PROGRAMACION PARALELA

TRABAJO PRÁCTICO INTEGRADOR

TEMA

Optimización de Algoritmo de Canny con OpenMP, MPI y CUDA

DOCENTE

Dr. Wolfmann, Gustavo

ALUMNOS

Aguilar, Mauricio

Tarazi, Pedro Esequiel

CARRERA

Ingeniería en Computación

AÑO

2018

TABLA DE CONTENIDO

INTRODUCCIÓN

En el área de procesamiento de imágenes, la detección de los bordes de una imagen es de suma importancia y utilidad, pues facilita muchas tareas, entre ellas, el reconocimiento de objetos, la segmentación de regiones, entre otras. Se han desarrollado variedad de algoritmos que ayudan a solucionar este inconveniente.

El algoritmo de Canny es usado para detectar todos los bordes existentes en una imagen. Este algoritmo está considerado como uno de los mejores métodos de detección de contornos mediante el empleo de máscaras de convolución y basado en la primera derivada. Los puntos de contorno son como zonas de píxeles en las que existe un cambio brusco de nivel de gris. En el tratamiento de imágenes, se trabaja con píxeles, y en un ambiente discreto, es así que en el algoritmo de Canny se utiliza máscaras, las cuales representan aproximaciones en diferencias finitas.

En el presente trabajo se realizó la implementación del algoritmo de Canny con OpenMP, MPI y CUDA, a fin de estudiar en qué medida la programación paralela nos permite acelerar la ejecución de grandes volúmenes de datos.

ENUNCIADO

Resolver en paralelo el algoritmo de Canny para la detección de contornos de objetos en imágenes.

El código secuencial sobre el cual tienen que trabajar se encuentra en el Clúster, en la carpeta pp16, y es el archivo “canny.c”. Se compila con *gcc* incluyendo la librería matemática con la directiva de compilación -lm:

```
gcc -lm -o canny canny.c
```

Se debe trabajar con una imagen relativamente grande, de tipo “. pgm”, donde pueda apreciarse la ventaja del algoritmo paralelo. Para paralelizar, usar OpenMP, MPI y CUDA.

Se debe detectar primero cuales son los puntos del programa que insumen más cómputo cuya paralelización sea justificada

Links:

- http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm
- http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html
- https://www.youtube.com/watch?v=sRFM5IEqR2w

MARCO TEORICO

El algoritmo de Canny toma como entrada una imagen y tres parámetros, sigma, umbral bajo, umbral alto, y en base a esos datos entrega en su salida los bordes detectados en la imagen de entrada.

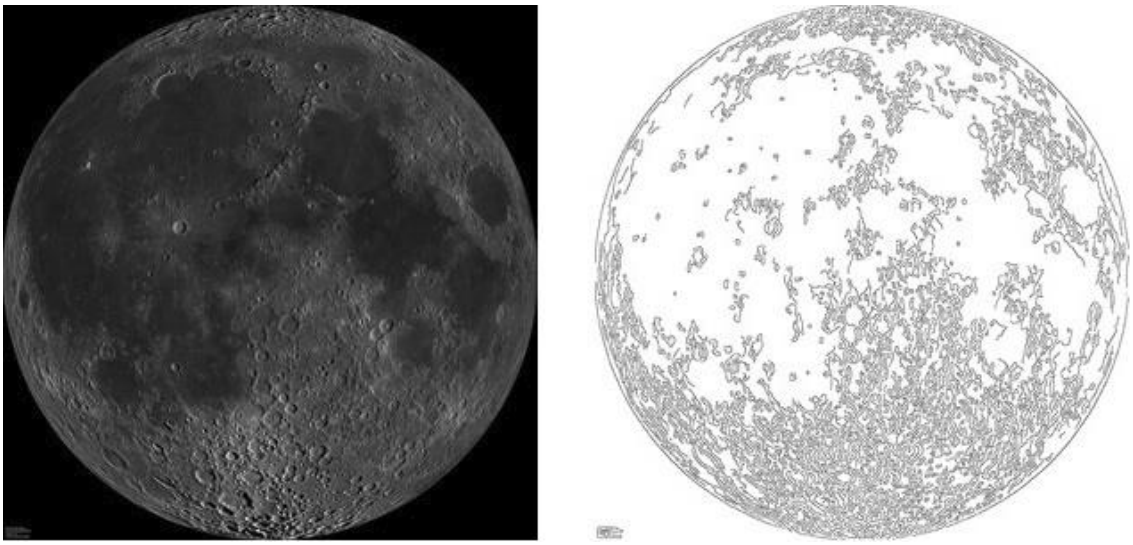


Figura 1 – Entrada (izquierda) -> Salida (derecha). Los parámetros utilizados fueron "2 0.33 0.66"

