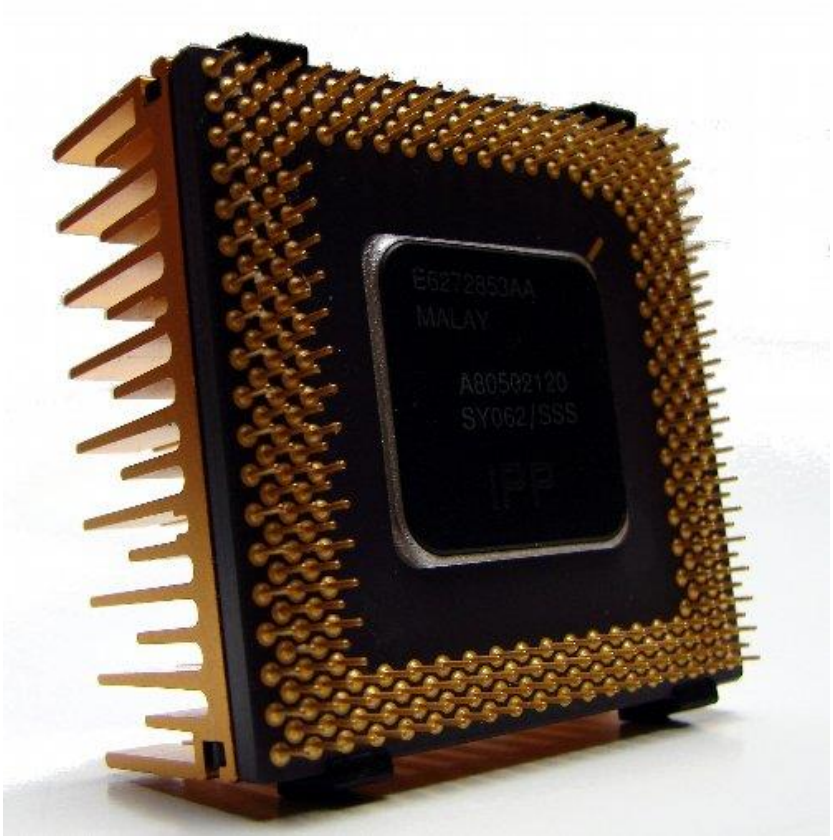
MATERIAL ADICIONAL F-IV

- **Desarrollo de los ejercicios en grupo**
- **Computhon**

PRÁCTICA 1



2017

Proyecto de programación con GPGPU

F-IV. Implementación de una rutina con CUDA para la aceleración de algoritmos utilizando GPGPU

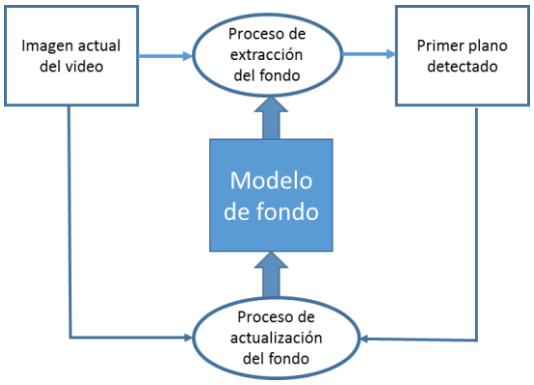
Arquitectura de los Computadores
Grado en Ingeniería Informática
Dpto. Tecnología Informática y Computación
Universidad de Alicante

MATERIAL ADICIONAL F-IV (2)

PROYECTO DE ACELERACIÓN DE ALGORITMOS CON GPGPU

I. EXTRACCIÓN DE LA REGIÓN DE INTERÉS EN UNA SECUENCIA DE IMÁGENES (BACKGROUND SUBSTRACTION)

El proceso de extracción de fondo (background subtraction) es una etapa importante en muchas aplicaciones de sistemas de visión. Consiste en diferenciar los pixeles de primer plano de la escena de fondo en una secuencia de imágenes y utilizar los componentes de interés en etapas de procesamiento posteriores. El resultado de este proceso puede ser una imagen binaria que contiene solo el primer plano o una imagen de probabilidad que contiene la probabilidad de que cada pixel pertenezca al primer plano. Una vez se ha obtenido el fondo se debe mantener el mismo para adaptar el modelo a los cambios que puedan presentarse en la escena a través del tiempo. En la figura se muestra el sistema completo de extracción del fondo.



