordinación entre estos; sin embargo, hilos agrupados en diferentes bloques no pueden comunicarse entre sí.

La estructura general de una aplicación CUDA incluye los pasos siguientes:

1) El computador (CPU) anfitrión ejecuta el cuerpo principal del programa (main())
2) Se reserva memoria dentro del dispositivo GPU
3) Se copian los datos del CPU anfitrión al dispositivo GPU
4) El CPU anfitrión llama a la función kernel del GPU
5) El dispositivo GPU ejecuta el código de manera paralela
6) Se copian los resultados de vuelta a la memoria del CPU anfitrión
7) Se libera la memoria reservada dentro del dispositivo GPU
8) Repetir los pasos del 2 al 7 tantas veces como sean necesarios para la solución del problema

Debido a los tiempos relativamente grandes de retardo (latencia) y al bajo ancho de banda en las transferencias de memoria entre la computadora anfitrión y el dispositivo GPU, es recomendable dividir la aplicación, de tal manera que cada componente de *hardware* haga únicamente el trabajo

que mejor realiza. El uso del GPU es solamente recomendado si (Nvidia, 2011), (Sánchez, Carbajal, Cortés, & Fernández, 2012):

- La complejidad de las operaciones a realizar en la GPU justifica el costo de mover datos desde y hacia el dispositivo GPU. Se deben minimizar las transferencias y mantener en GPU los datos tanto tiempo como sea posible.
- La aplicación realizará una misma operación sobre numerosos datos al mismo tiempo en paralelo, mediante operaciones simples que son asignadas a múltiples hilos.
- Los tipos de variables y arreglos utilizados, y su tamaño, son congruentes con los patrones de memoria y las instrucciones aritméticas de la GPU, garantizando su buen desempeño.

Tomando en cuenta estas consideraciones, resulta sumamente importante determinar que partes del proceso de simulación que se desea implementar usando CUDA deben ser paralelizadas y cuáles no.

Proceso general para simular la evolución de secuencias

A modo general, el proceso de simulación de la evolución molecular de secuencias genéticas se realiza mediante los pasos algorítmicos descritos a continuación:

Paso 1: Especificar la cantidad de descendientes que pudiera tener cada ancestro.

Paso 2: Generar una secuencia mutante aplicando la técnica de Markov seleccionada, a partir de la secuencia ancestral.

Paso 3: Eliminar los codones de parada existentes en las secuencias mutantes obtenidas.

Paso 4: Clasificar la secuencia recién creada a través de la secuencia de aminoácidos sintetizada. Si coincide con uno de los alelos de la población, se aumentará el contador de este en 1; de lo contrario, se agregará la mutante a la población como una nueva clase. En ambos casos, la cantidad total de secuencias de la población se incrementa en 1.

Paso 5: Calcular la distribución de probabilidad de la población actual.

Paso 6: Tomar la secuencia mutante generada como el nuevo ancestro y volver el algoritmo al paso 2 si la cantidad de descendientes no se ha acabado; de lo contrario, moverse al paso siguiente.

Paso 7: Seleccionar el nuevo ancestro entre las secuencias de la población actual, si el número de ancestros no se ha terminado, el algoritmo vuelve al paso 1; de otro modo, ir al paso siguiente.

Paso 8: Terminar el algoritmo.

Al realizar un análisis de este proceso y de las consideraciones abordadas con anterioridad para el uso de CUDA, se identificaron como procedimientos a ser paralelizados la mutación (Paso 2), y

la clasificación de las secuencias (Paso 4). Acorde a esta valoración el esquema de paralelización propuesto para la simulación de la evolución de secuencias es el que se muestra en la *Figura 2*.

Paralelización de las mutaciones

Los sitios de una secuencia genética mutan de forma independiente, es decir, que las mutaciones que ocurran en uno de ellos no influyen en las que puedan ocurrir en otros sitios de la secuencia.

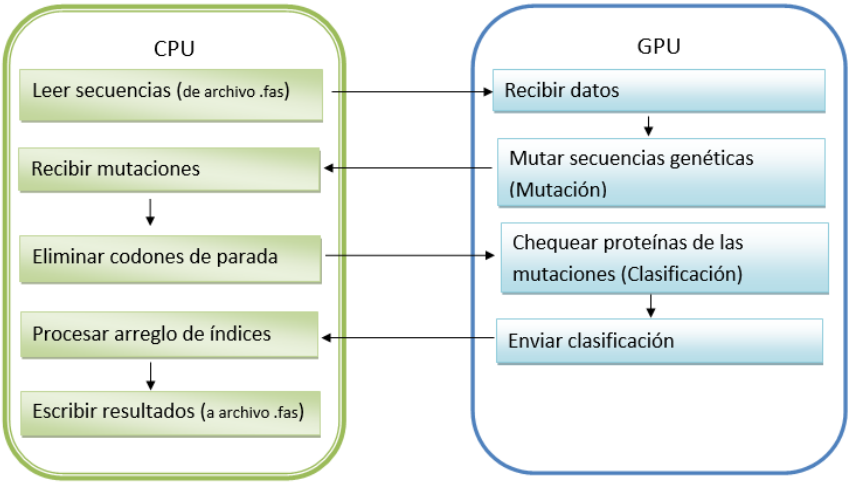

Figura 2. Esquema general de paralelización de la simulación.

Esta independencia facilita notablemente el proceso de paralelización; y dadas sus características se aplicó la técnica SIMD (Single Instructions Multiple Data), que permite conseguir paralelismo a nivel de datos y consiste en aplicar una misma operación sobre un conjunto de datos, en nuestro caso secuencias genómicas, y con una organización en donde una única unidad común de control despacha las instrucciones a diferentes unidades de procesamiento. Todas estas reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos.

Como el proceso de mutar un sitio es invariable y solo interesa la base nitrogenada del ancestro, entonces se programa dicho procedimiento como una operación y al aplicarlo a más de un sitio durante la ejecución se obtiene la paralelización. En la implementación en CUDA de este esquema se realiza un kernel para el proceso de mutar una generación de secuencias. Las secuencias se modelan como arreglos de tipo char, pues solo se cuenta con 4 bases. Esta variante posee un menor consumo en memoria, pero el procesamiento de un sitio necesita de más operaciones debido a la necesaria conversión de la base a una codificación numérica. En una implementación secuencial, si necesitamos obtener 10 descendientes el proceso debe de ejecutarse 10 veces, pues se obtiene solamente una mutación en cada iteración del algoritmo. Por su parte la implementación paralela del algoritmo logra esta tarea en un solo paso, generando de una vez todas las mutaciones deseadas para cada uno de los ancestros de una generación, en la *Figura 3* se muestra gráficamente el proceso de mutación paralelizado.

La sincronización de los hilos se logra utilizando _syncthreads(), una función propia de CUDA; esta sincronización resulta necesaria para el cálculo de los elementos del procedimiento evolutivo seleccionado (matriz Q* de tasas de sustitución, matriz P(t) de probabilidades de transición, o matriz

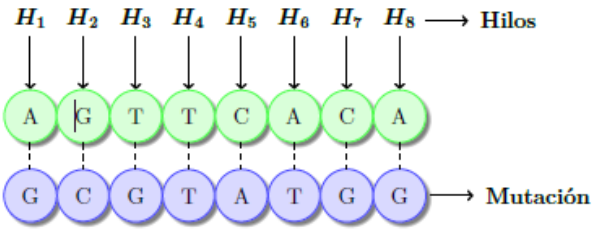

Figura 3. Proceso de mutación paralelizado por sitio.

de transición M) que en el esquema de paralelización propuesto se realiza en el hilo con índice 0 de cada bloque, y por tanto los demás hilos del bloque deben esperar a que este hilo coloque su resultado en memoria compartida para poder comenzar el proceso mutacional. Aunque esta forma de obtener los elementos del procedimiento seleccionado contiene divergencia entre los hilos de un bloque, la rapidez con la que se producen los accesos a la memoria compartida por parte de los hilos conduce a un menor consumo temporal.