Las diferentes etapas de este algoritmo están definidas por las siguientes funciones:

1. **Gaussian_Smooth**: realiza un suavizado de la imagen, mediante la aplicación de un filtro gaussiano.
2. **Derivative_x_y**: Aquí se calcula la primera derivada de "X" e "Y" para cada pixel de la imagen.
3. **Radian_direction:** Calcula la dirección del gradiente de la imagen, a partir de la derivada de "X" e "Y". El resultado es un ángulo en radianes, en el sentido contrario a las agujas del reloj.
4. **Magnitude_x_y:** Calcula la magnitud del gradiente.
5. **Non_Max_Supp**: Suprime valores no máximos. Se genera una imagen con los bordes adelgazados.
6. **Apply_Hysteresis**: Elimina el ruido de la imagen, de forma tal que se obtiene la imagen con los bordes detectados.

DESARROLLO

Para la ejecución de estos programas se dispone de un nodo (Franky) de **32 CPUS** con placa CUDA **GeForce GTX 680**.

En primer lugar, se analizó el algoritmo de Canny y se detectó qué funciones consumen la mayor parte del tiempo de procesamiento mediante el uso de la herramienta GPROF. Para esto se utilizó el flag -pg en la orden de compilación. Los comandos fueron los siguientes:

```
gcc -fopenmp -o ./bin/canny-serial ./src/canny-serial.c -lm -pg
./bin/canny-serial ./pgm/imagen.pgm 2 0.33 0.66
gprof ./bin/canny-serial
```

Los tiempos obtenidos con y sin el flag -O3 se muestran en la Figura 1 y 2.

```
[pedrtara@franky TPF]$ gprof ./bin/canny-serial
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 68.21     37.35    37.35        1    37.35    37.36  gaussian_smooth
 12.10     43.98     6.62        1     6.62     6.62  non_max_supp
  6.89     47.75     3.77        1     3.77     5.14  apply_hysteresis
  5.72     50.88     3.13        1     3.13     3.13  derrivative_x_y
  4.62     53.41     2.53        1     2.53     2.53  magnitude_x_y
  2.50     54.78     1.37   189074     0.00     0.00  follow_edges
  0.02     54.79     0.01        1     0.01     0.01  make_gaussian_kernel
  0.00     54.79     0.00        1     0.00    54.79  canny
  0.00     54.79     0.00        1     0.00     0.00  read_pgm_image
  0.00     54.79     0.00        1     0.00     0.00  write_pgm_image

 %         the percentage of the total running time of the
time       program used by this function.
```

Figura 2 – Tiempos sin -O3

```
gcc -fopenmp -o ./bin/canny-serial ./src/canny-serial.c -lm -pg -O3
./bin/canny-serial ./pgm/imagen.pgm 2 0.33 0.66
gprof ./bin/canny-serial
```

```
[pedrtara@franky TPF]$ gprof ./bin/canny-serial
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 61.06     15.20    15.20        1    15.20    15.20  gaussian_smooth
 15.92     19.16     3.96        1     3.96     3.96  non_max_supp
  8.20     21.20     2.04        1     2.04     2.04  derrivative_x_y
  8.00     23.19     1.99        1     1.99     1.99  magnitude_x_y
  4.30     24.26     1.07        1     1.07     1.71  apply_hysteresis
  2.57     24.90     0.64   228376     0.00     0.00  follow_edges
  0.00     24.90     0.00        1     0.00    24.90  canny
  0.00     24.90     0.00        1     0.00     0.00  make_gaussian_kernel
  0.00     24.90     0.00        1     0.00     0.00  read_pgm_image

 %         the percentage of the total running time of the
time       program used by this function.
```

Figura 3 – Tiempos con -O3

Podemos observar que las funciones que más tiempo consumen son:

Gaussian_Smooth

Derivative_x_y

Magnitude_x_y

Non_Max_Supp

Apply_Hysteresis

De esta manera se decidió paralelizar estas cinco funciones tanto en OpenMP, MPI como en CUDA, a fin de hacer más concreta la comparación.

En segundo lugar se procedió a reescribir aquellas porciones de código que haciendo hincapié en la claridad, descuidaban la eficiencia. Un ejemplo típico es el de los bucles “for” anidados, los cuales pueden ser reescritos con un solo “for” a fin de minimizar la cantidad de saltos en el procesador.