La técnica más sencilla para extraer el fondo consiste en la diferencia de imágenes del video como método de clasificación. A partir de las secuencias de imágenes de un vídeo se establece la escena de fondo B(x,y,t), si la imagen del video de entrada en el instante t es I(x,y,t) se pueden extraer los objetos de primer plano mediante aplicando un umbral a la diferencia absoluta de las dos imágenes:

$$\left|I(x,y,t) - B(x,y,t)\right| > Th$$

Aquellos pixeles cuyos valores de intensidad son superiores al umbral se considerarán como pertenecientes al primer plano, el resto de píxeles pertenecerán al fondo.

Un problema de este algoritmo es que el umbral *Th* es el mismo para todos los pixeles de la imagen con lo que no resuelve bien cambios en la iluminación ni la presencia de sombras.

En el caso de imágenes de color se puede establecer un umbral distinto para cada color de canal.

Para el caso de entradas con imágenes en escala de grises se puede adoptar una solución sencilla que mejora el algoritmo anterior y es considerar un umbral distinto para cada pixel. En esta aproximación se calcula un umbral local para cada pixel examinando el valor de las intensidades de sus vecinos. El umbral local se calcula como el valor medio de la distribución de intensidades locales utilizando un filtro de media que opera sobre una ventana 3x3 que selecciona la media de las intensidades de la ventana. Si el valor del pixel está por debajo del umbral se le considera fondo y en otro caso formará parte del primer plano. Puede todavía mejorarse más el resultado si en lugar de considerarse como umbral la media se considera la *media* –C, donde C es una constante a definir.

Para simplificar el sistema se considerará el fondo invariable a través del tiempo y se establecerá la imagen de fondo B(xy) como la primera de la secuencia de imágenes a tratar.

Entrada y salida del programa

Se establecerá como entrada al problema una secuencia de imágenes que forman parte de un vídeo estableciéndose como imagen de fondo la primera de la secuencia. Como salida se deben obtener una secuencia de imágenes en la que se muestren los objetos de primer plano mediante una imagen binaria en la que el fondo aparezca negro y blanco el primer plano.

$$R(x,y)=\begin{cases} Blanco & |f(x,y,t) - B(x,y)| > (T_{x,y} - C) \\ Negro & en\ otro\ caso \end{cases}$$

II. OPERACIÓN DE DILATACIÓN MORFOLÓGICA SOBRE UNA IMAGEN

El problema consiste en implementar en CUDA una versión simplificada de dilatación morfológica sobre una imagen en escala de grises.

La morfología matemática es una teoría y técnica para el análisis y tratamiento de las estructuras geométricas, basada en la teoría de conjuntos, teoría de retículos, topología y funciones aleatorias. La morfología matemática es comúnmente aplicada más a las imágenes digitales, pero puede ser empleada también en gráficos, mallas poligonales, sólidos y otras estructuras espaciales.

Dilatación morfológica en escala de grises

Sean f y b funciones de Z x Z → Z. La dilatación de una imagen de niveles de gris f(x,y) por un elemento estructural b(x,y) se define como:

$$(f \oplus b)(s,t) = \max\{f(s-x,t-y)+b(x,y) \ \text{ tal que } (s-x,t-y) \in D_f,\ (x,y) \in D_b\}$$

donde Df y Db son los dominios de f y b respectivamente.

Nótese que el elemento estructural, b, es una función entera. Los elementos estructurales más sencillos son aquéllos que vienen dados por la función constante, b(x,y)=0, llamados elementos estructurales “planos”. En ese caso,

$$(f \oplus b)(s,t) = \max\{f(s-x,t-y) \ \text{ tal que } (s-x,t-y) \in D_f,\ (x,y) \in D_b\}$$

Y entonces la dilatación consiste en hacer un filtro de máximo con la máscara dada por (suponiendo que el elemento estructural es simétrico respecto del origen).

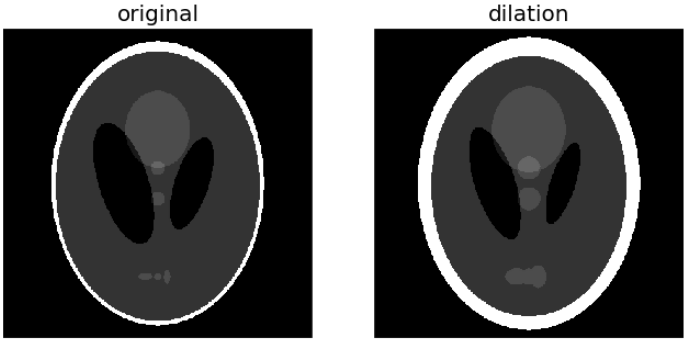
Por ejemplo, si se tiene un elemento estructurante cuadrado de lado 3, cada pixel toma el valor máximo de la ventana 3x3 alrededor del mismo.

3	28	5	4	6
14	21	22	14	2
12	12	14	33	56
2	3	5	45	23
11	23	45	22	34

28	28	28	22	14
28	28	28	56	56
21	22	45	56	56
23	45	45	56	56
23	45	45	45	45

NOTA: Si la ventana alrededor del pixel se sale de los bordes de la imagen, únicamente se consideran los puntos dentro de la imagen que se encuentran dentro de la ventana.

El resultado de aplicación de una dilatación a una imagen en niveles de gris es incrementar los detalles claros, reduciendo o eliminando los detalles más oscuros.



Entrada y salida de fichero

El código se llamará pasando dos parámetros, uno con el nombre de la imagen y otro con el tamaño del elemento estructurante. Por ejemplo, desde la ventana de comandos se podrá ejecutar como

```
cudaDilatacion.exe baboon.bmp 3
```

El tamaño del elemento estructurante indica cuantos pixels en cada dirección alrededor del pixel siendo procesado son considerados. Es decir, dado un tamaño SIZE para el elemento estructurante, la ventana será de un tamaño 2*SIZE+1.

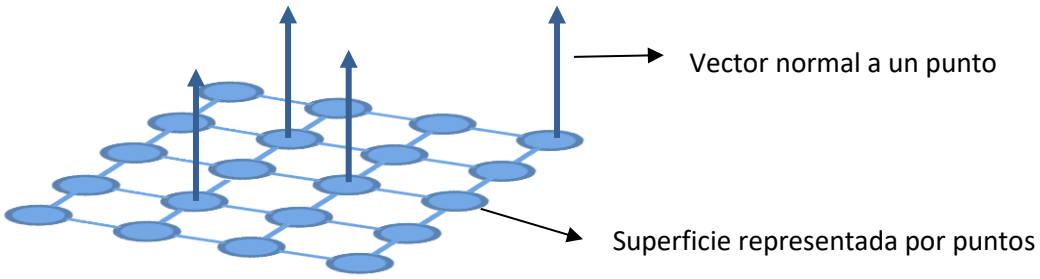
III. CÁLCULO DE LAS NORMALES PARA UNA MALLA 3D

Introducción

El problema consiste en implementar en CUDA la obtención de los vectores normales correspondientes a un conjunto amplio de puntos 3D en función de los vecinos más cercanos. El cálculo de normales para una superficie tridimensional es un aspecto muy importante en el campo de los gráficos por computador, visión artificial y robótica, entre otros.

Cálculo de las normales de una superficie

El cálculo de las normales de una superficie puede ser abordado desde diferentes puntos de vista. En el ejercicio que se presenta a continuación, se propone obtener las normales correspondientes a una superficie que está representada por un conjunto de puntos 3D. Para cada uno de los puntos de la superficie, se calculará el vector normal correspondiente utilizando los puntos de la vecindad del mismo.



Se considera que los puntos de la superficie están organizados en una estructura de malla que permite determinar la vecindad de un punto dado. Para el cálculo de la normal en ese punto, se establece que será el vector normal medio de las normales de las rectas que pueden establecerse entre el punto y cada uno de los puntos de su vecindad. El algoritmo podría ser el siguiente:

```
para toda P de los puntos de la superficie
     vectorMedio(P)=(0,0,0);
     para todo punto Q de la vecindad de P
         recta=CalcularRecta(P,Q);
         vectorMedio=vectorMedio+CalculaNormal(recta);
     fin para
     vectorMedio(P)=vectorMedio(P)/numPuntosVecindad;
fin para
```

Funciones

El algoritmo que resuelve el cálculo de las normales se aporta, en su versión secuencial, en la función *CalculaNormalesCPU* dentro del fichero *calculaNormales.cu*. Se plantea implementar la función *CalculaNormalesGPU*. Esta función debe ser funcionalmente equivalente a la función CPU del mismo nombre, realizando operaciones equivalentes para determinar el vector normal que corresponde a un conjunto de puntos de entrada correspondientes a una superficie. En esta función deberá usarse la programación basada en CUDA® para aprovechar la capacidad de cómputo paralelo de las GPUs de NVIDIA®.

