Rendimiento de algoritmo para la generación del efecto borroso de una imagen

Juan Gama, Raúl Ramírez
Universidad Nacional Bogotá, Colombia
Email:jcgamar@unal.edu.co, raaramirezpe@unal.edu.co

Figura 1: Escudo de la Universidad.

***Resumen*—Veremos las diferencias de rendimiento en diferentes arquitecturas paralelas para realizar efecto blur en imágenes de diferentes resoluciones, bajo la suposición que el algoritmo secuencial es el mejor posible. Las arquitecturas que compararemos serán el modelo POSIX, CUDA de Nvidia y OpenMP de intel. Se compara usando tablas y gráficas con medidas de rendimiento tiempo y speed-up**

I. Introducción.

En procesamiento de imágenes existen varios algoritmos para realizar un efecto borroso sobre una imagen, de ellos se utilizará el algoritmo de gauss conocido como **gaussian blur**. A continuación se realizarán las definiciones pertinentes para entender como funciona este algoritmo.

Convolución se puede definir como un operador matemático que transforma dos funciones $f(x)$ y $g(x)$ en una tercera función $h(x)$.

kernel gaussiano Se puede representar como un conjunto cuadrado de píxeles donde los valores de cada uno de ellos corresponden a los valores de una curva gaussiana (en 2d).Y que representan una porción de la imagen completa, tal y como se muestra en la **Figura 2**

gaussian blur Se puede definir de una manera simple el funcionamiento de este algoritmo como sigue:

Figura 2: kernel gaussiano

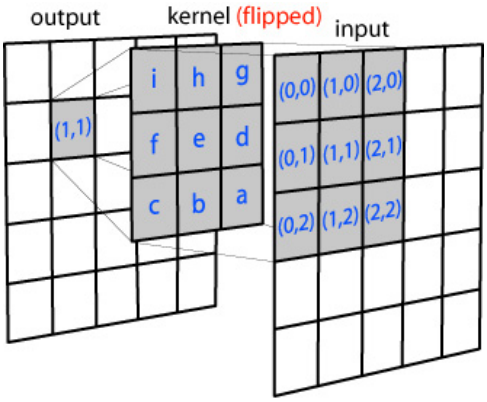


Figura 3: gaussian blur

Cada píxel de la imagen se multiplica por el núcleo gaussiano. Esto se hace colocando el píxel central del núcleo en el píxel de la imagen y multiplicando los valores en la imagen original por los píxeles del núcleo que se superponen. Los valores resultantes de estas multiplicaciones se suman y ese resulta

II. Algoritmo secuencial

Para el manejo de imágenes nos valimos de la librería **OpenCV** pues da facilidades de manejo de varios formatos de imágenes, acceso a transformaciones, y la mado

se utiliza para el valor en el píxel de destino, esto se ilustra en la **Figura 3**

triz de valores de los píxeles tanto en forma matricial como en su forma de dirección en memoria.

El algoritmo de difuminado, es realizar un recorrido sobre una imagen y modificar cada píxel de ella, donde la modificación de cada píxel es la suma ponderada de sus píxeles vecinos. En base a esto debemos recorrer la imagen y a su vez recorremos sus píxeles vecinos, al recorrido de sus píxeles vecinos en base a un radio lo llamaremos kernel. El recorrido de la imagen lo vemos en el Listing 1. A través de componentes de openCV accedemos a la dirección en memoria donde esta alojada la imagen.

```
//recorrer la imagen
for (int i = 0; i < cols*rows; i++) {
  uchar3 aux=prom(i,rows,cols,radio);
  (ans + i)->x = aux.x;
  (ans + i)->y = aux.y;
  (ans + i)->z = aux.z;
}
```

Listing 1: Este código representa el recorrido sobre los píxeles de la imagen, luego llama a la función prom que obtiene el promedio de sus vecinos y las asigna al píxel.

La función **prom()** realiza el recorrido por el kernel y le da un valor o peso a cada píxel, el peso se asemeja a una función de probabilidad, para esto tenemos dos opciones damos un peso igual a cada píxel (probabilidad uniforme) o a medida que se alejan del punto damos una probabilidad menor (distribución normal), esta ultima representada en la ecuación 1, la función normal es la que escogimos, pues la diferencia entre un proceso paralelo y secuencial es mas evidente.

$$e^{-\frac{x^2+y^2}{2r^2}} \tag{1}$$

III. Paralelización del algoritmo.

para la paralelización del algoritmo tenemos en cuenta que cada hilo debe trabajar en espacios de memoria diferentes de tal manera que no hayan condiciones de carrera, además queremos evitar la exclusión mutua, por tanto los índices de los hilos iniciaran en el índice con la misma identificación de su hilo y aumentaran el numero total de hilos invocados.

III-A. Paralelización en POSIX

Para aplicar lo dicho anteriormente debemos darle una identificación a los hilos que invoquemos. La ramificación en hilos se ve en el Listing 3 y la forma en que recorren la imagen es similar al código del Listing 1.

```
uchar3 prom(int pos,
            int rows,
            int cols,
            int radio) {

  float  sum_peso;
  float3 sum = { 0,0,0 };

  sum_peso = 0;

  int *ptr_aux = iToxy(pos, cols);
  int x = *ptr_aux;
  int y = *(ptr_aux + 1);
  free(ptr_aux);

  for (int k = -radio; k <= radio; k++)
  for (int j = -radio; j <= radio; j++){
    if ((x + k) >= 0 && (x + k) < cols &&
      (y + j) >= 0 && (y + j) < rows) {
      float peso = peso(k,j,radio);
      int i = xyToi(x+k, y+j, cols);
      sum.x += peso*(src + i)->x;
      sum.y += peso*(src + i)->y;
      sum.z += peso*(src + i)->z;
      sum_peso += peso;
        }
  }
  uchar3 ans;

  ans.x = (uchar)floor(sum.x/sum_peso);
  ans.y = (uchar)floor(sum.y/sum_peso);
  ans.z = (uchar)floor(sum.z/sum_peso);
}
```

Listing 2: Aquí calculamos el valor que debe tomar el píxel actual con base a la ecuación 1

III-B. Paralelización en CUDA

La paralelización para su ejecución en GPU es un poco más laboriosa pero sigue el mismo principio que ya mencionamos. Agregamos lineas para reservar la memoria y copiar los datos, tanto host-device como device-host. El código que ejecuta cada hilo esta en el Listing **??**. Además agregamos un par de lineas de código para optimizar el uso de nucleos en el device como muestra el Listing 5.

III-C. Paralelización en OpenMP

La paralelización para su ejecución en OpenMP es muy similar a la implementación que se realiza en el código secuencial y sigue el mismo principio que ya

```
//Invocar Threads
for (int i = 0; i < THREADS; i++) {
  threads.push_back(
     std::thread(thread_Blur,
                 src.data,
                 ans,
                 src.cols,
                 src.rows,
                 n_threads,
                 radio,
                 i ) );  //id
}
for (std::thread &t : threads) {
  t.join();
}
src.data = (uchar*)ans;
```

Listing 3: Ciclo para invocar los hilos a cada hilo se le pasa un id para que sepa en que píxeles debe actuar. Luego, espera que los hilos acaben su trabajo y a la matriz que contiene la imagen de la librería OpenCV le asigna la nueva dirección de memoria

```
//Caracteristicas del device
cudaSetDevice(0);
cudaDeviceProp d_Prop;
cudaGetDeviceProperties(&d_Prop, 0);
int max_threads = _ConvertSMVer2Cores(
        d_Prop.major, d_Prop.minor);

//Bloques y threads por bloque
BLOCKS = (n_threads/(max_threads*2))+1;
THREADS = n_threads/BLOCKS;
```

Listing 4: Sección de código para identificar cuantos procesadores por multiprocesador hay en el device y balancear de forma correcta las cargas por bloque. La función _ConvertSMVer2Cores es tomada de helper_cuda.h de los ejemplos de cuda.

mencionamos. La manera de lanzar hilos con OpenMP se puede hacer principalmente de 2 maneras, una es dejando esta responsabilidad a OpenMP, es decir la API gestiona cuantos hilos lanza, o definiendo manualmente cuantos hilos lanzamos, para ello se usa la función **omp_set_num_threads(N_THREADS)** donde *N_THREADS* es el número de hilos. y para paralelizar un segmento de código se usa la sentencia **pragma omp parallel**{ **bloque de sentencias** }. Con base en esto se puede ver la sección que se paraleliza en este caso en 6.

Para la implementación del efecto borroso el algoritmo sigue siendo muy similar al de las versiones anteriores, lo

```
__global__ void cudaBlur(
        uchar3 *src,
        uchar3 *ans,
        int cols,
        int rows,
        int n_threads,
        int radio){

  int id=blockIdx.x*blockDim.x+threadIdx.x;

  //Aumenta segun numero de threads
  for (int i=id;
       i<cols*rows;
       i+=n_threads){

    *(ans+i)= prom(src,
                   i,
                   rows,
                   cols,
                   radio);
  }

  return;
}
```

Listing 5: Sección de código que ejecuta cada hilo en GPU. Al igual que los hilos en POSIX estos aumentan en una razón igual al numero de threads invocados y la identificación de cada hilo la da las funciones predefinidas de cuda.

que se modificó en esta ocasión fue la forma de recorrer las posiciones del arreglo, las cuales se usan para aplicar el metodo gaussiano, así: 7.

IV. Experimentos y resultados.

Para probar los programas se realizarán pruebas sobre tres imágenes distintas de **720p, 1080p y 4k**, las pruebas son ejecutar el efecto borroso con radios distintos de **3, 6, 12 y 15**, tanto en el programa secuencial como en la versión en paralelo, para la version en paralelo de POSIX cada radio de kernel será probado textbf2, 4, 8 y 16 hilos mientras que para CUDA se probara con textbf64, 128, 256 y 1024 hilos.

El **cuadro 1** muestra los resultados de tiempo en segundos obtenidos con la versión serial del programa, aquí se puede observar como la relación entre la resolución de la imagen y el tiempo que toma en realizar el efecto borroso en la imagen es proporcional y se puede pensar que su comportamiento tiende a ser exponencial. La

```
sscanf(argv[3], "%i",&N_THREADS);
omp_set_num_threads(N_THREADS);
printf( "Numero de hilos %i \n",N_THREADS);

#pragma omp parallel
{
 int id = omp_get_thread_num();
 int n_threads = omp_get_num_threads();
 int nl = src.rows;
 int nc = src.cols * src.channels();

 for (int i = id; i <src.rows*nc ;
 i += n_threads) {
 uchar* dataout = dst.ptr<uchar>(i/nc);
 prom_punto(i,dataout);
         }
}
```

Listing 6: Se paraleliza el bloque que se encarga de recorrer el arreglo de la imagen, según la cantidad de hilos que se creen será el tamaño del arreglo que cada hilo recorra, cada hilo llama a la función que calcula el blur para el píxel en el que se encuentre, pasando como parámetros, el la posición en el arreglo y un apuntador a la fila en cual se encuentra.

imagen/ radio	720p	1080p	4k
3	3.36	6.28	8.53
6	22.08	43.29	57.91
12	80.83	154.81	210.10
15	139.13	268.66	355.99

Cuadro I: Resultados versión serial

relación anterior también se mantiene para los distintos radios de kernel que fueron probados.

V. Conclusiones

Al analizar los tiempos de ejecución nos damos cuenta que hay bastantes diferencias entre estos. Sin embargo, estas diferencias pueden contradecir la intuición, pues hay casos en donde resulta mucho más conveniente utilizar implementación que usen recursos externos como la GPU y existen otros en los que al contrario los recursos locales del host son mucho más convenientes.

V-1. GPU VS HOST : La ubicación juega un papel bastante importante. Primero, con un kernel de radio 3 obtenemos un resultado muy sorprendente. A pesar de que nuestra implementación en POSIX no se ve muy prometedora para este caso, pues no logra una ganancia real de tiempo, no podemos decir lo mismo con OpenMP, pues si la comparamos con CUDA podemos

Kernel	hilos	720p	1080p	4k
3	2	4.63	8.95	42.59
	4	3.69	6.7	35.42
	8	3.9	7.51	36.76
	16	4.12	6.9	35.05
6	2	15.03	29.42	145.41
	4	12.98	23.31	119.57
	8	12.98	24.27	120.66
	16	13.28	24.9	120.12
12	2	58.8	106.9	515.04
	4	44.63	82.48	421.33
	8	46.09	87	434.76
	16	47.77	89.89	437.75
15	2	91.46	173.32	839.48
	4	71.04	135.06	646.08
	8	64.36	121.76	604.46
	16	70.23	131.02	658.92

Cuadro II: Resultados de la ejecución con POSIX

Kernel	hilos	720p	1080p	4k
3	64	3.72	6.8	32.66
	128	2.74	5	24.26
	256	2.17	3.6	16.82
	1024	1.7	2.96	13.93
6	64	6.2	11.56	55.84
	128	3.44	5.89	28.66
	256	2.32	4.12	19.55
	1024	1.91	3.36	15.87
12	64	15.51	29.11	142.42
	128	8.05	14.89	73.19
	256	4.05	7.39	35.66
	1024	2.22	3.88	18.48
15	64	22.39	43.98	207.68
	128	11.48	21.53	105.58
	256	5.73	10.51	51.14
	1024	2.67	4.82	23.34

Cuadro III: Resultados de la ejecución con CUDA.

