Universidad Central “Marta Abreu” de Las Villas

Facultad de Matemática, Física y Computación

Trabajo para optar por Título de

Licenciado en Ciencia de la Computación

Programación paralela usando CUDA: aplicación en la Bioinformática

Autor:

Abel Cepero Alejo

Tutores:

MSc. Leonardo F. del Toro Melgarejo

Dr. Daniel Gálvez Lio

2011-2012

Declaración Jurada

El que suscribe, Abel Cepero Alejo, hago constar que el trabajo titulado “Programación paralela usando CUDA: aplicación en la Bioinformática” fue realizado en la Universidad Central “Marta Abreu” de Las Villas como parte de la culminación de los estudios de la especialidad de Licenciatura en Ciencia de la Computación, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

_____

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del tutor Firma del jefe del Laboratorio

Fecha: 24/septiembre/2012

Frase

Dedicatoria

A papi.

A Berkis Pino Martín.

Agradecimientos

A **Daniel Gálvez Lio** por hacerse cargo de mí.

A Leonardo F. del Toro por su ayuda.

Resumen

La tecnología CUDA de NVIDIA se ha extendido a nivel mundial como un nuevo paradigma de programación paralela. En este paradigma la memoria juega un papel fundamental, pues los cinco tipos de memoria que existen en CUDA tienen características diferentes y en muchos casos definen su usabilidad. Las aplicaciones desarrolladas usando la tecnología CUDA se extienden a varios campos, entre ellos el de la Bioinformática.

En este trabajo se explora el modelo de memoria de CUDA y se han valorado varias aplicaciones de CUDA en el campo de la Bioinformática, donde los autores de dichas aplicaciones explotan los diferentes espacios de memoria y ejecutan los *kernels* en el dispositivo CUDA con múltiples configuraciones de hilo, bloque y malla, estudiando su comportamiento.

Además, se presentan los elementos teóricos necesarios para el cálculo de la matriz de distancias como ejemplo de una tarea a realizar dentro del campo de la bioinformática. Este problema fue seleccionado con el objetivo de experimentar su implementación con la tecnología CUDA y obtener una primera solución computacional, la cual se estudia usando la herramienta *Visual Profiler* para valorar su efectividad.

Abstract

NVIDIA CUDA technology has spread worldwide as a new parallel programming paradigm. In this paradigm memory plays a key role, due to the five types of memory that exist in CUDA have different characteristics and often define their different usability. Applications, developed using CUDA technology, involve several knowledge fields, including the Bioinformatics.

In this work we explore the memory model of CUDA applications and also revised several CUDA applications in Bioinformatics field, where the authors of these applications exploit different memory spaces and the kernels running on the device with multiple CUDA configurations for threads, blocks and grids, studying their behavior.

In addition, the theoretical elements necessary for understand the calculation of the distance matrix as an example of a task to perform on the field of bioinformatics was present. This problem was selected in order to proof its implementation with CUDA technology and get a first computing solution, which is studied using the Visual Profiler tool to assess their effectiveness.

Tabla de contenido

Índice de figuras

Introducción

La uso de las Unidades de Procesamiento Gráfico (GPU) se ha popularizado y masificado en los últimos tiempos, en principio por el auge de juegos en las computadoras y la utilización en estos de cada vez más recursos gráficos que requieren de una alta capacidad de procesamiento. Pero para NVIDIA el desarrollo de sus tarjetas graficas ha ido más allá y ha facilitado acceso a los programadores a los recursos de dichas tarjetas a través de una interfaz.

Los programadores utilizan un modelo de programación de la GPU en un lenguaje de alto nivel y una arquitectura de programación paralela, estos dos elementos conforman lo que NVIDIA denomina *Compute Unified Device Architecture* (CUDA).

Un dispositivo CUDA en una GPU típicamente contiene miles de núcleos. Los núcleos se agrupan en un flujo de multiprocesadores (*Streaming Multiprocessors - SM*) y cada núcleo puede ejecutar miles de hilos. Trabajando en una arquitectura conocida *Single Instruction Multiple Threads* (SIMT), donde todos los núcleos ejecutan la misma instrucción al mismo tiempo. Estas características de hardware y las implementaciones por software hacen que una aplicación aproveche los recursos de cálculo de estas tarjetas.

La programación en CUDA deviene como un nuevo paradigma de programación paralela, pues aunque mantiene los principios generales, incluye características especiales como tipos nuevos de memoria y un modelo de ejecución de hilos diferente.

El auge que ha ido alcanzando esta tecnología viene dado por su accesibilidad y por la mejora en el rendimiento de las aplicaciones desarrolladas con ella. “Las aplicaciones desarrolladas para ejecutarse en las GPU de NVIDIA tienen mejor rendimiento por dólar y por *watt* que otras versiones desarrolladas solo para la CPU” (J. Sanders and E. Kandrot 2011). En la actualidad ya se encuentran en la literatura referencias a aplicaciones para procesamiento de imágenes médicas, en

sistemas para las ciencias medioambientales y en muchas ramas donde es necesaria la computación de alto rendimiento.

El **objetivo general** de este trabajo:

Proporcionar un marco referencial sobre las posibilidades de CUDA y su aplicación en el área de la Bioinformática.

Los **objetivos específicos**:

- Presentar elementos relacionados con los modelos de memoria utilizados en la programación paralela, haciendo énfasis en CUDA.
- Caracterizar algunas herramientas que ofrece CUDA en el análisis del rendimiento de aplicaciones hechas con esa tecnología.
- Analizar desarrollos en CUDA de aplicaciones en Bioinformática.
- Implementar un ejemplo sencillo del uso de CUDA en el área de la Bioinformática.

Esta tesis escrita consta de tres capítulos, conclusiones, recomendaciones, referencias bibliográficas y anexos. En el primer capítulo se analiza uno de los elementos fundamentales en la computación paralela: los modelos de memoria, en particular el modelo utilizado en CUDA. En el capítulo dos se describe la implementación de aplicaciones para Bioinformática basadas en la GPU. En el capítulo tres se abordan los elementos teóricos del problema específico de Bioinformática, cálculo de la matriz de distancias y se describe y analiza una implementación en CUDA para dar solución a este problema.

Capítulo 1: Elementos de Programación usando CUDA: modelos de la memoria en la programación paralela.

La computación paralela ha tenido un tremendo impacto en gran variedad de áreas que van desde aplicaciones para la simulación científica e ingenieriles a aplicaciones comerciales de minería de datos. En este capítulo se abordan elementos básicos de los modelos de memoria en la programación paralela, teniendo en cuenta los dos modelos trabajados en el grupo del programación paralela del laboratorio de Programación e Ingeniería de Software. Estos modelos se corresponden con la memoria distribuida al programar en MPI y la memoria compartida al programar en OpenMP, para luego dar paso al estudio de la memoria en CUDA y sus diferentes tipos. También se abordan otros elementos de programación y herramientas de software disponibles en CUDA que amplían el trabajo desarrollado en (Brito 2012).

1.1 Modelos de memoria en la programación paralela

En programación paralela uno de los elementos fundamentales a analizar es el modelo de memoria usado, en ocasiones esa característica define el paradigma de programación paralela a utilizar.

1.1.1 Memoria distribuida en MPI

En este paradigma de programación paralela una visión lógica de una máquina que lo soporta consiste de *p* procesadores cada uno con un espacio exclusivo de memoria. En cada procesador *p* se ejecuta el mismo programa de manera asincrónica (puede ejecutarse programas diferentes en diferentes procesadores) y este utiliza la memoria RAM asociada al procesador donde se ejecuta. Este paradigma es característico de los clústeres de computadoras donde un grupo de computadoras están conectadas entre sí por una red y colaboran en la solución de un problema ejecutando en cada procesador el programa paralelo correspondiente. Ver Figura 1.

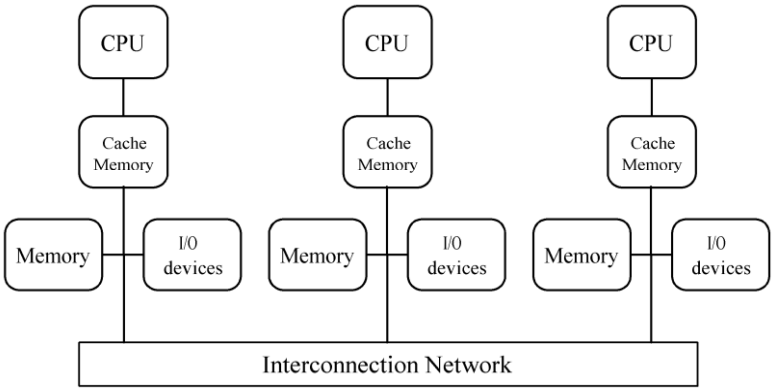


Figura 1 Arquitectura genérica de memoria distribuida en una computadora de múltiple CPU, tomada de (Quinn 2004)

Como cada procesador *p* ejecuta en programa que le corresponde en su memoria las variables que utiliza el programa son locales a él, es decir, la memoria total de ejecución está distribuida entre los p procesadores que ejecutan el programa. Pero en ocasiones es necesario que ciertos valores calculados en un procesador se conocido en otro procesador, entonces debemos "enviar" ese valor de un procesador a otro y en el otro procesador recibir ese valor. Este proceso de interacción entre procesos o nodos a través de mensajes se conoce como el nombre de "paso de mensajes". Es muy probable que en la literatura sobre este tema Ud. pueda encontrar más detalles sobre las plataformas de paso de mensajes o el paradigma de paso de mensajes.

Según (Ananth Grama 2003) hay dos atributos fundamentales que caracterizan el paradigma de la programación con paso de mensajes. El primero es que este paradigma asume un espacio de memoria particionado (memoria distribuida) y el segundo es que solo soporta paralelización explicita.

La primera característica presupone que cada dato pertenece a una de las particiones del espacio de memoria, por lo que los datos deben ser explícitamente particionados y colocados en su lugar. Esto adiciona complejidad a la programación. Por otro lado todas las interacciones requieren cooperación de dos procesos: el proceso que tiene el dato y el que desea acceder al dato. Este requerimiento de cooperación agrega trabajar con una gran complejidad adicional

en la programación que resulta en ocasiones un cuello de botella para el rendimiento del programa paralelo.

La segunda característica requiere que el paralelismo sea codificado explícitamente por el programador. Es decir, el programador es responsable de analizar el algoritmo secuencial e identificar las vías por las cuales descomponer los cálculos y extraer el paralelismo. Como resultado la programación usando este paradigma es difícil y demanda inteligencia, por otro lado, programas bien escritos en este paradigma pueden alcanzar altos rendimientos y ser escalables a un gran número de procesadores.

Existen varias metodologías para diseñar programas paralelos como la descrita en (Foster 2003) que consta de 4 etapas: particionamiento, comunicación, aglomeración y correspondencia (*map*).

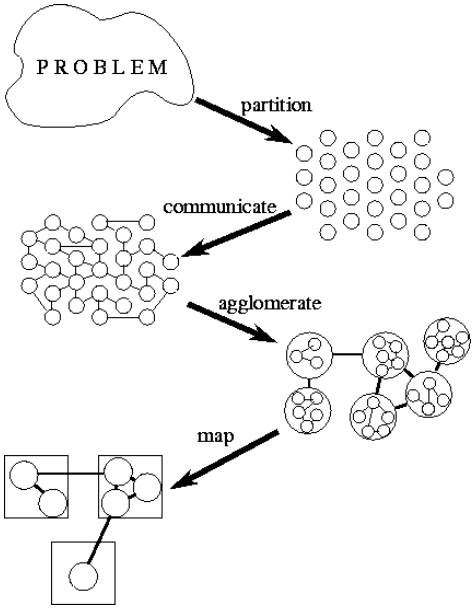


Figura 2 Representación de la Metodología PCAM tomada de (Foster 2003)

La última etapa hace corresponder las tareas que se pueden ejecutar en paralelo con los procesadores donde se ejecutaran y el programador debe en su programa distribuir los datos de entrada, de salida e intermedios y establecer la comunicación necesaria entre los procesadores para intercambiar información.

1.1.2 Memoria compartida en OpenMP

OpenMP es una API que soporta programación multiproceso de memoria compartida en múltiples plataformas (lenguajes C, C++ y Fortran), en la mayoría de las arquitecturas de procesadores (Intel, AMD) y en diferentes sistemas operativos, como son, Solaris, AIX, HP-UX, GNU/Linux, Mac OS X y Windows. Su definición provee directivas de compilador, biblioteca de funciones y variables de entorno que regulan el comportamiento de los programas, lo que forma en conjunto el núcleo de OpenMP (Wikipedia.org 2012).

De acuerdo a su modelo de trabajo pertenece al paradigma de programación implícita, permitiendo programar aplicaciones paralelas multi-hilos (multi-threaded). Su empleo resulta simple, ya que desde el punto de vista de la programación se usa un lenguaje de programación secuencial y el compilador, de acuerdo a las directivas definidas en esta API, inserta las instrucciones necesarias para ejecutar el programa en un computador paralelo. Por ello, la tarea del programador no es compleja ya que le corresponde al compilador analizar y comprender las dependencias para asegurar un mapeo eficiente en el computador paralelo.

Un programa desarrollado en OpenMP emplea el modelo de programación fork-join, con generación de múltiples hilos (threads). Esto implica que se ejecuta un hilo maestro hasta que aparece el primer constructor paralelo. A partir de este punto se crean hilos esclavos, los que son controlados por el hilo maestro. Al final del constructor se sincronizan los hilos y continúa la ejecución del maestro ver Figura 3.

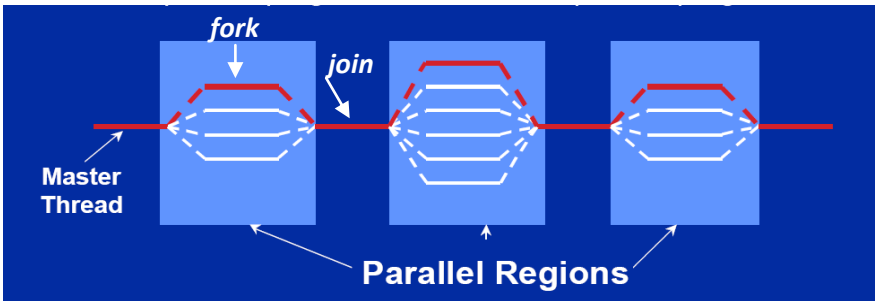


Figura 3 Proceso de ejecución de los hilos en la biblioteca OpenMP.

En los lenguajes C y C++ toda construcción de OpenMP emplea directivas de compilación o pragmas (#pragma omp construct [clause [clause]…]) para identificar las secciones que se ejecutarán en paralelo, como en la creación de hilos y la distribución de trabajos para uno o todos los hilos.

En cualquiera de los casos #pragma puede especificar el alcance de las variables con las que se relaciona por medio de las claúsulas siguientes:

- **private**(lista) privadas a los hilos, no se inicializan antes de entrar y no se guarda su valor al salir de la región paralela.
- **firstprivate**(lista) privadas a los hilos, se inicializan al entrar con el valor que tuviera la variable correspondiente.
- **lastprivate**(lista) privadas a los hilos, al salir quedan con el valor de la última iteración o sección.
- **shared**(lista) compartidas por todos los hilos.
- **default**(shared|none) indica cómo serán las variables por defecto.
- **copyin**(lista) para asignar el valor de una o varias variables del hilo master a sus respectivas variables locales en los hilos.

Entre los diferentes constructores encontramos:

- **parallel**: Especifica que región dentro del código se ejecutará en paralelo por los hilos (el numero de hilos puede especificarse a través de la función ***omp_set_num_threads( )*** o por medio de la variable de entorno ***OMP_NUM_THREADS***). Las cláusulas (private, firstprivate, default, shared, copyin y reduction) se emplean para indicar la forma en que se accede a las variables.
- **for**: Ejecuta un bucle en paralelo dividiendo las iteraciones entre los hilos existentes. El manejo de las variables empleadas en el bucle depende de las clausulas definidas. La definición de la clausula ***schedule*** permite establecer previamente como se dividen las iteraciones del ***for*** entre los hilos:
  - ✓ ***schedule***(static,tamaño) las iteraciones se dividen según el tamaño, y la asignación se hace estáticamente a los hilos. Si no se indica el tamaño se divide por igual entre los hilos.

- ✓ ***schedule***(dynamic,tamaño) las iteraciones se dividen según el tamaño y se asignan a los hilos dinámicamente cuando van acabando su trabajo.
- ✓ ***schedule***(guided,tamaño) las iteraciones se asignan dinámicamente a los hilos pero con tamaños decrecientes.
- ✓ ***schedule***(runtime) deja la decisión para el tiempo de ejecución, y se obtienen de la variable de entorno OMP_SCHEDULE.

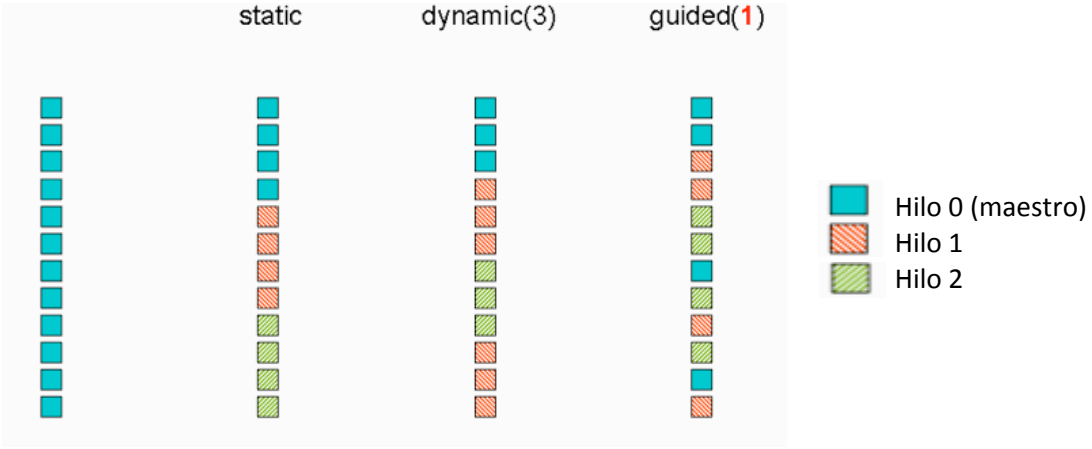


Figura 4 Formas de dividir las iteraciones de una directiva for entre 3 hilos.

- **sections**: Define diferentes secciones que se ejecutaran de manera independiente dentro de una región paralela.

Estos constructores se pueden combinar si solo se ejecutara un solo bucle ***for*** o una sola ***sección*** dentro de una región paralela.

a) #pragma omp parallel for [cláusulas]
   bucle for
b) #pragma omp parallel sections [cláusulas]

Aunque no se ejecuten en paralelo es posible definir un grupo de operaciones secuenciales asignadas a cualquier hilo o específicamente al hilo maestro (master). Para ello se emplean los constructores ***single*** y ***master*** respectivamente.

a) #pragma omp single [cláusulas]
b) #pragma omp master

Otras operaciones importantes permiten el acceso a variables compartidas de manera exclusiva o la sincronización de todos los hilos involucrados en una región paralela. Para ello se puede emplear los constructores ***critical*** y ***barrier*** respectivamente.

Una clausula especial (***reduction***) se puede combinar con el constructor ***for*** para ejecutar operaciones de reducción. La cláusula es ***reduction*** (<op>:<lista>), permite uno de los siguientes operadores (op) para C/C++:

+, -, *, &, ^, |, &&, ||

donde: <lista> es una lista de variables.

Esto permite combinar el valor de variables privadas calculadas en cada hilo.

1.1.2.1 Ejemplo de memoria compartida

El siguiente ejemplo muestra el uso de memoria compartida a través de la cláusula ***reduction*** unida a los constructores ***parallel*** y ***for***. En este se define el uso de 3 hilos por medio de la función ***omp_set_num_threads( )***.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char **argv){
   int i, tid, z=0, n=3;
   int x[3], y[3];