De forma análoga a la función CPU, el resultado de la función deberá devolver en los vectores *NormalUGPU, NormalVPU, NormalWGPU* los componentes de los vectores normales normalizados (entre 0 y 1) de cada uno de los puntos de la superficie.

Estos vectores serán posteriormente comparados en la función *runTest* (ya implementada) con el generado por la solución CPU para comprobar su corrección. Los tiempos que serán tenidos en cuenta para la competición por equipos serán exclusivamente los generados para la solución GPU y que muestra la función *runTest* por pantalla.

Función runTest

La función principal del programa tiene como objetivo, leer un fichero con la información de la superficie que se pasará como argumento y realizar el cálculo de las normales propuesta en el apartado anterior.

Una vez leída la información de la superficie, mediante las función *LeerSuperficie* (ya implementada) se lanzan dos algoritmos, uno basado en CPU-Secuencial (sin llamadas a CUDA) y que es aportado en la práctica y otro basado en GPU que debe ser implementado por el grupo. El tiempo de ejecución es obtenido para cada uno de ellos y, tras la conclusión de ambos, se comprueba si la solución generada es idéntica. En tal caso, el tiempo de GPU será tenido en cuenta a efectos de la competición.

Funciones auxiliares

LeerSuperficie. Implementada en calculaNormales.cu

Esta función lee un fichero (.sup) que es aportado en la práctica y construye una malla a partir de los puntos de la superficie. Esta malla se organiza en la estructura TSurf S y contiene:

- `UPoints` → `Número de puntos en la dirección U`
- `VPoints` → `Número de puntos en la dirección V`
- `Buffer`→ `Array bidimensional de puntos TPoints3D de tamaño VPoints*UPoints. Ejemplo, para acceder al punto de coordenadas (v,u), se accederá mediante el código S.Buffer[v][u]`

CrearSuperficie. Implementada en calculoNormales.h. Gestiona la creación de la malla.

BorrarSuperficie Implementada en calculoNormales.h. Gestiona la liberación de memoria de la malla.

Funciones de simulación

CalculaNormalesCPU. Implementada en *calculaNormales.cu*

Para calcular los vectores normales, implementa el algoritmo comentado anteriormente. Para mayor información, la función proporcionada comenta paso a paso cada línea de código.

El resultado final de la función es el cálculo de las normles como vector global (ya creado en la función principal) que tienen el tamaño del número de puntos a calcular. Estos vectores *NormalUCPU, NormalVCPU* y *NormalWCPU* contienen para cada posición, el componente de la normal asociado al punto.

CalculaNormalesGPU. NO implementada en *calculaNormales.cu*

Esta es la función que tiene que ser implementada por el grupo. El objetivo es que sea funcionalmente equivalente a la descrita anteriormente, con la diferencia de que los resultados se devolverán en los vectores globales (ya creados en la función principal) *NormalUGPU, NormalVGPU* y *NormalWGPU*. Como ya se ha dicho, este vector será comparado a efectos de corrección en la función *runTest*.

La función debe incluir la gestión de memoria para el paso de datos desde la CPU a la GPU y viceversa, así como la llamada a una o varias funciones kernel de CUDA que efectúen la paralelización de las

operaciones siguiendo el paradigma SIMD usando la GPU del computador. Esta función o funciones kernel deberán ser definidas por el grupo y se da libertad sobre su contenido y definición.

Entrada de fichero

Se proporcionará la lectura de un fichero que contendrá la estructura de la superficie:

puntos.sup

```
Alto: 1000
Ancho: 1000
Dimensión: 3
45.6 76.3 78.7
45.8 65.0 56.0
34.2 78.3 78.0
78.3 87.8 56.3
45.8 65.0 56.0
45.8 67.0 56.0
```

Finalmente, la salida se escribirá en fichero y contendrá los componentes del vector normal a cada punto. Por ejemplo:

A fin de que la comparativa entre grupos sea lo más justa posible, no será posible salvo causa justificada expresamente, modificar ninguna de las funciones ya implementadas en la práctica.

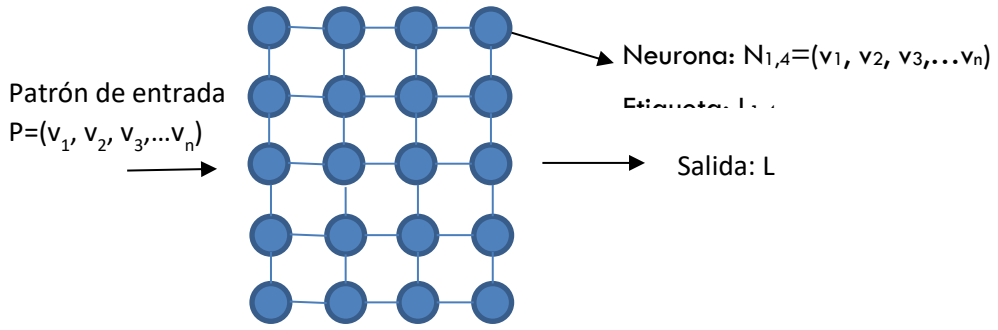
IV. CLASIFICADOR PARALELO BASADO EN SOM

Introducción

El problema consiste en implementar en CUDA una versión simplificada de un mapa auto-organizativo (Self Organizing Map). Se trata de una red neuronal que permite la clasificación de diferentes patrones de entrada. La red consta de dos modos: el modo entrenamiento que permite configurar la red y un modo de clasificación, que utiliza la configuración del entrenamiento para clasificar nuevos patrones de entrada. El problema se centrará exclusivamente en la fase de clasificación.

Estructura del SOM

El SOM consiste en una colección de neuronas (nodos) que contiene un vector de pesos de la misma dimensión que la de los patrones de entrada. Además cada neurona contendrá una etiqueta que será la salida del clasificador. Las neuronas estarán conectadas disponiendo una rejilla rectangular (grid).



Por ejemplo en el gráfico anterior, está representado un SOM de 5x4 neuronas con una vecindad de 4 neuronas (las marcadas por las aristas a cada neurona) excepto aquellas que están en los extremos del mapa.

Clasificación

El SOM original compara el patrón de entrada con cada una de las neuronas del mapa para establecer qué neurona es la ganadora, asociando la salida del mapa a la etiqueta que contenga dicha neurona. La neurona ganadora es aquella, cuya distancia euclídea entre el patrón de entrada y los pesos de la neurona es la menor en el mapa. En este caso, la implementación contemplará también la vecindad de una neurona. La neurona ganadora será aquella cuya suma de distancias euclídeas entre el patrón y las neurona de su vecindad (incluyendo a ella misma) sea la menor en el mapa:

```
para toda N de las neuronas del mapa
     distancia(N) = euclidea (N,P);
     para toda neurona V de la vecindad de N
          distancia(N) = distancia(N) + euclidea(V,P)
     fin para
fin para
ganadora = min(distancia)
```

Funciones

El algoritmo que resuelve la clasificación se aporta, en su versión secuencial, en la función *ClasificacionSOMCPU* dentro del fichero clasificacionSOM.*cu*. Se plantea implementar la función *ClasificacionSOMGPU*. Esta función debe ser funcionalmente equivalente a la función CPU del mismo nombre, realizando operaciones equivalentes para determinar la etiqueta que corresponde a un patrón de entrada. En esta función deberá usarse la programación basada en CUDA® para aprovechar la capacidad de cómputo paralelo de las GPUs de NVIDIA®.

De forma análoga a la función CPU, el resultado de la función deberá devolver en el vector *EtiquetaGPU*, la etiqueta asociada al vector de patrones de entrada *Patrones*.

Este vector será posteriormente comparado en la función *runTest* (ya implementada) con el generado por la solución CPU para comprobar su corrección. Los tiempos que serán tenidos en cuenta para la competición por equipos serán exclusivamente los generados para la solución GPU y que muestra la función *runTest* por pantalla.

Función runTest

La función principal del programa tiene como objetivo, leer un fichero con la información del SOM y otro con los patrones de entrada a clasificar que se pasan como argumentos y realizar la clasificación propuesta en los apartados anteriores.

Una vez leída la información del SOM y de los patrones, mediante las funciones *LeerSOM* y LeerPatrones (ya implementadas) se lanzan dos algoritmos, uno basado en CPU-Secuencial (sin llamadas a CUDA) y que es aportado en la práctica y otro basado en GPU que debe ser implementado por el grupo. El tiempo de ejecución es obtenido para cada uno de ellos y, tras la conclusión de ambos, se comprueba si la solución generada es idéntica. En tal caso, el tiempo de GPU será tenido en cuenta a efectos de la competición.