Kernel	hilos	720p	1080p	4k
3	2	4.66	8.28	42.24
	4	3.83	7.16	33.47
	8	3.57	6.67	31.53
	16	3.43	6.53	37.54
6	2	17.19	30.38	156.04
	4	13.18	23.6	113.35
	8	11.44	21.62	105.72
	16	11.45	21.5	105.9
12	2	53.16	102.28	498.86
	4	41.21	78.08	409.5
	8	43.3	81.95	416.85
	16	45.7	85.43	421.18
15	2	84.29	155.47	757.39
	4	62.68	118.02	619.9
	8	67.65	130.16	636.35
	16	62.31	118.34	581.48

Cuadro IV: Resultados de la ejecución con OpenMP.

```
for(int j=-radio_kernel; j<=radio_kernel; j++)
{
//dirección de la fila j +y
uchar* data = src.ptr<uchar>(y+j);
for (int i= -radio_kernel; i <= radio_kernel; i++){
 if (((x + i-3) >= 0
 && (x + i-3) <  src.cols*src.channels()
 && ((y + j) >= 0 && (y + j) < src.rows)){
  float peso = exp(-(i*i + j*j)
  / (float)(2*radio_kernel*radio_kernel))
  / (pi*2*radio_kernel*radio_kernel);
  sum1 += peso*data[x+i*3-3];
  sum2 += peso*data[x+i*3-2];
  sum3 += peso*data[x+i*3-1];
  sum_peso += peso;
  }
 }
}
if( x-3 >=0 )
{
 dataout[x-3]= (uchar)floor(sum1/sum_peso);
 dataout[x-2]= (uchar)floor(sum2/sum_peso);
 dataout[x-1]= (uchar)floor(sum3/sum_peso);
 }
}
```

Listing 7: Como la matriz de la imagen a color,cargada con OpenCV, se puede visualizar como una matriz de filas*columnas, donde cada posición de la columna está compuesta por el número de canales de la imagen, en este caso es 3 por RGB, entonces con base a eso se recorren las posiciones dependiendo el canal para hacer el efecto borroso y asignarlo a píxel correspondiente

ver una gran diferencia en el tiempo. OpenMP con un speed-up de 3, logra sacar una gran ventaja a la GPU con un speed-up de 2 para el caso de la imagen más liviana y siendo una constante en las imágenes con más peso. Podemos decir que esto se debe a que la tarjeta gráfica gasta más tiempo comparado con los cálculos, trasladando los datos que realizando cálculos pues un kernel de 3 en este caso no somete a los hilos a realizar grandes cálculos. Aunque no podemos decir lo mismo para radios mucho mayores, pues la ganancia de tiempo en la implementación con CUDA es bastante mayor comparada con las implementaciones que usan recursos del host, en este caso las operaciones por punto son más complejas de realizar y cada hilo debe gastar más tiempo realizando cálculos. Es allí donde la GPU al tener procesadores independientes ejecutando estas operaciones obtiene una ganancia de tiempo.

V-2. OpenMP VS POSIX.: Cabe mencionar que las diferencias de tiempo para un kernel de radio 3 también son importantes entre OpenMP y la implementación con POSIX, dado que ya no tenemos la necesidad de trasladar los datos de un espacio de memoria otro, seguimos teniendo menores tiempos de ejecución con el código implementado usando OpenMP. A pesar de que consideramos que nuestro código esta implementado de una manera eficiente, con uso de memoria dinámica, accesos al apuntador de datos directamente, y la equivalencia de código usada en OpenMP y POSIX. Open MP obtiene un mejor manejo de los recursos de sistema. Por lo cual concluimos que es mucho mejor usar una implementación con OpenMP al ser más intuitiva y eficiente. Sin embargo, cabe mencionar que hay que tener en cuenta las mismas restricciones y complicaciones que trae usar hilos manualmente pues debemos tener cuidado con las condiciones de carrera y en el caso de ser así usar exclusiones mutuas, pero esto solo de ser estrictamente necesario pues también es deseable evitar estas lo más posible.