Algoritmo paralelo para realizar las mutaciones en cada hilo

Entrada: arreglo de ancestros A, arreglo para almacenar las mutaciones M, número de ancestros nag, número de descendiente por cada ancestro ndv, número de sitios N, y otros elementos necesarios según procedimiento implementado (ej. frecuencias de aparición F, alpha1, alpha2, beta, etc.).

Salida: Mutaciones almacenadas en R.

```
1:  Reservar en memoria compartida arreglo P, Q, o M;
2:  tid ← id_hilo + id_Bloque * size_Bloque;
3:  while (tid < N) do
4:     for (i ← 0 hasta nag) do
5:        pos ←   i * nag + tid;
6:        c ← A[pos];
7:        old ← convert_base(c, 0);
8:        j ← 0;
9:        repeat
10:          if (id_hilo == 0) then
11:            computeMatrix(P, pos, …);  //calcular P, Q o M según caso
12:          end if
13:          Sincronizar_Hilos;
14:          u ← U(0, 1);
15:          v ← U(0, 1);
16:          // Mutar acorde al procedimiento elegido. En este ej. MH
17:          mutation(P, old, new, u, v, F);
18:          r ← convert_base(new, 1);
19:          R[ i*N*ndv + N*j + tid] ← r;
20:          old ← new;
21:          j++;
```

```
22:       until (j < ndv)
23:    end for
24:    tid ←  tid + size_Malla * size_Bloque;
25: end while
```

Paralelización de la clasificación de las mutaciones mediante sus proteínas

El proceso de clasificación, se llevaba a cabo con el objetivo de detectar cuando dos secuencias distintas sintetizan una misma proteína. De forma secuencial se hace comparando cada secuencia obtenida con cada una de las secuencias almacenadas, tomando en cuenta tanto las mutaciones generadas como las secuencias iniciales. En la versión paralela del algoritmo se han utilizado algunas estrategias en función de lograr mayor nivel de paralelización durante la ejecución paralela del algoritmo siguiente:

Algoritmo paralelo para clasificar las mutaciones

Entrada: aminoácidos de las mutaciones *R*, aminoácidos de los ancestros *A*, índices *I*, num_sitios, nag, ndv

Salida: Arreglo de índices (*I*) con valores de las clasificaciones.

```
1: codons ← num_sites/3;
2: tot_seqs ←   nag * ndv;
3: tid ← id_hilo + id_Bloque * size_Bloque;
4: while (tid < tot_seqs) do
5:     if (I[tid] == -100) then
6:       flag ← false; i ← 0;
7:       while (!flag && i < nag) do
8:            for (j ← 0; j < codons; j ← j + 1) do
9:               if (A[i * codons + j] != R[tid * codons + j]) then
10:                 break;
11:              end if
12:           end for
13:           if (j == codons) then
14               flag ← true;
15:              if (I[tid] == -100) then
16:                 I[tid] ← i;
17:              end if
18:           end if
19:           i ← i + 1;
20:     end while
21:     if (!flag && I[tid] = -100) then
22:       i ← tid + 1;
23:       while (i < tot_seqs) do
27:            for (j ← 0; j<codons; j ← j + 1) do
28:               if (R[tid * codons + j] != R[i * codons + j]) then
29:                  break;
30:              end if
32:           end for
33:           if (j == codons && I[i] == -100) then
34:             I[i] ← nag + tid;
35:           end if
36:           i ← i + 1;
37:       end while
38:     end if
```

```
39:   end if
40:   tid ← tid + size_Malla * size_Bloque;
41: end while
```

Para lograr una adecuada paralelización de este procedimiento de clasificación se ha utilizado un arreglo de índices (*I*), el cual tendrá una posición por cada una de las secuencias obtenidas en una generación, por lo cual su longitud será igual a (`nag * ndv`), estará inicializado con −100 en todas sus posiciones y se almacenará en la memoria global. En los índices de este arreglo se tendrá −100 en caso de que la correspondiente secuencia no sintetice una proteína obtenida anteriormente, un valor del intervalo [0, *nag* − 1] si la misma produce una proteína igual que algunas de las correspondientes a los ancestros o el mismo estará en el intervalo [`nag, nag + nag * ndv` −1] si la proteína se corresponde con algunas de las mutaciones que se generaron junto con ella en la actual generación.

En el algoritmo paralelo se asigna un hilo a cada una de las secuencias, diferente a como se paralelizaron las mutaciones, en las cuales un hilo era responsable de la mutación de un sitio dentro de la secuencia. Luego que el hilo conoce que mutación le corresponde entonces realiza la comparación entre su respectiva secuencia y todos los ancestros, si la misma no es clasificada por ningún ancestro se procede a compararla con cada una de las mutaciones que se obtuvieron de forma posterior a ella, buscando las que ella clasifica. En el proceso se actualiza la información contenida en el arreglo, como este se encuentra en la memoria compartida y es escrito y leído por los hilos de forma simultánea, entonces constituye una sección crítica y se hace uso de un semáforo para realizar tanto las lecturas como las escrituras de forma atómica.

3. Resultados y Discusión

Para evaluar el desempeño del modelo de paralelización propuesto, se decidió implementar el procedimiento de simulación Metrópolis Hastings sin repetición en dos aplicaciones: una secuencial y una paralela, utilizando la tecnología CUDA. Estas aplicaciones fueron desarrolladas empleando el lenguaje de programación C/C++ y se utilizó el Netbeans 7.4 como IDE de desarrollo, sobre la versión 7.3 del sistema operativo Debian.

Los experimentos se realizaron sobre dos computadoras con 3 GB de memoria RAM y un microprocesador Intel Core 2 Quad modelo Q8200, cada una de ellas cuenta con una tarjeta de video NVIDIA, cuyas especificaciones se muestran en la *Tabla 1*. Como se puede observar, la primera de las dos tarjetas posee una mayor capacidad de cómputo puesto que contiene el cuádruple de núcleos CUDA que la segunda.

Tabla 1. Especificaciones de las tarjetas gráficas.

Procesador	Arquitectura	Memoria	Núcleos
GeForce GT 630	GK107	2048 MB	192

GeForce GT 610	GF119	1024 MB	48

Las bases de secuencias genómicas fueron obtenidas del sitio NCBI (National Center for Biotechnology Information) que se encuentra accesible en la URL: http://www.ncbi.nlm.nih.gov/genomes/FLU/Database/nph-select.cgi?go=database, donde están disponibles de forma totalmente libre y gratuita. Los experimentos se realizaron sobre dos bases de casos (GenBank datasets). La primera de ellas contiene un alineamiento de secuencias de la variante H3N2 (archivo FASTA_H3N2.fas), conformada por un conjunto de 198 secuencias cada una con 1765 sitios, con la presencia de gaps en 73 de ellos. El segundo alineamiento se compone de 2451 secuencias de la variante H1N1 (archivo FASTA_USA.fas), con 1803 sitios dentro de los cuales podemos encontrar 830 gaps. Sobre estos gaps se realizó un proceso alineamiento múltiple empleando la herramienta Clustal Omega (clustalo), que puede descargarse desde http://www.clustal.org/omega/.

Se planificaron varias pruebas con el objetivo de verificar las ventajas que nos proporciona CUDA en cuanto al tiempo de ejecución. Los parámetros de las pruebas se tomaron de forma tal que el número de ancestros por generación (nag) varía su valor en el intervalo entero [2, 5], la cantidad de descendientes por ancestro (ndv) debe tomar uno de los valores {1000, 1500, 2000, 2500, 3000} y el número de generaciones (ngen) se establece en dependencia de la base de secuencias que se estén procesando (en los experimentos realizados se toma el valor 1 y 2). El resultado de las pruebas son mutaciones de las secuencias genéticas obtenidas de la simulación de la evolución y almacenadas en un archivo con formato FASTA (.fas).

En la *Tabla 2* se muestran los resultados obtenidos de simular la evolución molecular de los virus H3N2 y H1N1, donde la columna Mutaciones es el total de mutaciones obtenidas en cada experimento, y la columna Tiempo es el tiempo de ejecución dado en segundos para cada experimento; con los valores de los parámetros de simulación que se indican en la *Tabla 2* para las variables: número de generaciones virales generadas en cada simulación (ngen), número de ancestros para obtener una generación (nag), y el número de descendientes por cada ancestro (ndv).

Tabla 2. Resultados de la simulación

	Experimento	**H3N2 mutaciones obtenidas con nag = 5, ndv = 1000**		**H1N1 mutaciones obtenidas con nag = 4, ndv = 2000**	
		Mutaciones	**Tiempo**	**Mutaciones**	**Tiempo**
(ngen=1)	**Secuencial**	3751	2949.65	6002	3682,25
	GT 630	4750	4.29	8000	4,95
	GT 610	4750	10.07	8000	11,25
(ngen=2)	**Secuencial**	5327	4943.70	12003	14445,17
	GT 630	9500	13.4	14048	14,64
	GT 610	8581	29.92	8539	34,61

En la tabla anterior es apreciable la notoria diferencia de tiempos de ejecución que existe entre la implementación secuencial y la de nuestro modelo de paralelización, evidencia de la reducción de tiempo del modelo propuesto. En la representación gráfica (*Figura 4*) de estos resultados también se puede apreciar que con el aumento del número de secuencias a procesar se hace aún mayor la diferencia entre ambas implementaciones, mostrando un mayor factor de ganancia de la versión paralela.

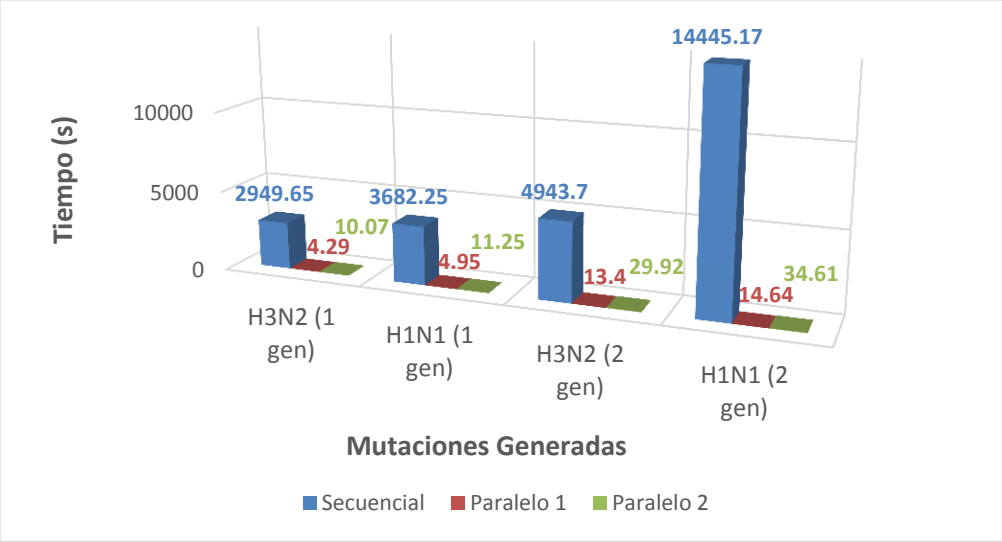

Figura 4. Mutaciones generadas respecto al tiempo

4. Conclusiones y Recomendaciones

Se propone un modelo de paralelización en CUDA de la simulación de la evolución de secuencias genéticas que combina la estructura general de una aplicación CUDA con el diseño paralelo de los dos procedimientos más complejos identificados dentro de la simulación. En la paralelización del procedimiento de mutación se utiliza la estrategia de asignarle a cada hilo una base nitrogenada de un aminoácido para calcularle sus posibles mutaciones y en el caso del procedimiento de clasificación se le asigna a cada hilo todo el aminoácido para ser clasificado.