Funciones auxiliares

LeerSOM. Implementada en clasificacionSOM.cu

Esta función lee un fichero (.som) que es aportado en la práctica y construye un mapa SOM. El mapa se organiza en la estructura TSOM SOM y TNeurona Neurona. El primero contiene:

- `Ancho` → `Ancho del mapa`
- `Alto` → `Alto del mapa`
- `Dimension` → `Dimensión de cada neurona del mapa`
- `Neurona`→ `Array bidimensional de TNeurona de tamaño Ancho*Alto. Ejemplo, para acceder a la neurona de coordenadas (v,u), se accederá mediante el código SOM.Neurona[v][u]`

La estructura TNeurona Neurona contiene los pesos de cada neurona del mapa y la etiqueta asociada.

LeerPatrones. Implementada en clasificacionSOM.cu

Esta función lee un fichero (.pat) que es aportado en la práctica y construye los patrones de entrada. Los patrones se organizan en la estructura TPatrones Patrones:

- `Cantidad` → `Números de patrones`
- `Dimension` → `Dimensión de cada patrón`
- `Pesos` → `Array que contiene los patrones de tamaño Cantidad*Dimension. Ejemplo, para acceder al peso u del patrón v, se accederá mediante el código Patrones.Pesos[v][u]`

CrearMapa. Implementada en clasificacionSOM.h. Gestiona la creación del mapa.

BorrarMapa Implementada en clasificacionSOM.h. Gestiona la liberación de memoria del mapa.

Funciones de simulación

ClasificacionSOMCPU. Implementada en *clasificacionSOM.cu*

Para clasificar el conjunto de patrones de entrada, implementa el algoritmo comentado anteriormente. Para mayor información, la función proporcionada comenta paso a paso cada línea de código.

El resultado final de la función es el cálculo de las etiquetas como vector global (ya creado en la función principal) que tienen el tamaño del número de patrones a clasificar. Este vector *EtiquetaCPU* contiene para cada posición, la etiqueta correspondiente al patrón de entrada.

ClasificacionSOMGPU. NO implementada en *clasificacionSOM.cu*

Esta es la función que tiene que ser implementada por el grupo. El objetivo es que sea funcionalmente equivalente a la descrita anteriormente, con la diferencia de que los resultados se devolverán en el vector global (ya creados en la función principal) *EtiquetaGPU*. Como ya se ha dicho, este vector será comparado a efectos de corrección en la función *runTest*.

La función debe incluir la gestión de memoria para el paso de datos desde la CPU a la GPU y viceversa, así como la llamada a una o varias funciones kernel de CUDA que efectúen la paralelización de las operaciones siguiendo el paradigma SIMD usando la GPU del computador. Esta función o funciones kernel deberán ser definidas por el grupo y se da libertad sobre su contenido y definición.

Entrada de ficheros

Se proporcionará la lectura de dos ficheros. Uno contendrá la estructura del mapa en el que se indicará el alto y ancho del mapa así como la dimensión de los pesos de cada neurona. Por ejemplo, para un mapa 3x2 con un vector de pesos de dimensión 3:

Estructura.som

```
Alto: 3
Ancho: 2
Dimensión: 3
N1,1: 45.6 76.3 78.7
L1,1: 1
N1,2: 45.8 65.0 56.0
L1,2: 2
N2,1: 34.2 78.3 78.0
L2,1: 3
N2,2: 78.3 87.8 56.3
L2,3: 4
N3,1: 45.8 65.0 56.0
L3,1: 1
N3,2: 45.8 67.0 56.0
L3,3: 1
```

El otro fichero de entrada, contendrá los patrones de entrada. Se indicará el número de patrones y la dimensión. Por ejemplo:

patrones.pat

```
Numero: 5
Dimension: 3
P1: 78.0 67.7 37.7
P2: 74.0 67.5 47.7
P3: 78.0 67.1 27.7
P4: 72.0 67.3 17.7
P5: 79.0 67.5 37.7
```

A fin de que la comparativa entre grupos sea lo más justa posible, no será posible salvo causa justificada expresamente, modificar ninguna de las funciones ya implementadas en la práctica.

V. SIMULACIÓN DEL MOVIMIENTO DE UNA HERRAMIENTA SOBRE UN TORNO DE MECANIZADO

Introducción

El diseño y fabricación asistidos por computador (CAD/CAM) es uno de los campos en donde se están aplicando con mayor éxito las técnicas de aceleración de algoritmos mediante unidades de procesamiento gráfico (GPUs). Esto es debido a que la mayoría de estos algoritmos tienen una naturaleza SIMD, realizando operaciones repetitivas sobre una gran cantidad de datos tridimensionales y con formato de doble precisión.

Tornos de mecanizado

En el ejercicio que se presenta a continuación, se propone un ejemplo de algoritmo geométrico propio del CAM en el que se realiza una simulación del movimiento de una herramienta sobre una superficie que se coloca en una máquina tipo torno. En este tipo de máquinas los objetos se colocan sobre un eje de rotación que gira a velocidad constante y en su giro, una herramienta se desplaza linealmente siguiendo el mismo eje (ver figura 1.a). El movimiento compuesto de ambos elementos (giro de la superficie y posicionamiento de la herramienta) produce un helicoide (una especie de espiral en tres dimensiones, como la que se puede observar en el surco de cualquier tornillo, ver figura 1.b).

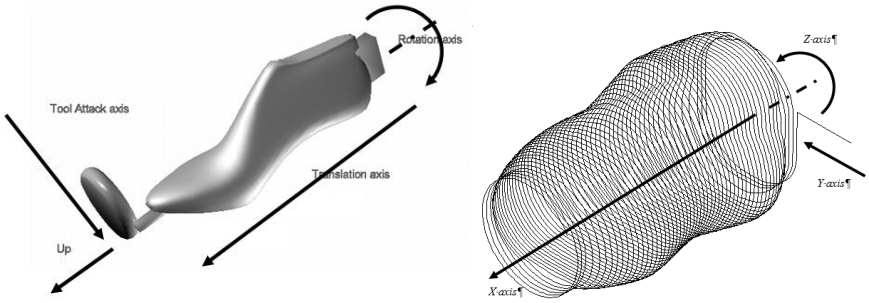


Figura 1. a) Izquierda. Posicionamiento de superficie sobre un torno de mecanizado. b) Derecha. Movimiento compuesto que efectúa la herramienta sobre la superficie.

Objetivo

Así pues, este ejercicio está diseñado para simular el movimiento helicoidal que la herramienta describe sobre una superficie 3D. El algoritmo que resuelve esta simulación se aporta, en su versión secuencial, en la función *SimulacionTornoCPU* dentro del fichero *simutorno.cu*.

Así pues, el objetivo de esta práctica consiste en paralelizar la función anterior, usando para ello, la arquitectura SIMD que proporciona la arquitectura de las tarjetas NVIDIA® usando el lenguaje CUDA® de forma similar a como se planteaba la resolución de los ejercicios individuales.

Ejercicio grupal

Tal y como se ha mencionado, el ejercicio consistirá en implementar la función *SimulacionTornoGPU*. Esta función debe ser funcionalmente equivalente a la función CPU del mismo nombre, realizando operaciones equivalentes sobre la superficie a fin de proporcionar, en cada paso de simulación, las posiciones de los puntos (coordenadas v y u de la malla) más cercanas a la herramienta del torno de mecanizado.

En esta función deberá usarse la programación basada en CUDA® para aprovechar la capacidad de cómputo paralelo de las GPUs de NVIDIA®.

De forma análoga a la función CPU, el resultado de la función deberá devolver en los vectores *GanadorUGPU* y *GanadorVGPU* las posiciones U y V respectivamente de los puntos más cercanos a la herramienta del torno en cada paso de simulación.

Estos vectores serán posteriormente comparados en la función *runTest* (ya implementada) con los generados por la solución CPU para comprobar su corrección. Los tiempos que serán tenidos en cuenta para la competición por equipos serán exclusivamente los generados para la solución GPU y que muestra la función *runTest* por pantalla.

Función runTest

La función principal del programa tiene como objetivo, leer una superficie almacenada como una malla de puntos 3D de un fichero que se pasa como argumento y realizar sobre ella una simulación de movimiento helicoidal alrededor del eje X, siguiendo unos pasos angulares y lineales fijos.