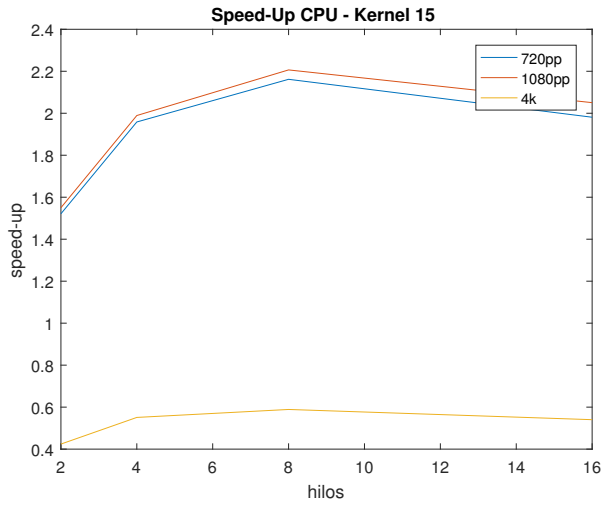
Referencias

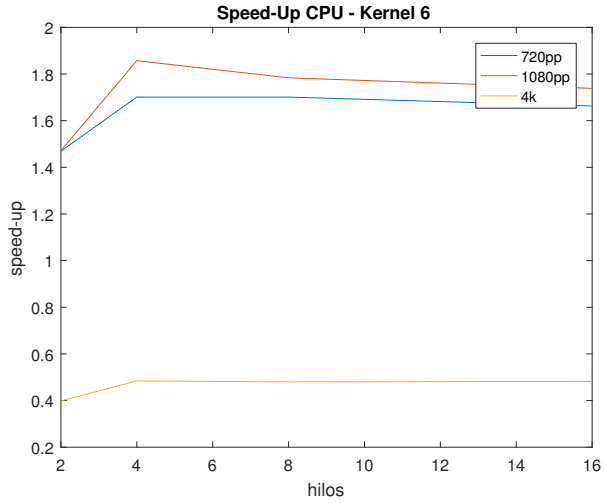
[1] ”Gaussian Blur”. En.wikipedia.org. N.p., 2017. Web. 26 Feb. 2017.

[2] https://ark.intel.com/es/products/65693/Intel-Core-i3-3220-Processor-3M-Cache-3_30-GH

[3] http://blog.ivank.net/fastest-gaussian-blur.html

[4] http://www.pixelstech.net/article/1353768112-Gaussian-Blur-Algorithm





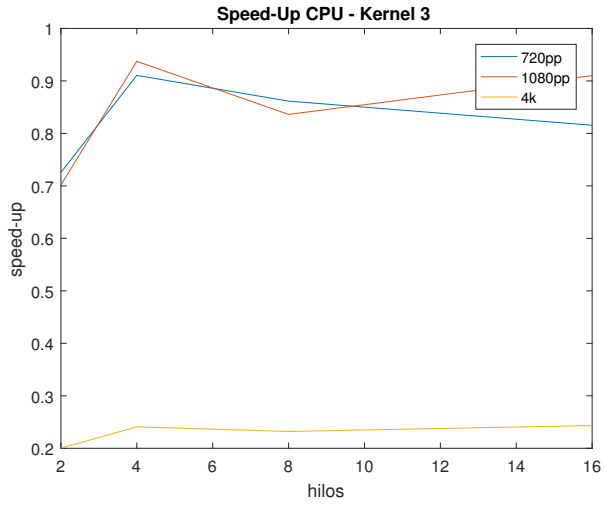
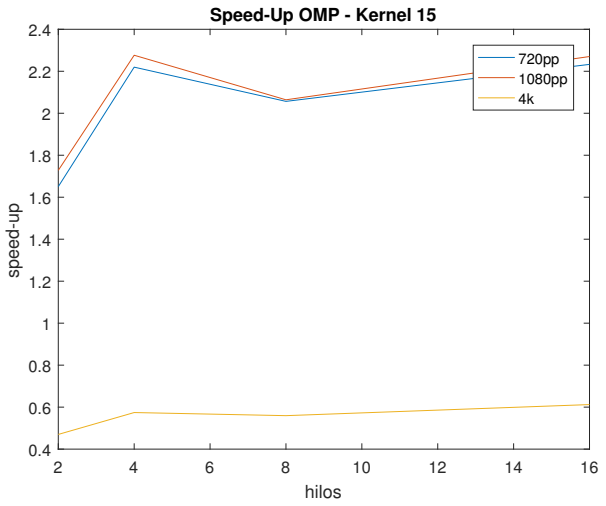
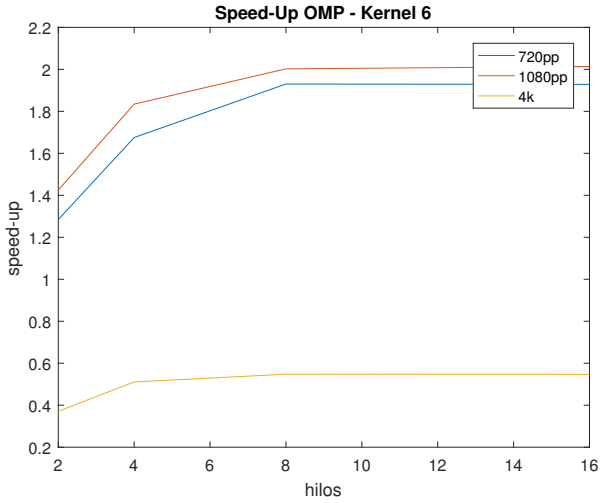


Figura 4: Speed-Up para diferentes tipos de kernel ejecutados con POSIX.





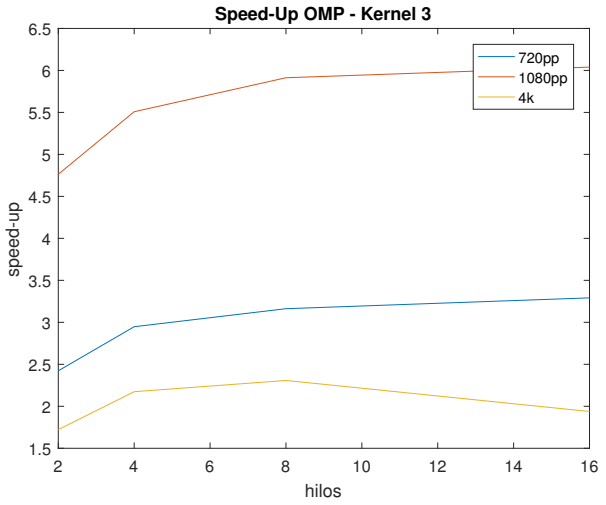
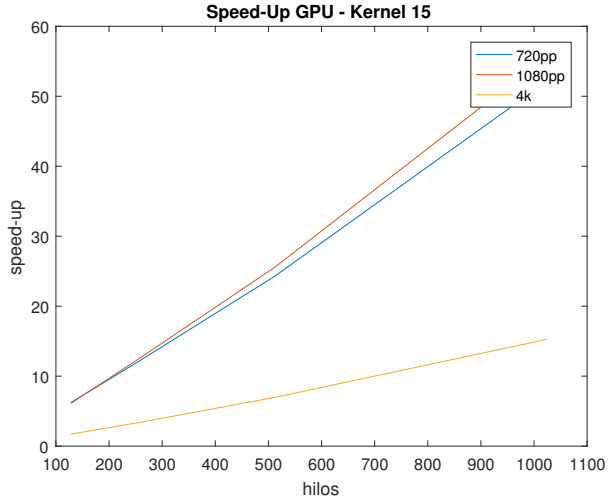
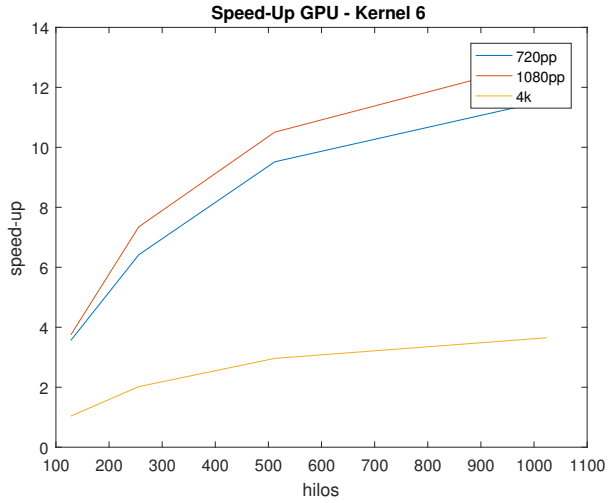


Figura 5: Speed-Up para diferentes tipos de kernel ejecutados con OpenMP.





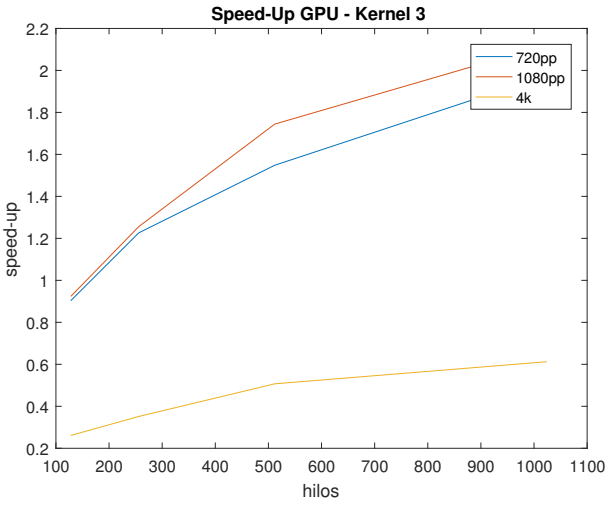
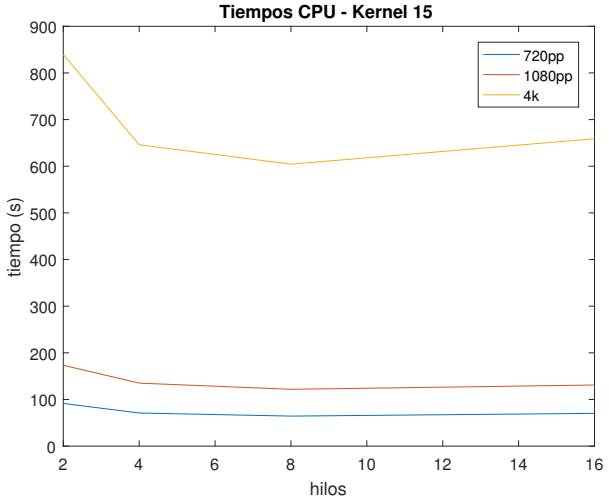
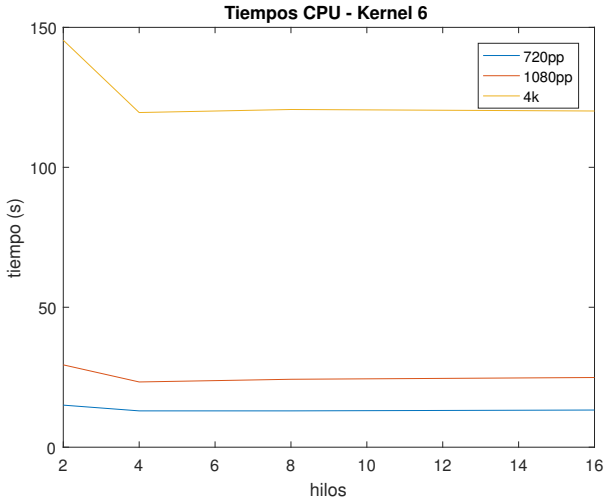


Figura 6: Speed-Up para diferentes tipos de kernel ejecutados con CUDA.





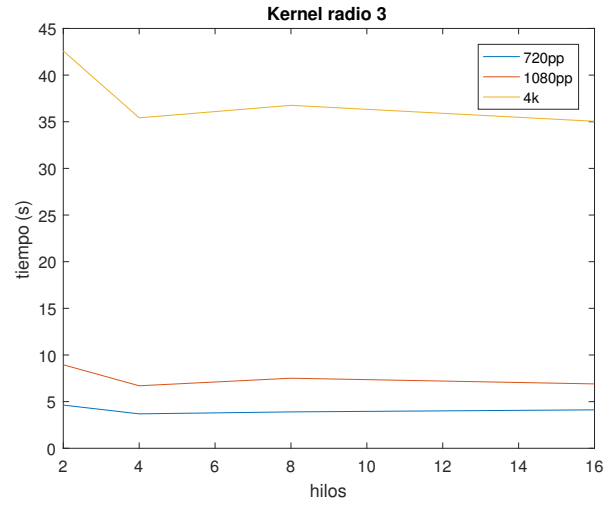
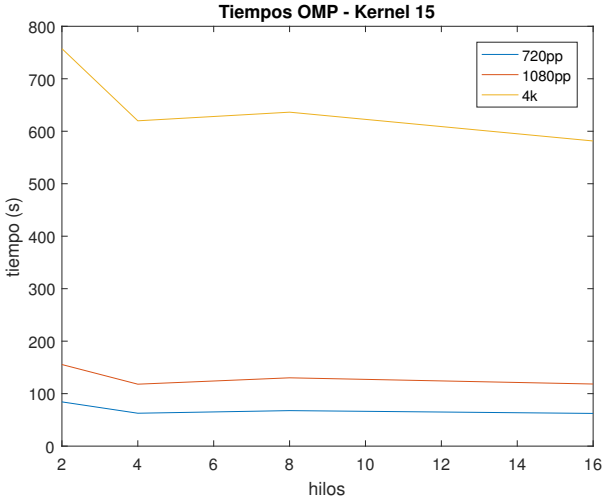
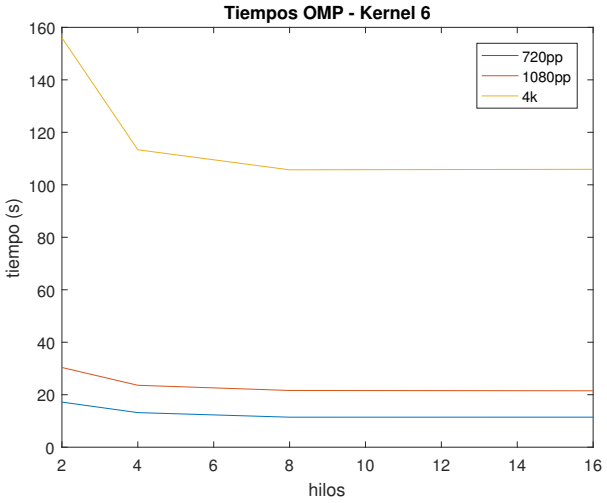


Figura 7: Tiempos de ejecución para diferentes tipos de kernel ejecutados con POSIX.





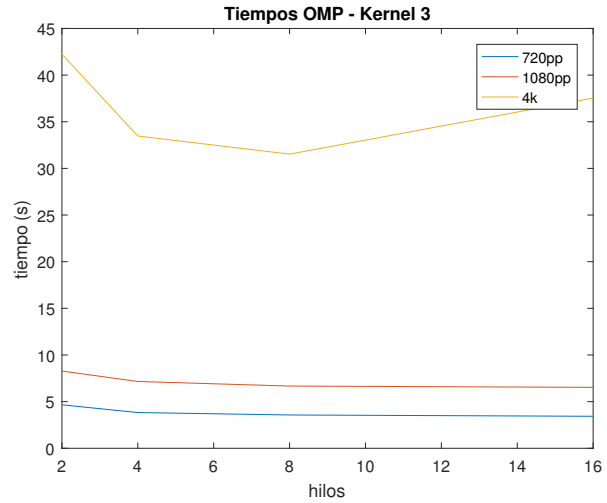
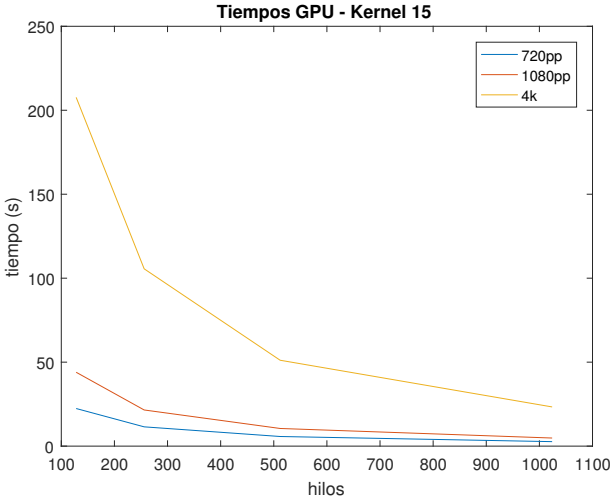
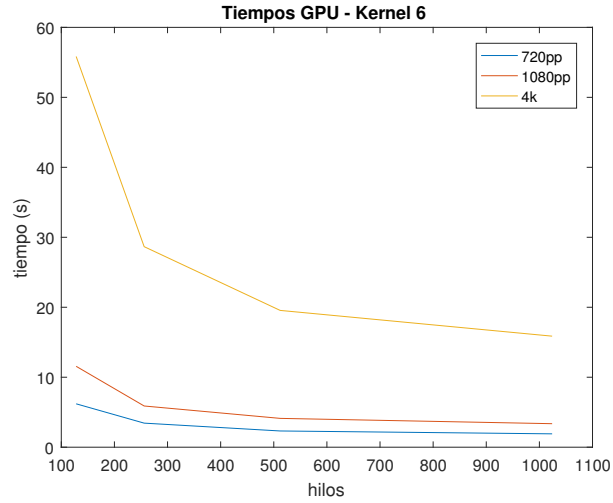


Figura 8: Tiempos de ejecución para diferentes tipos de kernel ejecutados con OpenMP.





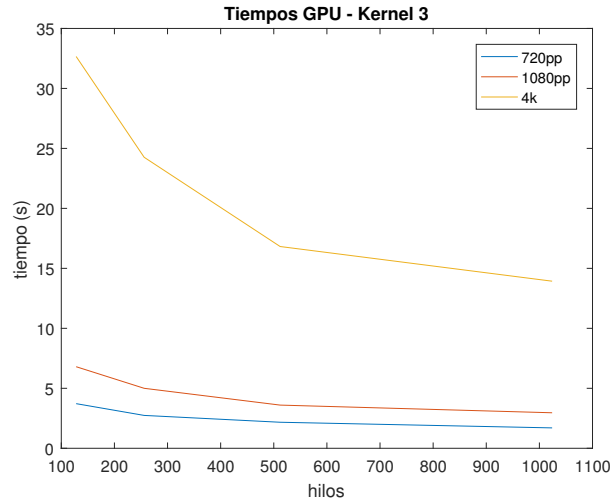


Figura 9: Tiempos de ejecución para diferentes tipos de kernel ejecutados con CUDA.