Los resultados experimentales muestran que el modelo paralelo propuesto al ser implementado ofrece un mejor desempeño que la implementación secuencial. Con el aumento del número de secuencias a procesar aumenta significativamente el tiempo de ejecución de la versión secuencial, sin ser significativo este aumento de tiempo en la versión paralela.

Bibliografía

Andrieu, C., & Doucet, A. (2010). Particle Markov chain Monte Carlo methods - Andrieu - 2010 - Journal of the Royal Statistical Society: Series B (Statistical Methodology) - Wiley Online Library. *Journal of the Royal*. Retrieved from

http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2009.00736.x/full%5Cnpapers2://publication/uuid/9E1D438C-88FE-4D49-8CFE-F4D4525C0C7D

Claver, J. M., Sanjuan, A., & Arnau, V. (2007). *Análisis paralelo de secuencias de ADN mediante el uso de GPU y CUDA.* Uv.es, 1–6. Retrieved from http://www.uv.es/VARNAU/115-GPU-ANACAP_2008.pdf

Cornebise, J., & Peters, G. W. (2009). Comments on “Particle Markov Chain Monte Carlo” by C. Andrieu, A. Doucet and R. Hollenstein. *Arxiv*, 1–9. Retrieved from http://arxiv.org/abs/0911.3866

Counsell, D. (2005). Bioinformatics and molecular evolution. *Comparative and Functional Genomics*, 6(5-6), 317–319. https://doi.org/10.1002/cfg.486

CUDA Zone | NVIDIA Developer. (2011). Retrieved January 4, 2017, from https://developer.nvidia.com/cuda-zone

Hwu, W. (2012). GPU computing gems. Applications of GPU computing series. https://doi.org/10.1017/CBO9781107415324.004

Link, W. A., & Eaton, M. J. (2012). On thinning of chains in MCMC. *Methods in Ecology and Evolutio*n, 3(1), 112–115. https://doi.org/10.1111/j.2041-210X.2011.00131.x

Liu, Y., Schmidt, B., & Maskell, D. L. (2012). CUSBHAW: A CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14), 1830–1837. https://doi.org/10.1093/bioinformatics/bts276

Liu, Y., Wirawan, A., & Schmidt, B. (2013). CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14, 117. https://doi.org/10.1186/1471-2105-14-117

Manavski, S. A., & Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9 Suppl 2, S10. https://doi.org/10.1186/1471-2105-9-S2-S10

NVIDIA. (2015). CUDA Toolkit 7.5 Documentation. Retrieved January 4, 2017, from http://docs.nvidia.com/cuda/index.html

Nvidia, C. (2011). NVIDIA CUDA C Programming Guide. *Changes*, (350), 173. https://doi.org/PG-02829-001_v6.0

Sánchez, G. A. L., Carbajal, M. O., Cortés, N. C., & Fernández, R. B. (2012). Sobre la programación paralela de un algoritmo de optimización por cúmulo de partículas en un

dispositivo GPU multi-hilos. *Intekhnia*, 6(2), 59–74.

Schatz, M., Trapnell, C., Delcher, A., & Varshney, A. (2007). High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8, 474. https://doi.org/10.1186/1471-2105-8-474

Weber, R. (2012). *Markov Chains*. Statslab.Cam.Ac.Uk, 28–49. https://doi.org/10.1017/CCOL0521534283.010

Yang, Z. (2006). Computational molecular evolution. *Oxford Series in Ecology and Evolution*, xvi, 357 p. https://doi.org/10.1093/acprof:oso/9780198567028.001.0001

Yang, Z., & Rodríguez, C. E. (2013). Searching for efficient Markov chain Monte Carlo proposal kernels. *Proceedings of the National Academy of Sciences of the United States of America*, 110(48), 19307–12. https://doi.org/10.1073/pnas.1311790110