Una vez leída la información de la superficie, mediante la función *LeerSuperficie* (ya implementada) se lanzan dos algoritmos, uno basado en CPU-Secuencial (sin llamadas a CUDA) y que es aportado en la práctica y otro basado en GPU que debe ser implementado por el grupo. El tiempo de ejecución es obtenido para cada uno de ellos y, tras la conclusión de ambos, se comprueba si la solución generada es idéntica. En tal caso, el tiempo de GPU será tenido en cuenta a efectos de la competición.

Funciones auxiliares

LeerSuperficie. Implementada en simutorno.cu

Esta función leer un fichero geométrico (.for) que es aportado en la práctica y construye una malla tridimensional de puntos. Esta malla se organiza en la estructura TSurf S y contiene:

- `UPoints` → Número de puntos en la dirección U
- `VPoints` → Número de puntos en la dirección V
- `Buffer`→ Array bidimensional de puntos TPoints3D de tamaño VPoints*UPoints. Ejemplo, para acceder al punto de coordenadas (v,u), se accederá mediante el código `S.Buffer[v][u]`

Respecto al fichero .for, se trata de un fichero de texto que contiene información de la geometría de una superficie correspondiente a una horma de calzado y también del helicoide de la simulación. En concreto:

- `"SECTION NUMBER"`, contiene el tamaño de la malla en la dirección U
- `"POINTS PER SECTION"`, contiene el tamaño de la malla en la dirección V
- `"STEP"`, contiene el paso de la helicoide (milímetros que avanza la herramienta a cada vuelta completa sobre el eje de giro X y que se almacena en la variable global "PasoHelicoide"
- `"POINTS PER ROUND"`, contiene el número de giros que compone cada vuelta en la helicoide y que se almacena en la variable global "PuntosVueltaHelicoide"

CrearSuperficie. Implementada en simutorno.h. Gestiona la creación de memoria de la malla.

BorrarSuperficie Implementada en simutorno.h. Gestiona la liberación de memoria de la malla.

Funciones de simulación

SimulacionTornoCPU. Implementada en *simutorno.cu*

Para simular un movimiento helicoidal sobre el eje x, se realizan, para cada paso de simulación, dos tipos de movimientos sobre todos los puntos de la superficie: un movimiento de rotación sobre el eje X, que afecta por tanto a las coordenadas Y y Z de cada punto, y un movimiento de traslación sobre el eje X que afectaría sólo a la coordenada X de cada punto. El primero se realiza mediante la composición de operaciones seno, coseno y el segundo como una simple suma.

Una vez completado un paso de simulación (el giro/traslación de todos los puntos) se comprueba qué punto es el más cercano a la situación de la herramienta dentro de su área de influencia. Para ello, se comprueba que la coordenada x del punto transformado esté frente a la herramienta, lo que se determina simplemente comprobando que *-TOOLWIDTH<px< +TOOLWIDTH*. Si se cumple esa condición, entonces se determina la distancia del punto a la herramienta mediante la longitud del vector que el punto define respecto al punto *(0,TOOLYPOS,0)*.

Para mayor información, la función proporcionada comenta paso a paso cada línea de código.

El resultado final de la función es el cálculo de dos vectores globales (ya creados en la función principal) que tienen el tamaño del número de pasos de simulación. Estos vectores *GanadorVCPU* y *GanadorUCPU* contienen las posiciones V y U respectivamente de los puntos que han resultado más cercanos a la herramienta para cada paso de simulación.

SimulacionTornoGPU. NO implementada en *simutorno.cu*

Esta es la función que tiene que ser implementada por el grupo. El objetivo es que sea funcionalmente equivalente a la descrita anteriormente, con la diferencia de que los resultados se devolverán en los vectores globales (ya creados en la función principal) *GanadorVGPU* y *GanadorUGPU*. Como ya se ha dicho, estos vectores serán comparados a efectos de corrección en la función *runTest*.

La función debe incluir la gestión de memoria para el paso de datos desde la CPU a la GPU y viceversa, así como la llamada a una o varias funciones kernel de CUDA que efectúen la paralelización de las operaciones siguiendo el paradigma SIMD usando la GPU del computador. Esta función o funciones kernel deberán ser definidas por el grupo y se da libertad sobre su contenido y definición.

A fin de que la comparativa entre grupos sea lo más justa posible, no será posible salvo causa justificada expresamente, modificar ninguna de las funciones ya implementadas en la práctica.