   omp_set_num_threads(3);
   #pragma omp parallel for reduction(+:z)
      for (i = 0; i < n; i++) {
          x[i]= i + 1;
          y[i] = x[i] * x[i];
          z = z + x[i] * y[i];
      }
   printf("Valor de z=%d\n", z);
   return 0;
}
```

Figura 5 Ejemplo de OpenMP

Cada hilo tiene una copia local las variables, pero los valores de cada variable local son “reducidos”, con el operador indicado, a una variable global (z). Observe

en la Figura 6, como OpenMP distribuye los valores de las variables entre cada hilo.

Cada iteración del bucle es ejecutada por un hilo diferente, ya que esa es la función del constructor #pragma… for, entonces la variable z (z' en la Figura 6) recibe valores diferentes para cada hilo que actualiza su valor. Observe como cada hilo maneja los valores de las variables x e y.

Finalmente todos los valores locales de la variable z (1, 8, 27) son reducidos según la operación suma y la variable z (36) tendrá el valor de sumar todos los valores parciales.

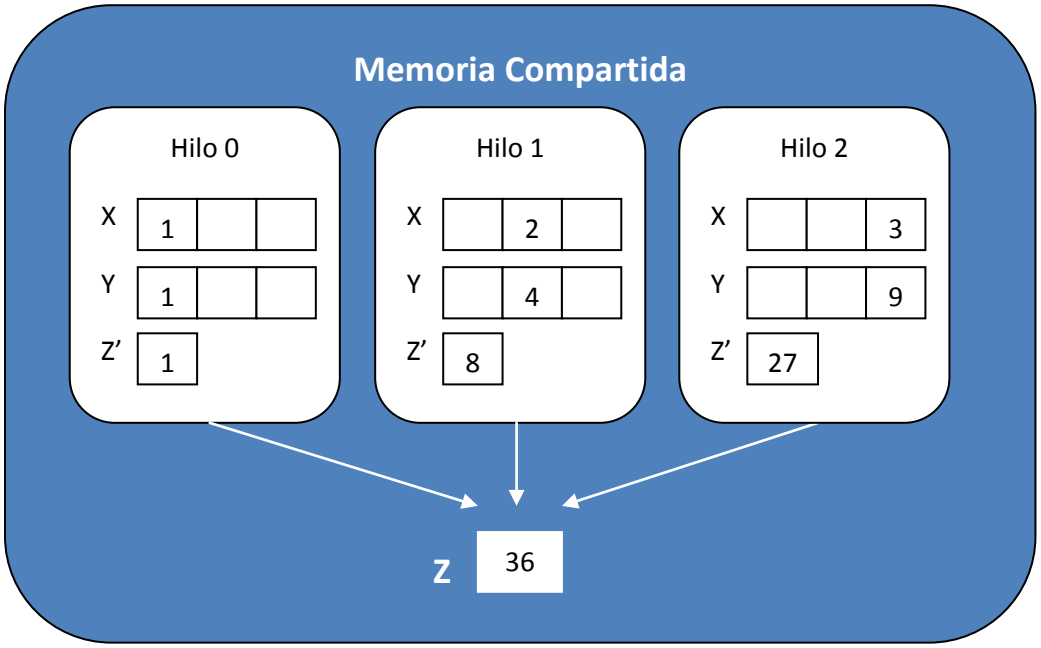


Figura 6 Distribución de los datos entre cada hilo y reducción del valor de la variable z.

1.2 La memoria en CUDA

CUDA como paradigma de programación paralela presenta características únicas en su modelo de memoria. Los dispositivos CUDA tienen varios espacios de memoria de acuerdo a su usabilidad (NVIDIA_Corporation_a 2011). Entre los distintos espacios de memoria se encuentran: global, local, compartida, textura y registros, tal como se muestra en la Figura 7.

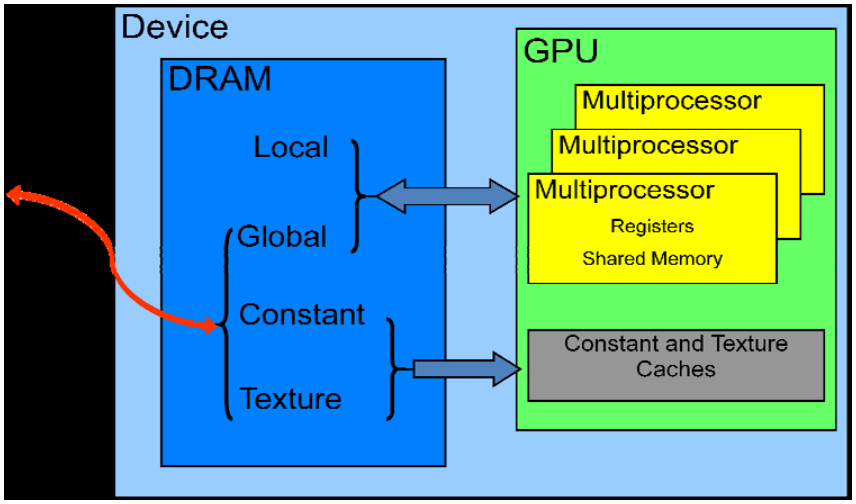


Figura 7 Espacios de Memoria. Tomado de (NVIDIA_Corporation_b 2011)

1.2.1 Memoria Global

El espacio de memoria global o memoria del dispositivo reside fuera del chip del núcleo de la tarjeta NVIDIA, lo que significa que está en la DRAM del dispositivo y su acceso es lento (NVIDIA_Corporation_a 2011). Es accedida a través de palabras alineadas de 32-, 64-, 128 byte. Una palabra está alineada si empieza con una dirección que es múltiplo de su tamaño, o sea, se puede acceder solo a la memoria que empieza en un múltiplo de 32. Cuando los hilos de un *warp* acceden a la memoria global, ya sea para escritura o lectura, el *warp* trata de fusionar los accesos a memoria, es decir reducir los accesos tanto como sea posible.

Este proceso consiste en fusionar los accesos a memoria de un *warp* en al menos una instrucción de 64-*byte* o 128-*byte*, si se cumplen ciertas condiciones en el acceso a memoria.

Si miramos la memoria global como un segmento de direcciones adyacentes, una condición típica, es cuando los accesos de los hilos de un *warp* estén en la misma línea de hasta 128-*byte*. La Figura 8 es un ejemplo.

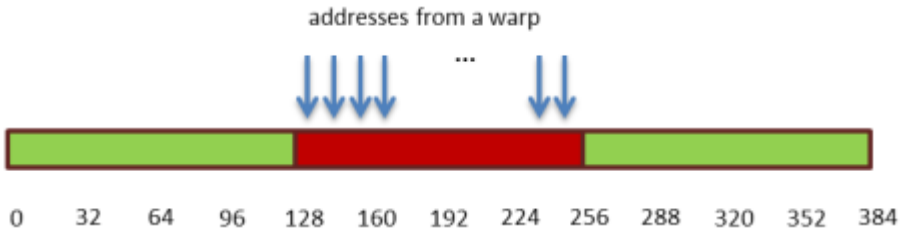


Figura 8 Fusionado de acceso continuo en un warp (NVIDIA_Corporation_b 2011).

Para una capacidad de cálculo 1.2 o superior el fusionado de los accesos a memoria es aplicable, algunas diferencias existen con capacidades de cálculo inferiores, como, los accesos a memoria se agrupan por la mitad de un *warp* y además el máximo número de *bytes* por palabras es 64.

Para capacidades de cálculo inferiores a 1.2 si algún hilo no participa en el acceso entonces se realizan tantas transacciones como accesos tenga la mitad del *warp*, los accesos tiene que ser contiguos para esta capacidad de cálculo.

Si la capacidad de cálculo es superior a 1.2 entonces aunque algún hilo no participe en el acceso, la memoria es satisfactoriamente fusionada. El caso de acceso no alineados se explica a como sigue.

Si los accesos se realizan dentro de un segmento alineado de 128-bytes entonces se realiza una transacción; si los accesos están a través de dos segmentos de 128-bytes se realizan dos transacciones: una de 64-bytes, para cubrir una parte del primer segmento y otra transacción de 32-bytes, para cubrir la parte del otro segmento de 128-byte. La figura 3.4 muestra estos dos casos.

En la Figura 9 los espacios de color azul representan dos secuencias de 64-*bytes* cada una alineada y los elementos de color naranja representan una secuencia alineada de 128-*bytes*. Los rectángulos rojos significan las transacciones a realizar.

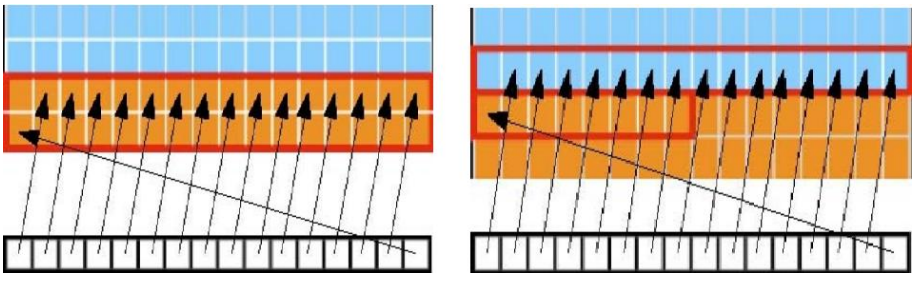


Figura 9 Dos casos de fusionado de memoria (NVIDIA_Corporation_b 2011).

Cuando se reserva memoria a través de la función `cudaMalloc ()` se garantizan 256-bytes de memoria alineada. También se pueden definir estructuras bajo los calificadores `aling (8)` y `aling (16)` para asegurar que se guarden en memoria 8 y 16 bytes alineados respectivamente.

Un elemento significativo en el proceso de optimización de aplicaciones desarrolladas con CUDA es el fusionado de memoria global.

1.2.1 Memoria Compartida

La memoria compartida se encuentra físicamente en el chip del núcleo de la tarjeta gráfica, lo que significa que el acceso a este tipo de memoria es rápido. Los parámetros o argumentos pasados a una función *kernel* son guardados en la memoria compartida, pero solo en el tiempo de la ejecución, por eso si la lista de parámetros es muy grande es usual guardar parte de esos argumentos en la memoria constante y luego referenciar esos valores (NVIDIA_Corporation_a 2011).

La memoria es compartida para todas los hilos dentro de un bloque de hilos. Una variable se guarda en la memoria compartida solo si se especifica con el modificador `__shared__` (NVIDIA_Corporation_a 2011). Dado que el acceso a esta memoria es rápido, es recomendado explotar este tipo de memoria. Es aconsejable que: cuando múltiples hilos en un bloque acceden al mismo dato en memoria global, se copie ese dato en la memoria compartida de esta forma se lee el dato una sola vez desde la memoria global y los hilos siguen su ejecución normal pero ahora acceden desde la memoria compartida.

1.2.2 Memoria Local

La memoria local recibe su nombre por el alcance de las variables dentro de un hilo, pero no por su estado físico. A diferencia de la memoria compartida, la memoria local se encuentra fuera del chip del núcleo de la tarjeta de video. El acceso a la memoria local es tan lento como el acceso a la memoria global. En la memoria local solo se guardan variable automáticas; es trabajo del compilador “nvcc” el ocupar espacio local para ciertas variables, solo en caso que el registro

no pueda mantener dicha variable o que el compilador entienda que un arreglo muy grande puede ser indexado dinámicamente (NVIDIA_Corporation_b 2011).

1.2.3 Memoria de textura

La memoria de textura esta fuera del chip de la tarjeta gráfica, pero es almacenada en cache. Este tipo de memoria es de solo lectura. La memoria global se lee a través de la memoria de texturas usando funciones de CUDA. O sea, usando las funciones se especifica qué tipo de dato se va a guardar en cache y que cantidad. Una vez los datos estén cache, el acceso a ellos se realiza a través de otras funciones que optimizan la comunicación. Si un acceso no está dentro de la cache entonces ninguna ventaja se aprovecha, pues ese acceso va directamente a la memoria global.

La memoria de textura fue creada inicialmente para almacenar texturas gráficas, pero CUDA extiende su uso a todo tipo de datos. Las funciones que más optimización brindan son las funciones que tratan datos espaciales en dos o tres dimensiones (NVIDIA_Corporation_a 2011).

Este tipo de memoria debe usarse cuando se cumplen ciertos patrones de acceso, específicamente cuando hilos cercanos acceden a cercanas localizaciones de memoria, la Figura 10 muestra cómo. El cache de texturas de la GPU está especialmente diseñado para acelerar este tipo de accesos.

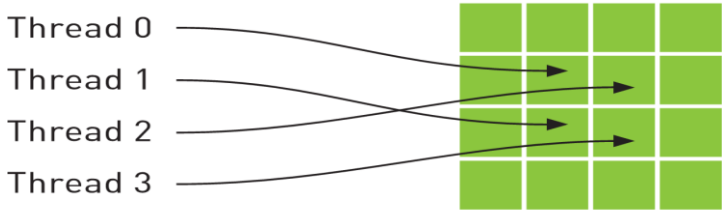


Figura 10 Accesos para la memoria de texturas (NVIDIA_Corporation_a 2011)

1.2.4 Memoria constante

En un dispositivo se encuentran normalmente 64 KB de memoria constante; esta memoria es almacenada en cache. Por lo tanto la lectura aplica el mismo principio que la memoria de texturas (NVIDIA_Corporation_a 2011). Esta memoria también se encuentra físicamente fuera del dispositivo del núcleo de la tarjeta gráfica. Su

uso se recomienda para valores que permanecen intactos durante toda la aplicación.

1.2.5 Registros

Los registros son la unidad de almacenamiento por defecto dentro de un hilo en CUDA. Los registros se encuentran en el chip de la tarjeta gráfica. Los registros se reparten por igual entre la cantidad de hilos paralelos, también hay formas de asignar una cantidad limitada de registros por hilo.

El acceso a los registros no consume ciclos de reloj extra pero existen ciertas situaciones que ocupan esperas por dependencias del tipo lectura-después-escritura. La latencia de estas dependencias es de 24 ciclos de reloj, por eso para cubrir esas dependencias es necesario tener al menos 192 hilos activos en cada multiprocesador. Para una capacidad de cálculo 1.x, 8 núcleos CUDA por multiprocesador por 24 ciclos, es igual a los 192 hilos necesarios para cubrir la latencia. En capacidad de cálculo 2.x que tiene 32 núcleos CUDA por multiprocesadores por 24 ciclos es igual a 768 hilos, lo mínimo requerido para ocultar la latencia (NVIDIA_Corporation_b 2011).

El acceso a estos espacios de memoria se comporta de manera distinta. Por ejemplo, las memorias global, local y de textura tienen las mayores latencias por acceso y luego le siguen, la memoria constante, registros y memoria compartida por ese orden.

1.3 Herramientas utilizadas en el análisis del rendimiento de aplicaciones CUDA

El desarrollo de aplicaciones en CUDA requiere de herramientas que apoyen al programador en el mejor desarrollo de sus programas. Estas herramientas están orientadas a los usuarios que intentan medir el rendimiento de sus programas CUDA y buscan oportunidades para optimizar sus programas.

1.3.1 *Command Line Profiler*

Esta es una herramienta que automáticamente, si está habilitada, recolecta información referente a cada función ejecutada en el dispositivo CUDA. Las opciones de "*profiling*" son controladas por variables de entorno y por un archivo de configuración; las salidas de la herramienta son archivos texto según el valor de estas variables de entorno pueden ser en formatos Separado-por-Coma o Par-Clave-Valor (NVIDIA_Corporation_c 2011).

1.3.1.1 Control del *Command Line Profiler*

El control de la herramienta se realiza usando las variables de entorno siguientes:

COMPUTE_PROFILE: tiene valor 1 o 0 para habilitar o deshabilitar el *profiling*.

COMPUTE_PROFILE_LOG: cuyo valor representa el camino de la salida del *profiling*, si no tiene valor se guarda por defecto en ""cuda_profile_%d.log", donde %d se remplaza por un número, significando el dispositivo donde se ejecuta.

COMPUTE_PROFILE_CSV: tiene valor 1 o 0 para habilitar o deshabilitar el formato de salida separado por coma.

COMPUTE_PROFILE_CONFIG: su valor representa el archivo de configuración a usar por el *profling*. (NVIDIA_Corporation_c 2011)

1.3.1.2 Configuración del *Command Line Profiler*

El archivo de configuración del *profiler* es usado para seleccionar las opciones y los cantadores que van a ser recolectados para la salida después de la ejecución del programa. El archivo tiene un formato simple, que se define con una línea por opción. Cada línea u opción puede ser comentada usando el carácter "#" al inicio de una línea. Cada opción puede representar una o varias columnas en la salida del *profiler* (NVIDIA_Corporation_c 2011).

1.3.1.3 Opciones del *Command Line Profiler*

En la Figura 11 se muestran algunas de las opciones válidas para el *profiler*. Típicamente cada opción representa una columna en el archivo salida, pero hay casos en que varias columnas forman parte de la salida de una opción, en la tabla

se destacan si es necesario, generalmente los nombres de las columnas del archivo de salida son los mismos del nombre de la opción, las excepciones se salvan en la tabla. Generalmente el valor de las columnas es un entero decimal de 32-bit, las excepciones se listan en la tabla.

Opción	Descripción
Timestamp	Tiempo para la ejecución del kernel y transferencias de memoria. Los valores son microsegundos en punto flotante de simple precisión.
gpustarttimestamp	Tiempo del kernel cuando comienza a ejecutar en la GPU. Los valores son nanosegundos en hexadecimal en 64-bits sin signo.
gpuendtimestamp	Tiempo del kernel cuando termina la ejecución en la GPU. Los valores son nanosegundos en hexadecimal en 64-bits sin signo.
Gridsize	Numero de bloques en una malla, por las dimensiones X e Y. Esta opción brinda 2 columnas: gridsizeX gridsizeY
gridsize3d	Numero de bloques en una malla, por las dimensiones X, Y y Z. Esta opción brinda 3 columnas: gridsizeX gridsizeY gridsizeZ
Opción	**Descripción**

Figura 11 Opciones del Command Line Profiler (NVIDIA_Corporation_c 2011)

En el Anexo 1 se muestra la tabla completa.

1.3.1.4 Salida del *command line profiler*

Si la variable de entorno COMPUTE_PROFILE tiene el valor 1 para habilitar el *profiling*, el *profiler* guarda información para cada ejecución de una función *kernel* y cada operación de memoria realizada por el driver de CUDA.

La Figura 12 muestra la salida típica para un *kernel* con las opciones y contadores especificados. Apréciese en la figura algunas de las opciones utilizadas, como referencia a posibles análisis.

```
gridsize
threadblocksize
memtransfersize
memtransferdir
instructions
branch
cta launched
```

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE_NAME 0 GeForce GTX 280
timestamp,method,gputime,cputime,gridsizeX,gridsizeY,threadblocksizeX,t
hreadblocksizeY,
threadblocksizeZ,occupancy,instructions,branch,cta_launched,memtransfer
size,memtransferdir
timestamp=[ 6492.515 ] method=[ _Z10dmatrixmulPfiiS_iiS_ ] gputime=[
25.472 ] cputime=[ 203.797 ]
gridSize=[ 2, 1 ] threadblocksize=[ 32, 8, 8 ] occupancy=[ 0.333 ]
instructions=[ 2261 ]
branch=[ 312 ] cta_launched=[ 2 ]
timestamp=[ 7031.061 ] method=[ memcopy ] gputime=[ 8.896 ] cputime=[
230.686 ]
memtransfersize=[ 8192 ] memtransferdir=[ 1 ]
```

Figura 12 Salida del Command Line Profiler (NVIDIA_Corporation_c 2011).

1.3.2 *Compute Visual Profiler*

El *Visual Profiler* tiene estrecha relación con el *Cammand Line Profiler*, la relación se basa en las opciones y contadores permitidos por cada uno, las cuales son un subconjunto del *Cammnad Line* para el *Visual Profiler*. Es decir, el *Visual Profiler* muestra una idea visual de los análisis realizados por al *Command Line Profiler*, además de incluir otras opciones y contadores que son la combinación de

opciones y contadores estándares en el *Command Line Profiler* (NVIDIA_Corporation_d 2011).

Después de la correcta instalación de todo el software para tener puesto a punto la tecnología CUDA, según (Brito 2012), la herramienta *Visual Profiler* está lista para ser usada.

1.3.2.1 Ejecutando el *Visual Profiler*

Para ejecutar el Visual Profiler, es necesario ir a:

Start->All Programs->NVIDIA Corporation->CUDA Toolkit->3.2-.Compute Visual Profiler

Ejemplos pre-ejecutados se encuentran disponibles en el subdirectorio, \computeprof\projects, del directorio base donde se encuentre instalado el CUDAToolkit. Cuando el Visual Profiler es ejecutado se muestra una ventana, como en la Figura 13:

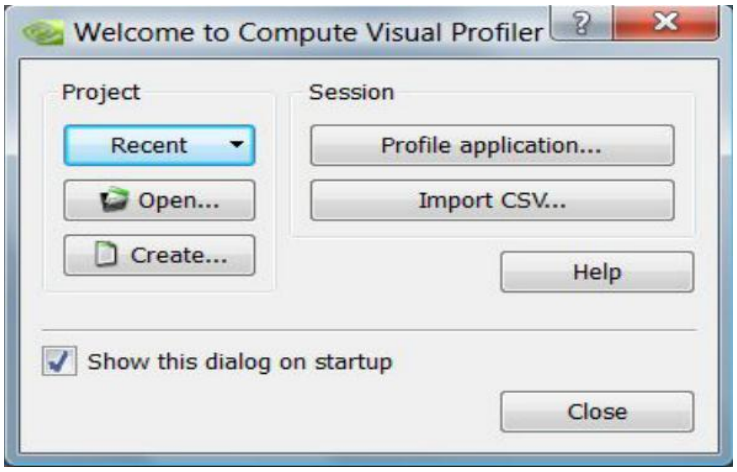


Figura 13 Ventana de inicio del Visual Profiler

1.3.2.2 Archivos y configuraciones en *Compute Visual Profiler*.

Durante la ejecución de la aplicación el proceso de *profiling* puede ser habilitado o deshabilitado a través del menú principal, en opción de la barra de herramientas o en el *checkbox* en la configuración del inicio de sesión.

Cada ejecución de una aplicación es conocida como una sesión. Se recomienda guardar datos de cada *profiling* para múltiples sesiones. Un conjunto de sesiones es conocido como un proyecto. El proceso de *profiling* salva tres archivos

diferentes: .cvp, archivo que contiene el proyecto, .csv, archivo que contiene los datos para una sesión, .trc, archivo que contiene rastro de la API (NVIDIA_Corporation_d 2011).

1.3.2.3 Interfaz Gráfica de Usuario.

En esta sección se describe brevemente la interfaz gráfica de usuario del *Visual Profiler*. La Figura 14, muestra la interfaz gráfica de usuario luego de cargar un proyecto pre-ejecutado.

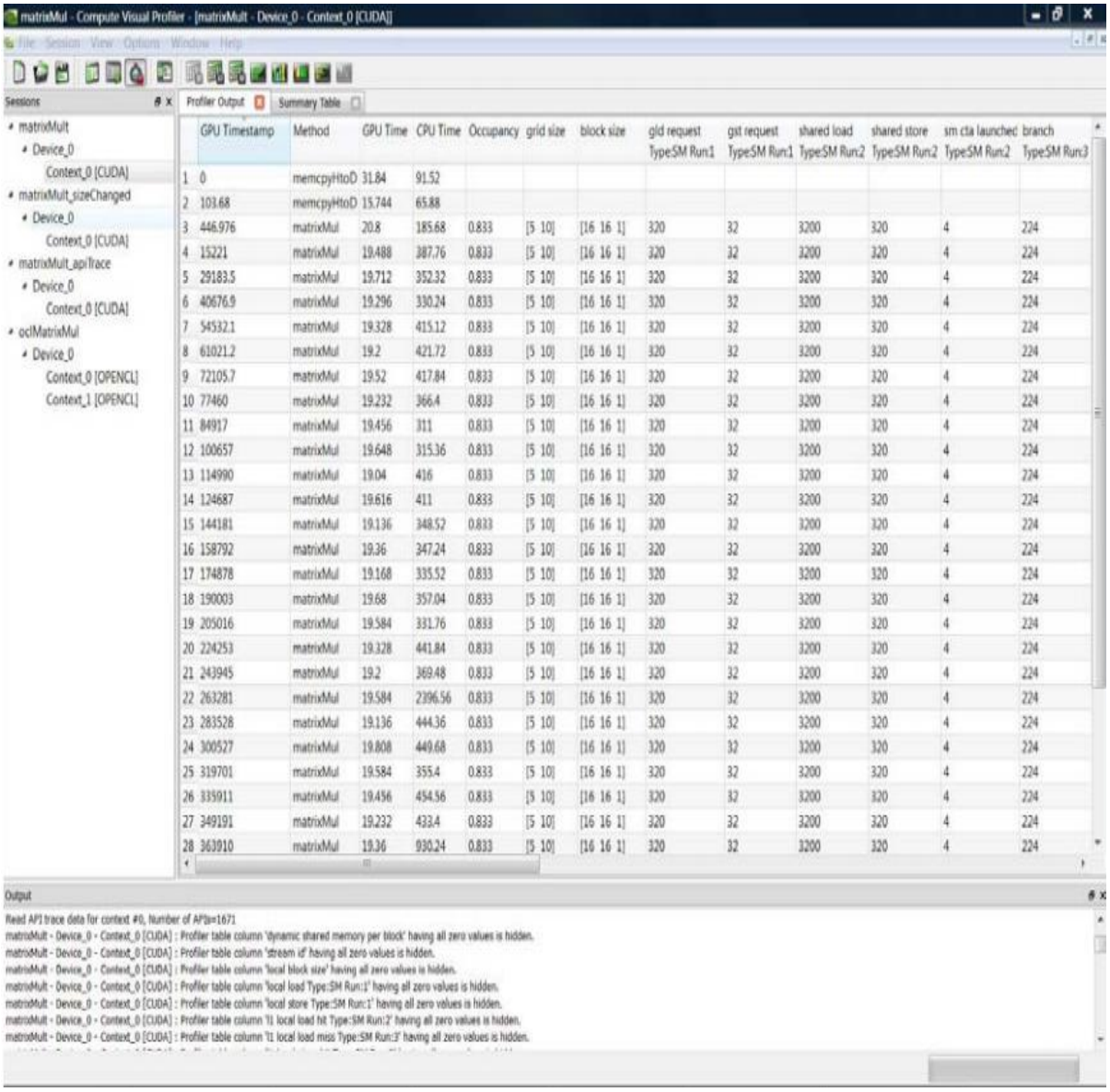


Figura 14 Interfaz gráfica de usuario.

Como se ve en la Figura 14, en el panel izquierda llamado *Session* se organizan las sesiones de un proyecto. Las sesiones son representadas según un árbol de tres niveles.

En el primer nivel se encuentra la sesión cuyo nombre por defecto es *session1*, session2, etc. En este nivel se guardan los datos de las opciones del *profiler*, datos que son mostrados a través de tablas, *plots*, contadores y gráficos.

En el segundo nivel se encuentra el *device*. Información acerca de cada GPU fisica en la computadora donde se ejecuta la aplicación. Las GPUs se nombran como *device*_#, donde # se cambia por el número del *device* donde se ejecuta la aplicación. Para configuraciones con más de una tarjeta grafica, los *devices* se enumeran secuencialmente comenzando por cero. En el tercer nivel se representa el *context*. Cuyo nombre es *Context_<context_number> [CUDA|OPENCL]* donde *context_number* empieza en 0 (NVIDIA_Corporation_d 2011).

Haciendo click derecho en una sesión se muestra un menú con tres opciones, relacionadas con personalizar la sesión, las cuales son:

- *Rename*: renombrar la sesion.
- *Delete*: borrar la sesion.
- *Copy setting to current*: copiar la configuacion actual para ser usada en el proximo proceso de profiling.
- *Plots* que resumen la sesion: muestra los *plots* de utilización de la GPU.

1.3.2.4 El Panel de la derecha.

El panel de la derecha muestra la información relativa a lo que esta seleccionado en el panel de la izquierda. Como son los contadores seleccionados o las opciones del *profiling*.

Cuando se hace click en el panel izquierdo sobre una sesión, en el panel derecho se muestra información básica sobre la sesión, como proyecto, *device*, nombre, ubicación, contadores, opciones, etc.

Desde el panel izquierdo se puede seleccionar una vista especial del *device* haciendo click derecho en el y seleccionando *Device Summary Plot*, lo que muestra un gráfico como el de la Figura 15.

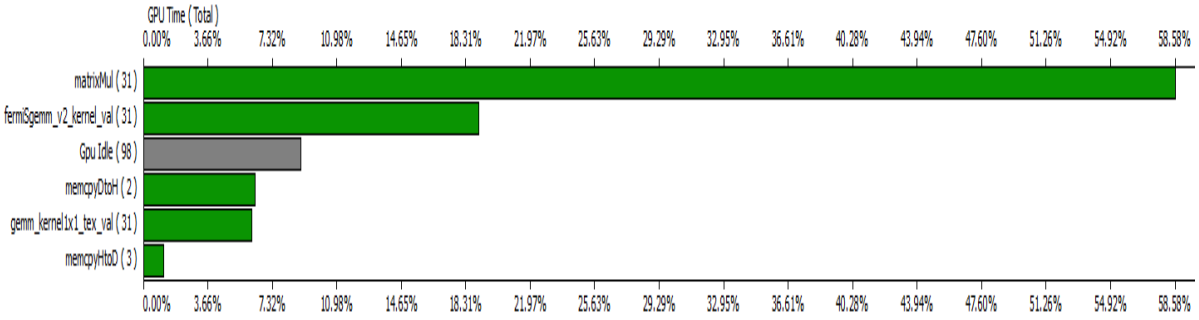


Figura 15 *Device Summary Plot*.

Haciendo click derecho en un *contex*, se pueden seleccionar una de las siguientes opciones Summary Table, Kernel Table, Memcopy table, GPU time summary plot, GPU time height plot, GPU time width plot, Comparisson summary Plot, CUDA API trace. Para cada opción selecciona, en el panel de la derecha se muestran en forma de *tabs* las opciones requeridas. Haciendo click derecho en cada *tab*, se puede personalizar la tabla, como mostrar más o menos columnas, etc.

1.3.2.5 Panel de abajo, *Output Frame*

En el panel de abajo o panel de salida se muestra los errores que pueden suceder y cualquier mensaje o estado del programa que se está ejecutando.

1.3.2.6 Panel de Análisis

Compute Visual Profiler contiene una poderosa herramienta de análisis al nivel de *kernel, device, context* y *session*. El panel de análisis se muestra debajo del panel derecho y contiene información sobre el nivel que está señalado en el panel izquierdo. Además muestra indicaciones generales a manera de mejorar el rendimiento de la aplicación y propone otros análisis que se han realizado para la aplicación.

1.3.2.7 Interpretación de los contadores

Parte importante del *Visual Profiler* es saber el significado que tienen cada uno de los valores y análisis que realiza el *profiler*. En particular los contadores no son solo números después de la inspección de la ejecución de un *kernel*, sino que

tienen que tomarse como una referencia en cuanto a diferencias sobre un código sin optimizar y otro optimizado.

Por ejemplo si en el programa original se tienen N cargas de memoria global no fusionada, el éxito se logra si en los posteriores análisis del programa mejorado el número de cargas de memoria global no fusionada fuese menor que N. En otras palabras, se trata de llevar N a 0 para lograr un funcionamiento perfecto.

En el anexo 1 se muestra la tabla con todos los contadores disponibles.

Capítulo 2: Aplicaciones de CUDA en el área de la Bioinformática

En Bioinformática se han implementado aplicaciones basadas en la Unidad Grafica de Procesamiento (GPU), como análisis de datos de genes expresados en micro arreglos, alineación de secuencias y simulación de sistemas biológicos. En este capítulo se presenta un conjunto de aplicaciones desarrolladas por investigadores en el campo de la Bioinformática que demuestran el creciente estudio y uso de la tecnología CUDA en este campo.

2.1 Análisis paralelo de secuencias de ADN mediante el uso de GPU y CUDA.

El número de genomas secuenciados es cada vez mayor y es necesario que las herramientas informáticas se adapten a las nuevas tecnologías para explotar estas nuevas formas y para en primera instancia acelerar el análisis de estas secuencias. La comparación de secuencias cortas de ADN es, posiblemente, el análisis más repetido en bioinformática (S. Vinga 2003). El número de comparaciones se realiza de acuerdo al tamaño *k* de nucleótidos que se busca dentro de la secuencia.

En el trabajo (A. J. Peña 2010), los autores desarrollaron una metodología para la obtención de las frecuencias de aparición de todas las palabras de *k* nucleótidos de una secuencia de ADN. Para ello, el carácter A lo codificaron como “00”, la C como “01”, la G como “10” y la T como “11”. Según se van conformando las palabras leídas en la secuencia de ADN, se obtiene un código binario de *k*2* bits que se utiliza para acceder directamente a la posición de la tabla que tiene las frecuencias de aparición.

2.1.1 Primera Aproximación usando la GPU.

La secuencia de ADN se almacenará en la memoria global de la GPU mientras que los hilos se reparten por igual las posiciones de la tabla de frecuencias. La secuencia de ADN será procesada por partes, de modo que su tamaño no exceda el disponible en la memoria compartida de los bloques. O sea, una vez cargada en

memoria global, se carga una porción en la memoria compartida de cada bloque y cada hilo dentro del bloque recorre esa secuencia para buscar la sub-secuencia de *k* elementos que le corresponde.

Este *kernel* se ejecutó con una malla de 16 bloques y 64 hilos en cada bloque (1024 hilos en total) y los resultados fueron peores que los esperados, aun probando con otros tamaños de malla y bloque, no lograron una mejora del rendimiento.

2.1.2 Segunda aproximación usando GPU.

En la segunda versión se mapea la secuencia de ADN sobre la memoria de texturas, para así provechar su menor tiempo de acceso y el hecho de que está en caché. Los resultados son similares a la primera aproximación y por tanto sin mejoras significativas.

2.1.3 Tercera aproximación usando GPU.

La idea principal de este algoritmo es posibilitar la actualización de la misma posición de memoria compartida por distintos hilos, según el ejemplo "historiogram256" (NVIDIA_Corporation_e 2011). En esta aproximación se asignó una tabla de frecuencias por *warp* en la memoria compartida de cada bloque. Así, los accesos a la memoria compartida se dividen entre los distintos *warps* de cada bloque.

Cada hilo recorre la memoria global según la cantidad de frecuencias que tenga que calcular. Cada *warp* mantiene en la memoria compartida una tabla de frecuencias. Cuando todos los hilos alcancen el fin de sus respectivas tablas de frecuencias, se resumen estas frecuencias en una nueva tabla de frecuencias en la memoria global. De este modo en la memoria global permanece una tabla por cada bloque de hilos, la cual es resumida a una sola tabla final después de la ejecución de otro *kernel*.

Los resultados para esta aproximación superan los anteriores pero no los esperados al principio del proyecto, por lo tanto se avanza a una nueva aproximación de la mejor solución.

2.1.4 Cuarta Aproximación usando GPU.

En este caso, la secuencia de ADN a analizar se divide en 32 partes, una por cada uno de los hilos que forman un *warp* y la tabla de frecuencias se divide entre los distintos *warps* de todos los bloques. Debido a esta división de las frecuencias se tendrá una sola copia en memoria compartida de la tabla de frecuencias que al finalizar se envía a la memoria global. Cuando toda la ejecución termine cada *warp* debió recorrer toda la secuencia en la memoria global.

Los resultados obtenidos en esta aproximación son peores que los de la aproximación anterior.

2.1.5 Quinta aproximación usando GPU.

La quinta y última aproximación se basa en pre-procesar la secuencia de ADN en la CPU y luego enviar a la GPU todas las sub-secuencias de tamaño *k*, siguiendo el mismo procedimiento que en la primera aproximación a partir de que la secuencia esté en la memoria global. En esta versión el rendimiento disminuye más lento que en las aproximaciones anteriores, pero nuevamente el resultado no es el esperado al inicio de su trabajo.

2.1.6 Observaciones para este epígrafe.

A pesar que se elaboraron cinco versiones de la aplicación usando la GPU, no fueron capaces de sacarle provecho a las ventajas de este modelo de programación. Una de las recomendaciones principales para optimizar una aplicación usando CUDA es el uso de la memoria (NVIDIA_Corporation_b 2011).

La memoria en este caso fue usada como un recurso poco valioso, teniendo en los cinco ejemplos copias del dato, por la secuencia de ADN y por la tabla de frecuencias. Ambas copias se tenían en las diferentes memorias del dispositivo CUDA. Además la tabla de frecuencias estaba diseñada para todas las posibles combinaciones de sub-secuencias de *k* elementos, incluyendo sub-secuencias que nunca van a estar en la secuencia que se analiza.

2.2 Cálculo de *Mutual Information* para inferir redes regulares de genes (Shi 2011).

Mutual Information es usada como una cantidad para medir la dependencia de dos variables discretas aleatorias con *M* posibles valores. Para variables aleatorias continuas es necesario dividir el conjunto continuo en *R* elementos discretos denominados *Bins {A1...AR}.* Cada *Bin* puede contener varios datos. Se asume que la variable continua está dividida en *M* medidas. Dado lo anterior de define una función, θ_j (X_M), que determina si un valor *X* de *M* está dentro del *Bin j*. Se introduce además, una matriz de pesos *WM* (X_{ij}), que significa el peso de X_i en el j-ésimo *Bin*.

$$p'(a_j) = \frac{1}{M}\sum_{i=1}^{M}\theta_j(x_i) \qquad (2.1)$$

$$\theta_j(x_i) = \begin{cases} 1 \text{ if } x_i \in a_j \\ 0 \text{ otherwise} \end{cases} \qquad (2.2)$$

Para calcular la *Mutual Information* entre dos variables aleatorias se necesita calcular la probabilidad de cada *Bin* para cada variable. Una vez calculadas las probabilidades, se calcula la entropía de cada variable y después se calcula la probabilidad conjunta entre las dos variables. Finalmente, se calcula la entropía conjunta entre las dos variables y se termina calculando la *Mutual Information*. Nótese en las ecuaciones 2.3, 2.4, 2.5 y 2.6 las dependencias de cálculo.

$$H(X) = -\sum_{i=1}^{M} p(x_i)log(p(x_i)) \qquad (2.3)$$

$$H(X,Y) = -\sum_{i=1}^{M}\sum_{j=1}^{M} p(x_i,y_j)log(p(x_i,y_j)) \qquad (2.4)$$

$$p'(a_j, b_k) = \frac{1}{M} \sum_{i=1}^{M} (\theta_j(x_i) \times \theta_k(y_i)) \qquad (2.5)$$

$$MI(X, Y) = H(X) + H(Y) - H(X, Y) \qquad (2.6)$$

2.2.1 Función B-*spline*.

La función B-*spline* determina los pesos que se utilizan en la matriz anteriormente descrita, no para construir la matriz. Un vector t_i con *R Bins* y el orden *k* es definido en la ecuación 2.7. El orden *k* debe cumplir la condición siguiente *k € {1,..., R - 1}*.

$$t_i = \begin{cases} 0 \text{ if } i < k \\ i - k + 1 \text{ if } k \leq i \leq R - 1 \\ R - k + 1 \quad i > R - 1 \end{cases} \qquad (2.7)$$

La función se B-*spline* se define como sigue:

$$B_{i,1}(z) = \begin{cases} 1 & \text{if } t_i \leq z \leq t_{i+1} \\ 0 & \text{otherwise} \end{cases} \qquad (2.8)$$

$$B_{i,k}(z) = B_{i,k-1}(z) \frac{z - t_i}{t_{i+k-1} - t_i} + B_{i+1,k-1}(z) \frac{t_{i+k} - z}{t_{i+k} - t_{i+1}} \qquad (2.9)$$

Donde *i* € [1, R] y *z* € [0, R - k + 1] es el dominio de la función.

2.2.2 Implementación

En esta solución se emplearon cuatro *kernels*, la que necesita la secuencia de genes a analizar y también un número de medidas o experimentos. El proceso comienza con la carga en memoria de la GPU de la representación de los genes y se ejecutan los *kernel*. El *kernel 1* calcula la matriz *WM* para cada gen. Dada esta situación todos los hilos en cada bloque trabajan en paralelo para calcular las probabilidades de su sub-secuencia local de genes. Las probabilidades de cada

gen son guardadas en una matriz de pesos. Posteriormente, se ejecuta un *kernel 2* para chequear la integridad de los genes en estas matrices. Este chequeo solo funciona para los elementos en blanco en una matriz, dados por experimentos perdidos en la secuencia de genes.

2.3 Paralelización de algoritmos de optimización basados en poblaciones por medio de GPGPU (LIERA 2011).

El interés de este trabajo se centró en los métodos heurísticos basados en poblaciones cuyos principales exponentes son los Algoritmos Genéticos y los Métodos de Optimización por Enjambres de Partículas.

En el método de Optimización por Enjambres de Partículas se utilizó la siguiente estrategia:

- Se definieron B enjambres de t partículas para ser procesados por un bloque. Figura 16

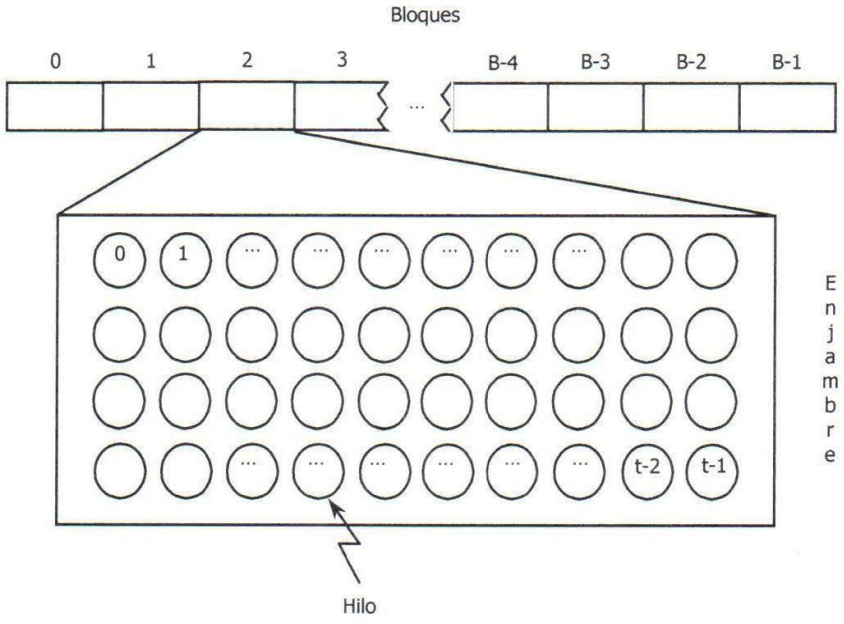


Figura 16 Un enjambre de t partículas se procesó en un bloque.

- A cada hilo dentro del bloque se le asignó la tarea de actualizar los parámetros de una partícula.

- Se definió un periodo migratorio del 1% del total de generaciones. La migración se implementó recibiendo la información de la mejor partícula del bloque siguiente, en forma circular, como se muestra en la Figura 17.

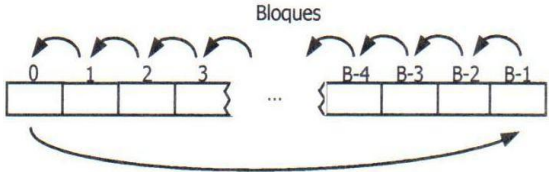


Figura 17 Período de migración.

Para el método de Algoritmos Genéticos se utilizó la siguiente estrategia:

- Se definieron B poblaciones de μ individuos para ser procesadas por un bloque. Figura 18.
- A cada hilo del bloque, se le asignó la tarea de actualizar los parámetros de un individuo. Figura 18.

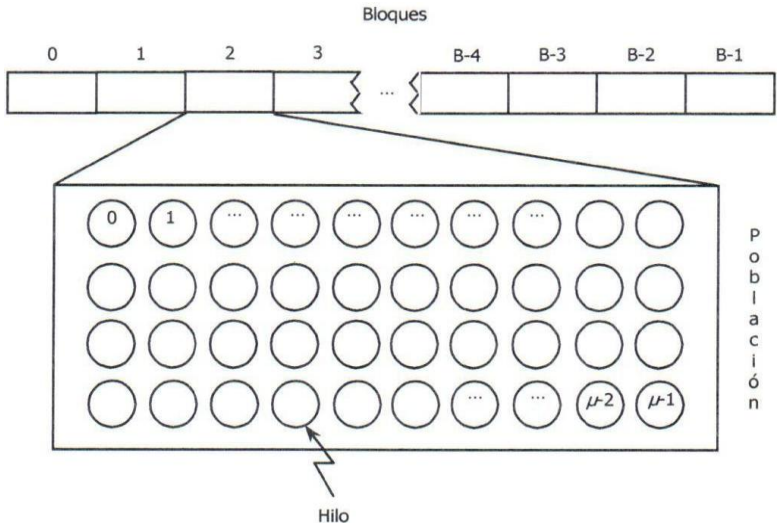


Figura 18 Distribución de hilos por bloques.

- Se paralelizaron los operadores de selección, mutación y migración. El cruce se realizó en la GPU, utilizando un solo hilo para procesar todos los individuos.
- La migración se llevó a cabo con un periodo del 10% del total de generaciones y una tasa de migración (MR) aproximada al 3% de la población. Primeramente, se ordenó la población, para poder migrar en los

MR peores individuos del bloque actual (ubicados al final del bloque), los MR mejores del bloque siguiente. Figuras 19 y 20.

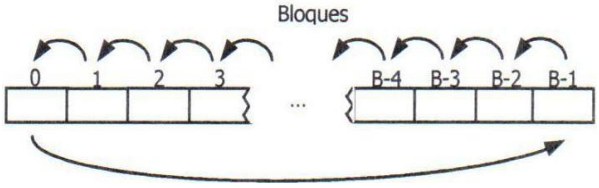


Figura 19 Migración

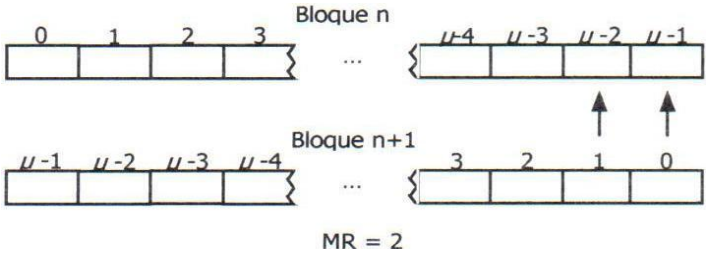


Figura 20 Un paso en la migración.

Debido a las limitaciones en cuanto a la memoria disponible en la GPU, como estrategia de paralelización se determinó utilizar varios bloques en los que se alojó una población con pocos individuos, (en el caso del Algoritmo Genético) o un enjambre con pocas partículas, (en el caso de la Optimización por Enjambre de Partículas). Para que los bloques pudieran colaborar a través de la migración, fue necesario implementar las distintas etapas de los algoritmos en *kernels* separados; por este motivo se requirió sincronizar los hilos en el host, con el costo en tiempo de ejecución que ello implica.

La implementación propuesta con arquitectura CUDA es más eficiente en cuanto al tiempo de ejecución comparado con los resultados obtenidos en un clúster de 16 computadoras.

Capítulo 3: Implementación de un caso de estudio en el área de la Bioinformática.

En este capítulo se abordan los elementos teóricos necesarios para introducir el caso de estudio: cálculo de la matriz de distancia. A partir de esa información se explica una variante de paralelización en CUDA del problema anterior y se valora usando *Visual Profiler* la implementación realizada.

3.1 Modelos evolutivos en el análisis de secuencias genéticas.

El análisis del proceso evolutivo entre diferentes secuencias de nucleótidos o aminoácidos se ha convertido en un proceso vital en el análisis de la evolución de la vida. La comprensión de los cambios ocurridos durante el proceso evolutivo permite interpolar adecuadamente nuestras observaciones actuales con el fin de poder hacer predicciones inteligentes sobre observaciones futuras.

Este estudio se realiza a través de la aplicación de modelos evolutivos probabilísticos, también conocidos como procesos de Markov. La modelación mediante procesos de Markov de las sustituciones nucleotídicas usada en el cálculo de la distancia entre secuencias, forma la base del análisis bayesiano y de verosimilitud de múltiples secuencias en una filogenia (Yang 2006). Esta modelación permite la interpretación evolutiva y aplicación de las filogenias entre diferentes secuencias.

Actualmente, existen dos aproximaciones para construir modelos de evolución de secuencias.

- Construcción de modelos empíricos basados en propiedades del proceso de sustitución calculadas a partir de comparaciones de un gran número de alineamientos. Los modelos empíricos resultan en valores fijos de los parámetros, los cuales son estimados sólo una vez, suponiéndose que son adecuados para el análisis de otros conjuntos de datos.

- Construcción de modelos paramétricos basado en el modelado de propiedades químicas o genéticas de aminoácidos y nucleótidos. En los modelos paramétricos los valores de los parámetros pueden ser inferidos de cada conjunto de datos al hacer un análisis de los mismos usando métodos de máxima verosimilitud (ML) o Bayesianos (Bayes).

Ambos métodos resultan en modelos de procesos de Markov, definidos por matrices que contienen las tasas relativas de ocurrencia de todos los tipos posibles de sustituciones.

Las sustituciones (reemplazos evolutivos de los estados de carácter) se describen como el resultado de mutaciones al azar. Su aparición en cada posición de una secuencia a lo largo del tiempo es modelado por un proceso de Markov. La probabilidad de intercambio de un estado de carácter por otro viene modelada en esencia por una distribución de Poisson (de eventos raros).

Así, un proceso Markoviano constituye un modelo matemático de eventos raros de cambios en estados (discretos o de carácter) a lo largo del tiempo, en el que los eventos futuros suceden por azar y dependen únicamente del estado actual, y no de la historia que llevó a dicho estado.

En filogenética, los estados del proceso son los nucleótidos presentes en una posición particular de una secuencia (estados de carácter) en un tiempo dado; los cambios de estado representan las mutaciones en dichas secuencias.

El modelado del proceso de sustitución nucleotídica se ha concentrado en la aproximación paramétrica.

Se manejan tres tipos principales de parámetros en estos modelos:

- Parámetros de frecuencia.

- Parámetros de tasas de intercambio.
- Parámetros de heterogeneidad de tasas de sustitución entre sitios.

Los parámetros de frecuencia describen las frecuencias de las bases (A, C, G, T) en una secuencia DNA, promediadas sobre todas las posiciones y a lo largo del árbol.

Los parámetros de tasa (de intercambiabilidad) describen las tendencias relativas de las bases de ser sustituidas unas por otras. Así podemos encontrar transiciones entre purinas (A $\leftrightarrow$ G) o pirimidinas (C $\leftrightarrow$ T) y transversiones entre purinas y pirimidinas (A $\leftrightarrow$ C, A $\leftrightarrow$ T, C $\leftrightarrow$ G, y G $\leftrightarrow$ T).

La aproximación más utilizada para modelar la heterogeneidad de tasas entre sitios es la de describir la tasa de sustitución de cada posición como una muestra aleatoria de una distribución GAMMA. El uso de esta distribución para modelar la heterogeneidad de tasas entre sitios representa un factor muy significativo para incrementar el ajuste entre los modelos y los datos. Esta distribución gamma se emplea en conjunción con los parámetros de frecuencia y tasa.

Los diversos modelos evolutivos se distinguen por su grado de parametrización:

a) Frecuencias de los nucleótidos

   - Los que consideran que todas las bases tienen igual frecuencia de sustitución para las cuatro bases (πA = πC = πG = πT = 0.25), entre ellos **JC69** (Jukes and Cantor, 1969), y **K80** (Kimura, 1980) entre otros.

   - Los que consideran que todas las bases tienen diferente frecuencia de sustitución para las cuatro bases (πA ≠ πC ≠ πG ≠ πT), entre ellos **F81** (Felsenstein 1981), **HKY85** (Hasegawa, Kishino and Yano 1985) y **TN93** (Tamura and Nei 1993), entre otros.

b) Tasas de sustitución transicionales (ti) y transversionales (tv)

   - Existen 4 tipos de sustituciones ti y 8 tv.

   - Los modelos evolutivos se diferencian también en la cantidad de parámetros que utilizan para acomodar diversas tasas de sustitución.

JC69, **F81** → todas las sustituciones tienen igual tasa $\alpha=\beta$

K80 → distintas tasas de sustitución $\alpha\neq\beta$

De la anterior descripción encontramos modelos más realistas que consideran distintas frecuencias para las bases: $\pi A \neq \pi C \neq \pi G \neq \pi T$, así como distintas tasas de sustituciones ti y tv; $\alpha\neq\beta$, entre ellos HKY85 y TN93. El modelo TN93, a diferencia del HKY85 considera 3 tasas de sustitución: α_1, α_2 y β.

Otros modelos más recientes consideran el proceso de evolución con una base nucleotídica adicional (A, G, C, T y D) (Sánchez y Grau 2009).

3.1.1 El modelo TN93.

Para este modelo TN93 las transiciones entre purinas ocurren a una razón distinta a la de aquella entre pirimidinas. También se asume que todas las transversiones ocurren con la misma velocidad, aunque esta puede ser distinta de las tasas de transición. Por último se permite que las cuatro bases tengan distintas distribuciones estacionarias.

Como resultado la matriz de las tasas de sustitución de nucleótidos es:

$$Q_r = \begin{bmatrix} -(\alpha_2\pi_G + \beta\pi_Y) & \beta\pi_C & \alpha_2\pi_G & \beta\pi_T \\ \beta\pi_A & -(\alpha_1\pi_T + \beta\pi_R) & \beta\pi_G & \alpha_1\pi_T \\ \alpha_2\pi_A & \beta\pi_C & -(\alpha_2\pi_A + \beta\pi_Y) & \beta\pi_T \\ \beta\pi_A & \alpha_1\pi_C & \beta\pi_G & -(\alpha_1\pi_C + \beta\pi_R) \end{bmatrix} \quad (3.1)$$

donde:

π_A, π_C, π_G y π_T son las frecuencias estacionarias de los nucleótidos A, C, T y G respectivamente y $\pi_R = \pi_A + \pi_G$ y $\pi_Y = \pi_C + \pi_T$. Los parámetros α_1, α_2 y β son, respectivamente, la tasa de transiciones entre pirimidinas, entre purinas y la tasa de transversiones.

Para deducir una fórmula que permita estimar la divergencia o distancia (d) entre dos secuencias nucleotídicas alineadas, se debe conocer la proporción esperada de sitios con diferencias transicionales entre purinas (P1) y entre pirimidinas (P2) y de aquellos con diferencias transversionales (Q), expresadas en función de las tasas de sustitución y el tiempo evolutivo (Tamura 1992).

Estos se pueden determinar según las fórmulas siguientes:

$$P_1 = \frac{2\pi_A\pi_G}{\pi_R}\left(1-\frac{\pi_R}{2\pi_A\pi_G}e^{-2\beta t}-\frac{1}{2\pi_R}e^{-2(\pi_R\alpha_1+\pi_Y\beta)t}\right) \quad (3.2)$$

$$P_2 = \frac{2\pi_C\pi_T}{\pi_Y}\left(1-\frac{\pi_Y}{2\pi_C\pi_T}e^{-2\beta t}-\frac{1}{2\pi_Y}e^{-2(\pi_Y\alpha_2+\pi_R\beta)t}\right) \quad (3.3)$$

$$Q = 2\pi_R\pi_Y\left(1-e^{-2\beta t}\right) \quad (3.4)$$

Como P1, P2, Q, πA, πC, πG y πT pueden ser estimados a partir del alineamiento de las dos secuencias que se están comparando, se puede obtener entonces el siguiente estimado de d:

$$d' = -\frac{2\pi_A\pi_G}{\pi_R}\log_e\left(1-\frac{\pi_R}{2\pi_A\pi_G}P_1'-\frac{1}{2\pi_R}Q'\right)-\frac{2\pi_C\pi_T}{\pi_Y}\log_e\left(1-\frac{\pi_Y}{2\pi_C\pi_T}P_2'-\frac{1}{2\pi_Y}Q'\right) \quad (3.5)$$

$$-2\left(\pi_R\pi_Y-\frac{\pi_A\pi_G\pi_Y}{\pi_R}-\frac{\pi_C\pi_T\pi_R}{\pi_Y}\right)\log_e\left(1-\frac{1}{2\pi_R\pi_Y}Q'\right) \quad (3.6)$$

donde x´ es el estimado del parámetro x correspondiente. El símbolo ‘ fue omitido en los estimados de las frecuencias para ganar en claridad. (Tamura and Nei 1993).

Si este procesamiento se realiza a un conjunto de secuencias en estudio obtendremos la matriz de distancias evolutivas entre las mismas.

El cálculo de esta matriz puede acarrear un costo computacional de acuerdo al número de secuencias a procesar y a la longitud de las mismas. Debido a ello se ha explorado su implementación paralela usando la biblioteca de paso de mensajes MPI (Moreira y otros 2011).

Esta matriz de distancias evolutivas podrá ser empleada posteriormente para la construcción del árbol filogenético que describe la filogenia entre estas especies.

3.2 Implementación en CUDA del cálculo de la matriz de distancia.

Teniendo en cuenta las características del cálculo de la matriz de distancia se ha dividido el análisis de su paralelización en dos momentos. El primero es durante el cálculo de los parámetros de frecuencia, parámetros de tasas de intercambio y los parámetros de heterogeneidad de tasas de sustitución entre sitios y el segundo es durante la obtención de los valores de cada una de las distancias de la matriz de distancias.

3.2.1 Cálculo de la matriz de distancia: paralelización del cálculo de los parámetros

La implementación consiste en paralelizar el análisis de cada par de secuencias. Esto se hace a través de una función *kernel* llamada `paso01` que se ejecuta en la GPU y realiza el análisis de dos secuencias. Este análisis consiste en calcular los contadores teniendo en cuenta las bases de cada posición en las dos secuencias de entrada de la función. En la GPU se garantizan N hilos, cada uno analizando un elemento (posición) de cada secuencia, siendo N el tamaño de una secuencia. El resultado de analizar las bases nitrogenadas implica incrementar en uno una posición de la matriz de contadores, según sea el caso. Por cada posición de la secuencia se tiene un contador para cada una de los parámetros a calcular. Ver en la figura 29 una representación de la función `paso01`.

La matriz de contadores tiene siete columnas, pi_A, pi_C, pi_G, pi_T, P1, P2, Q. Representando las columnas pi_A, pi_C, pi_G y pi_T la cantidad de bases “A”, “C”, “G” y “T” (adenina, citosina, guanina y timina) respectivamente. Las columnas P1 y P2 representan transiciones entre purinas y pirimidinas. La columna Q representa las transversiones entre purinas y pirimidinas.

			Matriz de Contadores						
	Sec1	**Sec2**	**PiA**	**PiC**	**PiG**	**PiT**	**P1**	**P2**	**Q**
Hilo 0 →	A	A	2	0	0	0	0	0	0
Hilo 1 →	T	A	1	0	0	1	0	0	1
Hilo 2 →	G	G	0	0	2	0	0	0	0
…	A	G	…						
	A	C							
	G	A							
	G	A							
	C	T							
	A	A							
	A	T							
…			…						
Hilo N →	C	T	0	1	0	1	0	1	0

Figura 21 Matriz de contadores.

La matriz de distancias se calcula en la CPU secuencialmente, dentro de esta secuenciación se ejecuta la función *kernel* `paso01`, tantas veces como elementos tenga la matriz de distancias, es decir la función `paso01` recibe como parámetros una combinación de dos secuencias en cada llamada y un total de S*(S+1)/2 llamadas, donde S es la cantidad de secuencias.

A continuación debe compilarse un resumen de la matriz de contadores. Este resumen consiste de obtener la suma por columnas de la matriz de contadores, es decir, para cada parámetro obtener su suma total.

	Matriz de Contadores						
	PiA	PiC	PiG	PiT	P1	P2	Q
	2	0	0	0	0	0	0
	1	0	0	1	0	0	1
	0	0	2	0	0	0	0
	…						
	0	1	0	1	0	1	0
TOTAL	**3**	**1**	**2**	**2**	**0**	**1**	**1**

Figura 22 Resumen de la Matriz de contadores.

Se define una función *device* de nombre `paso02`. Esta función recibe una matriz de contadores y obtiene un vector de 7 elementos (uno por cada parámetro).

Esta función `paso02` se ejecuta en el *host* dentro de un ciclo de $\log_2 N$ iteraciones. En su primera activación `paso02` suma los valores de las posiciones pares con los de las posiciones impares, dos a dos, y coloca el resultado en la posición par, por lo tanto a partir de este momento todas las posiciones pares tienen acumulado una suma parcial. Ver figura 31.

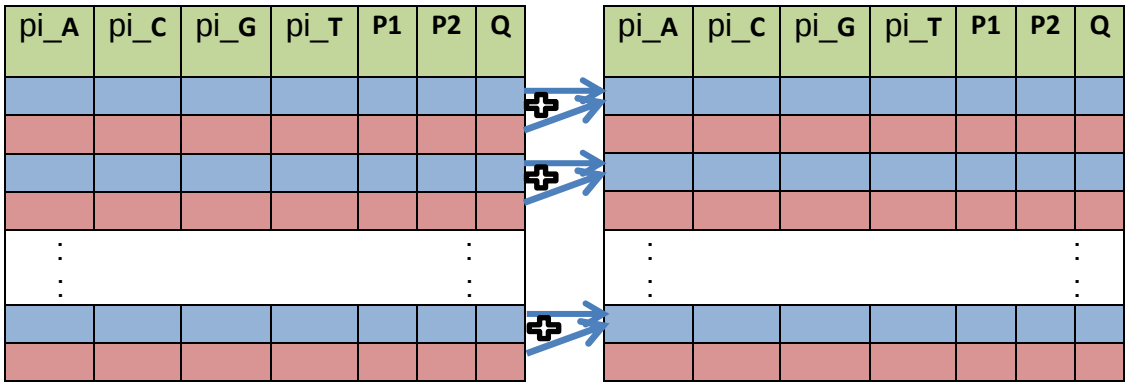


Figura 23 Primer paso en la suma.

Este procedimiento sigue de manera secuencial, pero entre cada par de filas previamente resumidas. Siempre se suman dos filas y se guarda el resultado en la primera de esas dos. El procedimiento se realiza $\log_2 N$ que es la cantidad de iteraciones necesarias hasta alcanzar la suma de todas las filas. La suma se hace para las filas pares, luego para las filas múltiplo de cuatro, a continuación para las filas múltiplo de ocho, y así sucesivamente para cada par de filas múltiplo de 2^N. El resultado final queda en la primera fila de la matriz.

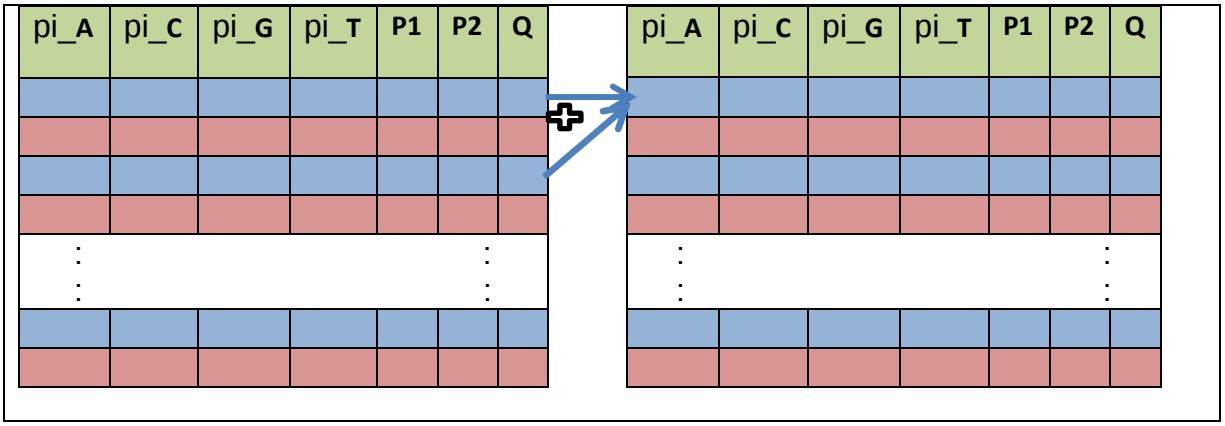


Figura 24 Paso 2 en la suma.

Cuando este procedimiento concluye, se copia desde la GPU la primera fila de la matriz hacia la memoria de la CPU y esos siete valores se copian a siete variables para ser utilizadas en el cálculo de la distancia entre las dos secuencias analizadas, es decir, un elemento de la matriz de distancia. A continuación se muestra un algoritmo general del procedimiento:

P1: Leer todas las secuencias desde el archivo FASTA hacia la variable `Sec`.
P2: Para i = 0… S hacer:
P3 `sec1 = Sec[i]`
P4: Para j = i+1… S hacer:
P5: `sec2 = Sec[j]`
P6: Copiar información a la GPU (`sec1` y `sec2`)
P7: Activar *kernel* `paso01(sec1, sec2, cont)`
P8: Activar *kernel* `paso2(cont, paso)`
P9: Copiar en `piA`, `piC`, `piG`, `piT`, `P1`, `P2` y `Q` los valores de `cont[0]` … `cont[6]`
P10: `MatrizDist[i,j]= MatrizDist[j,i]= dist(piA, piC, piG, piT, P1, P2, Q)`
P11: Fin Para j
P12: Fin Para i

La matriz de distancia es una matriz simétrica, pues la distancia entre una secuencia *x* y una secuencia *y* es la misma que la distancia entre *y* y *x*. Durante la ejecución del algoritmo, en todo momento se necesita espacio en memoria para almacenar la matriz de *SxN* nucleótidos o matriz de secuencias y la matriz de distancias de *SxS*, además de un número constante de variables auxiliares para la realización de los cálculos.

3.3 Valoración de la versión implementada.

Para evaluar la implementación sobre CUDA del cálculo de la matriz de distancias descrita en el epígrafe anterior se decidió utilizar la herramienta “Command Line Profiler”. Para su utilización es necesario habilitar la variable de entorno que activa la ejecución automática de esta herramienta (COMPUTE_PROFILER=1).

Con esta herramienta activada, al ejecutarse una aplicación CUDA, se genera un archivo texto con la información del *profiler*, el archivo por defecto generado es el que se muestra en la Figura 33 para una ejecución del programa con una cantidad de secuencias igual a 125 (archivo Fasta_Matrix_125.fas).

El archivo resultante del *profiler* contiene en las tres primeras líneas las propiedades del dispositivo CUDA y la versión del *profiler*. La línea siguiente representa cuales son los valores a recolectar por el *profiler* y a continuación aparece la información de todos los métodos activados según los valores a recolectar.

Haciendo un análisis de los resultados en los tiempos de ejecución de la CPU y de la GPU se observó que la GPU es más utilizada que la CPU. Sin embargo, solo se reporta como ocupación de la GPU un valor de 0.667, lo que indica que poco más de la mitad de los *warps* de la GPU están activos durante la ejecución de los *kernels*. También puede observarse que durante las transferencias de memoria entre el dispositivo y el *host* el tiempo de la CPU es generalmente mucho mayor que el de la GPU.

Analizando la secuencia de métodos reportados en el archivo esta coincide con la secuencia explicada en el algoritmo del epígrafe anterior: comienza el `paso01` y luego le siguen las llamadas al método `paso02`, y así sucesivamente.

```
# CUDA_DEVICE 0 GeForce 9400 GT
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR 12b6e99e2a724018
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 5.088 ] cputime=[ 11.520 ]
method=[ memcpyHtoD ] gputime=[ 4.288 ] cputime=[ 6.912 ]
method=[ _Z6paso01PcS_Piii ] gputime=[ 35.072 ] cputime=[ 22.271 ] occupancy=[ 0.667 ]
method=[ memcpyDtoH ] gputime=[ 19.232 ] cputime=[ 102.526 ]
method=[ _Z6paso02Pii ] gputime=[ 73.312 ] cputime=[ 4.224 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 55.616 ] cputime=[ 2.688 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 54.368 ] cputime=[ 3.456 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 53.632 ] cputime=[ 3.072 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 53.216 ] cputime=[ 3.072 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 41.568 ] cputime=[ 2.688 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 37.376 ] cputime=[ 2.688 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 35.424 ] cputime=[ 2.688 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 35.328 ] cputime=[ 3.072 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 35.136 ] cputime=[ 2.304 ] occupancy=[ 0.667 ]
method=[ _Z6paso02Pii ] gputime=[ 24.800 ] cputime=[ 2.688 ] occupancy=[ 0.667 ]
method=[ memcpyDtoH ] gputime=[ 4.032 ] cputime=[ 577.906 ]
method=[ memcpyHtoD ] gputime=[ 5.120 ] cputime=[ 9.984 ]
method=[ memcpyHtoD ] gputime=[ 4.320 ] cputime=[ 7.296 ]
method=[ _Z6paso01PcS_Piii ] gputime=[ 34.304 ] cputime=[ 9.600 ] occupancy=[ 0.667 ]
```

Figura 25 Recorte del archivo de salida del command line profiler.

Un análisis similar se realizó en la herramienta *Visual Profiler*. El análisis se realizó con dos archivos de secuencias con diferentes números de secuencias (125 y 150). Las Figuras 25 y 26 muestran un resumen del uso de la GPU para ejecuciones del programa con ambos archivos de secuencias.

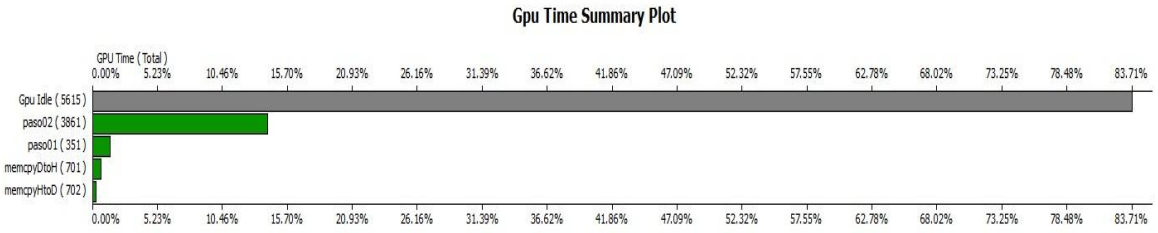


Figura 26 Resumen tiempo utilizado por 150 secuencias.

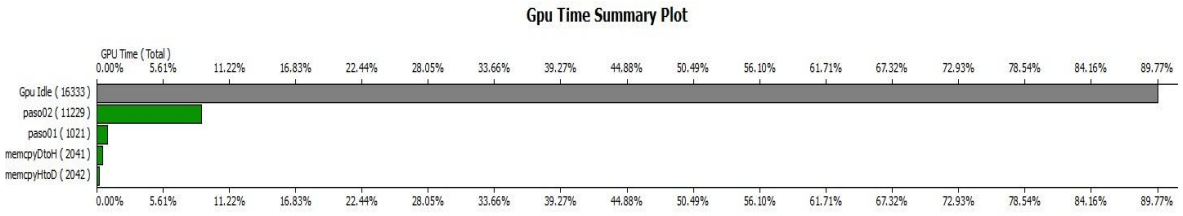


Figura 27 Resumen tiempo utilizado por 125 secuencias.

Se puede observar que en ambos casos la GPU esta sin uso hasta un 89%, lo cual indica que se debe explorar otras variantes de implementación que intenten hacer un mayor uso de la GPU.

3.4 Propuesta de Implementación

En este epígrafe se hace una propuesta de otra implementación del cálculo de la matriz de distancias. La propuesta consiste en lograr asignar a un hilo el cálculo de una distancia, de tal manera que se tenga un hilo por cada uno de los elementos de la matriz de distancia.

La función a ejecutar en la GPU debe estar preparada para determinar que par de secuencias aportan información al elemento de la matriz de distancia que calcula (según el hilo que la está ejecutando). Después de determinar esas dos secuencias calcula los parámetros para ellas y la distancia en una función *device* llamada desde la función *kernel*.

Con esta estrategia se ha movido todo el cálculo de la matriz de distancia a la GPU, esperando un mayor uso de la misma y por tanto un mayor rendimiento.

Algoritmo general propuesto:

P1: Leer todas las secuencias desde el archivo FASTA hacia la variable `Sec`.
P2: Copiar información a la GPU (`Sec`)
P3: Activar *kernel* `paso01(Sec, MD, lenSec, contSec)`
P4: Copiar en la CPU la matriz de distancia MD
P5: Fin

Algoritmo propuesto para la función kernel `paso01`
P1: sea i = idHilo/contSecs //idHilo es un identificador único para cada hilo
P2: sea j = idHilo%contSecs //idHilo es un identificador único para cada hilo
P3: sea `f = paso02 (Sec[i], Sec[j], lenSec)`
P4: MD[idHilo] = f
P5: Fin

Algoritmo propuesto para la función *device* `paso02`
P1: para `j=0` hasta `lensec`
P2: para i=`0` hasta `lensec`
    P3： Actualizar contadores según cada par de elementos
P4: retornar `distFormula(contadores)`

Conclusiones

- Los modelos de memoria para la programación paralela fueron caracterizados en aras de distinguir sus particularidades, describiéndose los 5 tipos de memorias disponibles en CUDA.

- Se constató el acelerado ritmo de desarrollo de CUDA y de las aplicaciones en el área de la Bioinformática, donde se revisaron y comentaron varias aplicaciones.

- Se caracterizaron las variables a medir para el mejor desarrollo de una aplicación paralela a partir del estudio de varias herramientas disponibles para el análisis del rendimiento de aplicaciones con CUDA, haciendo énfasis en el *Visual Profiler Tools* (y su versión de línea de comandos).

- Se implementa el cálculo de la matriz de distancia de un conjunto de secuencias de ADN que es la base de todo análisis filogenético. Como primera aproximación de solución al problema cumple con los elementos fundamentales de la programación usando CUDA.

Recomendaciones

- Continuar estudiando los avances de CUDA, especialmente las aplicaciones de esta tecnología en el área de la Bioinformática.

- Explorar posibles desarrollos en CUDA de algunas de las técnicas utilizadas por el Laboratorio de Bioinformática del Centro de Estudios de Informática.

Referencias Bibliográficas

A. J. Peña, J. M. C., A. Sanjuan, V. Arnau (2010). "Análisis paralelo de secuencias de ADN mediante el uso de GPU y CUDA."

Ananth Grama, A. G., George Karypis, Vipin Kumar (2003). Introduction to Parallel Computing.

Brito, D. I. T. (2012). Acercamiento a la programación paralela usando CUDA. Programación e Ingeniería de Software. Santa Clara, Universidad Central Marta Abreu de las Villas.

Felsenstein, J. (1981). "Evolutionary Trees from DNA Sequences: A Maximum Likelihood Approach." Journal of Molecular Evolution 17: 368-376.

Foster, I. (2003). Designing and Building Parallel Programs.

Hasegawa, M., H. Kishino, et al. (1985). "Dating the human–ape splitting by a molecular clock of mitochondrial DNA." Journal of Molecular Evolution 22: 160-174.

Jason Sanders, Edward Kandrot (2011)." CUDA by example : an introduction to general-purpose GPU programming"

Jukes, T. H. and C. R. Cantor (1969). Evolution of protein molecules in Mammalian protein metabolism. New York, Academic Press.

Kimura, M. (1980). "A simple method for estimating evolutionary rate of base substitution through comparative studies of nucleotide sequences." Journal of Molecular Evolution 16: 111-120.

LIERA, I. C. (2011). PARALELIZACIÓN DE ALGORITMOS DE OPTIMIZACIÓN BASADOS EN POBLACIONES POR MEDIO DE GPGPU. DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN. La Paz, INSTITUTO TECNOLÓGICO DE LA PAZ.

Moreira, J.E.; Chávez, M.C.; Sánchez, R.; Del Toro, L.F; Grau, R. (2011). Implementación paralela de algoritmos básicos empleados en el análisis filogenético de secuencias del virus de la Influenza A H1N1, Poster en CICI 2011, Convención Internacional INFORMÁTICA 2011, febrero 2011, Habana, Cuba, ISBN: 978-959-7213-01-7

NVIDIA_Corporation_a (2011). "C Programming Guide."

NVIDIA_Corporation_b (2011). "C Programming Best Practices Guide."

NVIDIA_Corporation_c (2011). "Compute Command Line Profiler."

NVIDIA_Corporation_d (2011). "Compute Visual Profiler."

NVIDIA_Corporation_e (2011). Examples in CUDA development kit

Quinn, M. J. (2004). Parallel programming in C with MPI and OpenMP.

Sánchez, R. and R. Grau (2009). "An algebraic hypothesis about the primeval genetic code architecture." Mathematical Bioscienses 221: 60-76.

S. Vinga, J. A. (2003). "Alignment-free sequence comparison – a review." 513:523.

Shi, H. (2011). "Parallel mutual information estimation for inferring gene regulatory networks on GPUs." BMC Research Notes **4**(189).

Tamura, K. (1992). "Estimation of the Number of Nucleotide Substitutions when there are strong Transition-Transversion and G+C-Content biases." Molecular Biology and Evolution **9**: 678-687.

Tamura, K. and M. Nei (1993). "Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees." Molecular Biology and Evolution **10**: 512-526.

Yang, Z. (2006). Computational Molecular Evolution, Oxford University Press.

Wikipedia.org. (2012, 15 de Junio 2012). "OpenMP." from http://en.wikipedia.org/wiki/OpenMP.

Yang, Z. (2006). Computational Molecular Evolution, Oxford University Press.

Anexos

Anexo 1 Opciones y contadores para el *Command Line Profiler* y el *Visual Profiler*.

Option	Description
timestamp	Time stamps for kernel launches and memory transfers. This can be used for timeline analysis. The column values are single precision floating point value in microseconds.
gpustarttimestamp	Time stamp when kernel starts execution in GPU. The column values are 64-bit unsigned value in nanoseconds in hexadecimal format.
gpuendtimestamp	Time stamp when kernel ends execution in GPU. The column values are 64-bit unsigned value in nanoseconds in hexadecimal format.
gridsize	Number of blocks in a grid along the X and Y dimensions for a kernel launch. This option outputs the following two columns: CUDA: • gridsizeX • gridsizeY OpenCL: • ndrangesizeX • ndrangesizeY
gridsize3d	Number of blocks in a grid along the X, Y and Z dimensions for a kernel launch. This option outputs the following three columns: CUDA: • gridsizeX • gridsizeY • gridsizeZ OpenCL: • ndrangesizeX • ndrangesizeY • ndrangesizeZ

threadblocksize	Number of threads in a block along the X, Y and Z dimensions for a kernel launch. This option outputs the following three columns: CUDA: • threadblocksizeX • threadblocksizeY • threadblocksizeZ OpenCL: • workgroupsizeX • workgroupsizeY • workgroupsizeZ
dynsmemperblock	Size of dynamically allocated shared memory per block in bytes for a kernel launch. (Only CUDA)

Option	Description
stasmemperblock	Size of statically allocated shared memory per block in bytes for a kernel launch. This option outputs the following columns: CUDA: • stasmemperblock OpenCL: • stasmemperworkgroup
regperthread	Number of registers used per thread for a kernel launch. This option outputs the following columns: CUDA: • regperthread OpenCL: • regperworkitem
memtransferdir	Memory transfer direction, a direction value of 0 is used for host to device memory copies and a value of 1 is used for device to host memory copies.
memtransfersize	Memory transfer size in bytes. This option shows the amount of memory transferred between source (host/device) to destination (host/device).
memtransferhostmem type	Host memory type (pageable or page-locked). This option implies whether during a memory transfer, the host memory type is pageable or page-locked.
streamid	Stream Id for a kernel launch.
localblocksize	If workgroupsize has been specified by the user, this option would be 1, otherwise it would be 0.(Only OpenCL). This option outputs the following column: • localworkgroupsize

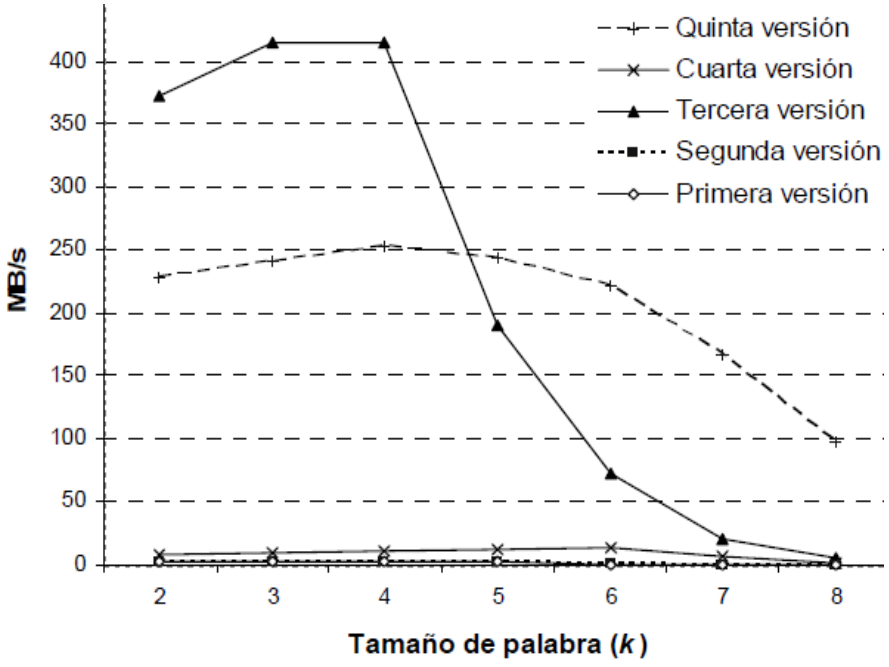
cacheconfigrequested	Requested cache configuration option for a kernel launch: • 0 CU_FUNC_CACHE_PREFER_NONE - no preference for shared memory or L1 (default) • 1 CU_FUNC_CACHE_PREFER_SHARED - prefer larger shared memory and smaller L1 cache • 2 CU_FUNC_CACHE_PREFER_L1 - prefer larger L1 cache and smaller shared memory • 3 CU_FUNC_CACHE_PREFER_EQUAL - prefer equal sized L1 cache and shared memory
cacheconfigexecuted	Cache configuration which was used for the kernel launch. The values are same as those listed under cacheconfigrequested.

Command Line Profiler Counter Name	Description	Type SM= Single Multiprocessor FB = Frame Buffer (GPU DRAM or Device Memory)	Compute Capability Support Y= Yes N= No					
			1.0	1.1	1.2	1.3	2.0	2.1
inst_issued	Number of instructions issued including replays.	SM	N	N	N	N	Y	N
inst_issued1_0	Number of cycles that issue one instruction for instruction pipeline 0	SM	N	N	N	N	N	Y
inst_issued2_0	Number of cycles that issue two instructions for instruction pipeline 0	SM	N	N	N	N	N	Y
inst_issued1_1	Number of cycles that issue one instruction for instruction pipeline 1	SM	N	N	N	N	N	Y
inst_issued2_1	Number of cycles that issue two instructions for instruction pipeline 1	SM	N	N	N	N	N	Y
thread_inst_executed_0	Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 0.	SM	N	N	N	N	Y	Y
thread_inst_executed_1	Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 1.	SM	N	N	N	N	Y	Y
thread_inst_executed_2	Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 2.	SM	N	N	N	N	Y	Y

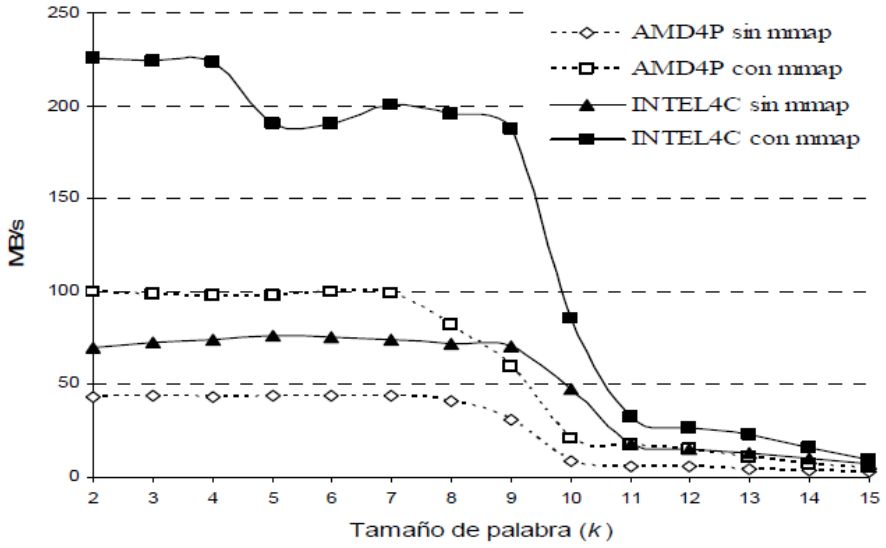
Command Line Profiler Counter Name	Description	Type SM= Single Multiprocessor FB = Frame Buffer (GPU DRAM or Device Memory)	Compute Capability Support Y= Yes N= No					
			1.0	1.1	1.2	1.3	2.0	2.1
thread_inst_executed_3	Number of instructions executed by all threads. This does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 3.	SM	N	N	N	N	Y	Y
l2_subp0_read_sector_queries	Accumulated read sector queries from L1 to L2 cache for slice 0 of all the L2 cache units	FB	N	N	N	N	Y	Y
l2_subp1_read_sector_queries	Accumulated read sector queries from L1 to L2 cache for slice 1 of all the L2 cache units	FB	N	N	N	N	Y	Y*
l2_subp0_read_tex_sector_queries	Accumulated read sector queries from texture cache to L2 cache for slice 0 of all the L2 cache units	FB	N	N	N	N	Y	Y
l2_subp1_read_tex_sector_queries	Accumulated read sector queries from texture cache to L2 cache for slice 1 of all the L2 cache units	FB	N	N	N	N	Y	Y*
l2_subp0_write_sector_queries	Accumulated write sector queries from L1 to L2 cache for slice 0 of all the L2 cache units	FB	N	N	N	N	Y	Y
l2_subp1_write_sector_queries	Accumulated write sector queries from L1 to L2 cache for slice 1 of all the L2 cache units	FB	N	N	N	N	Y	Y*
l2_subp0_read_sector_misses	Accumulated read sectors misses from L2 cache for slice 0 for all the L2 cache units	FB	N	N	N	N	Y	Y
l2_subp1_read_sector_misses	Accumulated read sectors misses from L2 cache for slice 1 for all the L2 cache units	FB	N	N	N	N	Y	Y*
l2_subp0_write_sector_misses	Accumulated write sector misses from L2 cache for slice 0 for all the L2 cache units	FB	N	N	N	N	Y	Y
l2_subp1_write_sector_misses	Accumulated write sectors misses from L2 cache for slice 1 for all the L2 cache units	FB	N	N	N	N	Y	Y*
fb_subp0_read_sectors	Number of read requests sent to sub-partition 0 of all the DRAM units	FB	N	N	N	N	Y	Y
fb_subp1_read_sectors	Number of read requests sent to sub-partition 1 of all the DRAM units	FB	N	N	N	N	Y	Y
fb0_subp0_read_sectors	Number of read requests sent to sub-partition 0 of DRAM unit 0	FB	N	N	N	N	Y	Y
fb0_subp1_read_sectors	Number of read requests sent to sub-partition 1 of DRAM unit 0	FB	N	N	N	N	Y	Y
fb1_subp0_read_sectors	Number of read requests sent to sub-partition 0 of DRAM unit 1	FB	N	N	N	N	Y	Y
fb1_subp1_read_sectors	Number of read requests sent to sub-partition 1 of DRAM unit 1	FB	N	N	N	N	Y	Y

Command Line Profiler Counter Name	Description	Type SM= Single Multiprocessor FB = Frame Buffer (GPU DRAM or Device Memory)	Compute Capability Support Y= Yes N= No					
			1.0	1.1	1.2	1.3	2.0	2.1
fb_subp0_write_sectors	Number of write requests sent to sub-partition 0 of all the DRAM units	FB	N	N	N	N	Y	Y
fb_subp1_write_sectors	Number of read requests sent to sub-partition 1 of all the DRAM units	FB	N	N	N	N	Y	Y
fb0_subp0_write_sectors	Number of write requests sent to sub-partition 0 of DRAM unit 0	FB	N	N	N	N	N	Y*
fb0_subp1_write_sectors	Number of write requests sent to sub-partition 1 of DRAM unit 0	FB	N	N	N	N	N	Y*
fb1_subp0_write_sectors	Number of write requests sent to sub-partition 0 of DRAM unit 1	FB	N	N	N	N	N	Y*
fb1_subp1_write_sectors	Number of write requests sent to sub-partition 1 of DRAM unit 1	FB	N	N	N	N	N	Y*
tex0_cache_sector_queries	Number of texture cache sector queries for texture unit 0	SM	N	N	N	N	Y	Y
tex1_cache_sector_queries	Number of texture cache sector queries for texture unit 1	SM	N	N	N	N	N	Y
tex0_cache_sector_misses	Number of texture cache sector misses for texture unit 0	SM	N	N	N	N	Y	Y
tex1_cache_sector_misses	Number of texture cache sector misses for texture unit 1	SM	N	N	N	N	N	Y

Anexo 2 Tablas de comparación en la aplicación “Análisis paralelo de secuencias de ADN mediante el uso de GPU y CUDA”



En la GPU.


AMD4P sin mmap
AMD4P con mmap
INTEL4C sin mmap
INTEL4C con mmap
MB/s
250
200
150
100
50
0
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Tamaño de palabra (k)

En la CPU.