// Código Original

```
for(r=0, pos=0; r<rows; r++){
        for(c=0; c<cols; c++, pos++){
            if(nms[pos] == POSSIBLE_EDGE) edge[pos] = POSSIBLE_EDGE;
            else edge[pos] = NOEDGE;
        }
```

```
}
// Código Optimizado

for(pos=0; pos<cols*rows; pos++) {
if(nms[pos] == POSSIBLE_EDGE) edge[pos] = POSSIBLE_EDGE;
else edge[pos] = NOEDGE;
}
```

Se realizó una optimización adicional, utilizando el flag -O3 para el compilador GCC, la cual realiza un unrolling de los "for" a fin de eliminar los saltos (con el consecuente uso de memoria adicional) y complementar así al punto anterior.

En tercer lugar, se procedió a la codificación de las cinco funciones antes mencionadas y se midieron sus tiempos para la ejecución serial, OpenMP, MPI, y CUDA, y de esta manera se los comparó y se calculó el speedup de cada uno.

OpenMP

Para esta librería, dado que se trata de un algoritmo que hace gran utilización del de bucles "for", para OpenMP se utilizó varias veces la directiva ***#pragma omp for***, la cual entrega un segmento de datos a cada uno de los hilos en ejecución. En la función **Non_Max_Sum** se utilizó la directiva **#pragma omp critical**, debido a que los hilos debían compartir ciertas variables inicializadoras en un bucle for, pero no debían ser modificadas durante la ejecución de la iteración.

MPI

Para paralelizar con este Framework se utilizó la mayoría de las veces las funciones:

- **MPI_Bcast**: Entrega los mismos datos a todos los hilos del grupo
- **MPI_AllGather**: Comparte y recibe los datos de todos los hilos.
- **MPI_AllReduce**: Comparte y recibe los datos de todos los hilos, y realiza la operación indicada en uno de sus argumentos.
- **MPI_Barrier**: Espera que todos los hilos terminen su ejecución para continuar.

Los tiempos de las funciones fueron medidos con **MPI_Wtime()**.

CUDA

Para programar en paralelo en arquitectura CUDA, se dispone de una placa **GeForce GTX 380**. Esta placa cuenta con **1536 cores** y **2048 MB** de memoria global de velocidad de 6.0 Gbps y se conecta el host por medio de un puerto **PCI Express 3.0**.

Para las funciones propuestas, se procedió a realizar dos kernels para cada una, focalizados en realizar la ejecución del trabajo pesado de estas.

Compilación y Ejecución

Finalmente, las órdenes de compilación en bash son:

```
gcc -fopenmp -o ./bin/canny-serial ./src/canny-serial.c $librerias
gcc -fopenmp -o ./bin/canny-openmp ./src/canny-openmp.c $librerias
mpicc -fopenmp -o ./bin/canny-mpi ./src/canny-mpi.c $librerias
nvcc -o ./bin/canny-cudaM ./src/canny-cudaM.cu $librerias
```

Donde:

librerias="-lm -O3"

Las órdenes de ejecución en bash son:

```
./bin/canny-serial ./pgm/$imagen.pgm $parametros
./bin/canny-openmp ./pgm/$imagen.pgm $parametros
mpirun -np $tareas ./bin/canny-mpi ./pgm/$imagen.pgm $parametros
./bin/canny-cudaM ./pgm/$imagen.pgm $parametros
```

Donde:

imagen=luna10000.pgm

parametros="2 0.33 0.66"

RESULTADOS

Se realizó una ejecución utilizando una imagen de **100MB** con los parámetros **2 0.33 0.66**, y se obtuvo los siguientes resultados:

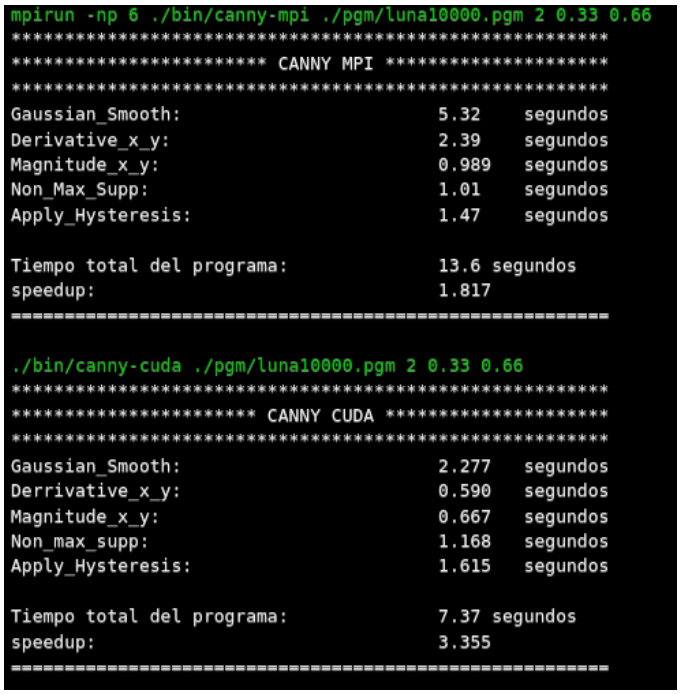
```
* EJECUCION

./bin/canny-serial ./pgm/luna10000.pgm 2 0.33 0.66
******************************************************
******************** CANNY SERIAL ********************
******************************************************
Gaussian_Smooth:                    15.2    segundos
Derivative_x_y:                     1.95    segundos
Magnitude_x_y:                      1.95    segundos
Non_Max_Supp:                       3.78    segundos
Apply_Hysteresis:                   1.62    segundos

Tiempo total del programa:          24.7 segundos
======================================================

./bin/canny-openmp ./pgm/luna10000.pgm 2 0.33 0.66 6
******************************************************
******************** CANNY OPENMP ********************
******************************************************
Gaussian_Smooth:                    3.41    segundos
Derivative_x_y:                     0.0947  segundos
Magnitude_x_y:                      0.34    segundos
Non_Max_Supp:                       0.672   segundos
Apply_hysteresis:                   0.785   segundos

Tiempo total del programa:          6.39 segundos
speedup:                            3.872
======================================================
```



```
mpirun -np 6 ./bin/canny-mpi ./pgm/luna10000.pgm 2 0.33 0.66
******************************************************
********************** CANNY MPI *********************
******************************************************
Gaussian_Smooth:                    5.32    segundos
Derivative_x_y:                     2.39    segundos
Magnitude_x_y:                      0.989   segundos
Non_Max_Supp:                       1.01    segundos
Apply_Hysteresis:                   1.47    segundos

Tiempo total del programa:          13.6 segundos
speedup:                            1.817
======================================================

./bin/canny-cuda ./pgm/luna10000.pgm 2 0.33 0.66
******************************************************
********************** CANNY CUDA ********************
******************************************************
Gaussian_Smooth:                    2.277   segundos
Derrivative_x_y:                    0.590   segundos
Magnitude_x_y:                      0.667   segundos
Non_max_supp:                       1.168   segundos
Apply_Hysteresis:                   1.615   segundos

Tiempo total del programa:          7.37 segundos
speedup:                            3.355
======================================================
```

Para comprobar que los resultados son exactamente los mismos a los de la ejecución serial, se utilizó un comando de suma de control md5:

```
md5sum ./out/imagen-serial.pgm
md5sum ./out/imagen-openmp.pgm
md5sum ./out/imagen-mpi.pgm
md5sum ./out/imagen-cuda.pgm
```

Y se obtiene una salida como la que sigue:

```
* COMPARACION POR CHECKSUM
557fae1095e7d6ecb4f142629dcfb5f1  ./out/imagen-serial.pgm
557fae1095e7d6ecb4f142629dcfb5f1  ./out/imagen-openmp.pgm
557fae1095e7d6ecb4f142629dcfb5f1  ./out/imagen-mpi.pgm
557fae1095e7d6ecb4f142629dcfb5f1  ./out/imagen-cuda.pgm
```

Estas sumas de control dan exactamente el mismo resultado, por lo que podemos decir que los resultados son exactamente el mismo.

CONCLUSION

Se pudo realizar tres versiones paralelas de un mismo algoritmo implementando las metodologías de OpenMP, MPI, y CUDA, y se obtuvo interesantes mejoras, con speedups que rondan los 2, 3 y 3 unidades en promedio, respectivamente.

Se comprobó a medida que se desarrollaba el código, que los resultados eran exactamente iguales al resultado de la ejecución serial, por medio la suma de control md5.

OpenMP resultó ser una librería bastante limpia de usar ya que no genera demasiadas modificaciones internas del código.

MPI nos pareció el método más rebuscado debido a la cantidad de variables a tener en cuenta en sus llamadas, y errores a veces no muy claros.

La arquitectura CUDA nos pareció muy interesante debido a la facilidad de transportar el código a un kernel y hacerlo funcionar sobre cientos de procesadores con solo unas modificaciones, y además por su gran rendimiento y capacidad de optimización.

BIBLIOGRAFIA

Algoritmo de Canny:

https://www.researchgate.net/publication/267240432_Deteccion_de_bordes_mediante_el_algoritmo_de_Canny

Funciones de MPI:

https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php

Especificaciones GeForce GTX 380:

https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications

API CUDA:

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions