PROYECTO GPU

Juan Siverio Rojas

20/01/2021

Contenido

Objetivo

Se realizará un histograma de un vector V de un número elevado N de elementos enteros aleatorios. El histograma consiste en un vector H que tiene M elementos que representan "cajas". En cada caja se cuenta el número de veces que ha aparecido un elemento del vector V con el valor adecuado para asignarlo a esa caja (normalmente cada caja representa un rango o intervalo de valores).

En nuestro caso, para simplificar la asignación del elemento de V a su caja correspondiente del histograma, vamos a realizar la operación ValorElementoV Modulo M, que directamente nos da el índice de la caja del histograma al que pertenecerá ese elemento y que deberemos incrementar. Se sugiere como N un valor del orden de millones de elementos y como M, 8 cajas.

Se utilizarán varios algoritmos basados en GPU Nvidia, con su programación basada en CUDA. También se ha programa la versión del algoritmo para CPU.

Se realizarán diferentes pruebas y se medirán los tiempos para hacer comparaciones.

Previos

El programa se adecua a varios parámetros mediante unas definiciones previas. Se puede parametrizar lo siguiente:

NUMELEMENTSV -> número de elementos del vector de que se quiere calcular su histograma.

NUMELEMENTSH -> número de elementos del histograma.

TREADSPERBLOCK -> número de hilos por bloque. Siendo este adaptado al mayor posible que soporta la GPU en caso de indicar un valor superior a este.

LOOPS -> Indica el número de repeticiones que se ejecutará cada algoritmo, de esta forma obtendremos resultados más fiables y también el impacto de las memorias cache, en la ejecución de los diferentes algoritmos.

Nota: Se debe considerar que los números de bloque sean Par, puesto que con los algoritmos de reducción, aún no está completamente contemplada la posibilidad de los bloques impares. Para cumplir con esta restricción, solo hay que tener en cuenta que NUMELEMENSTV módulo por THREADSPERBLOCK, sea igual a cero.

Descripción

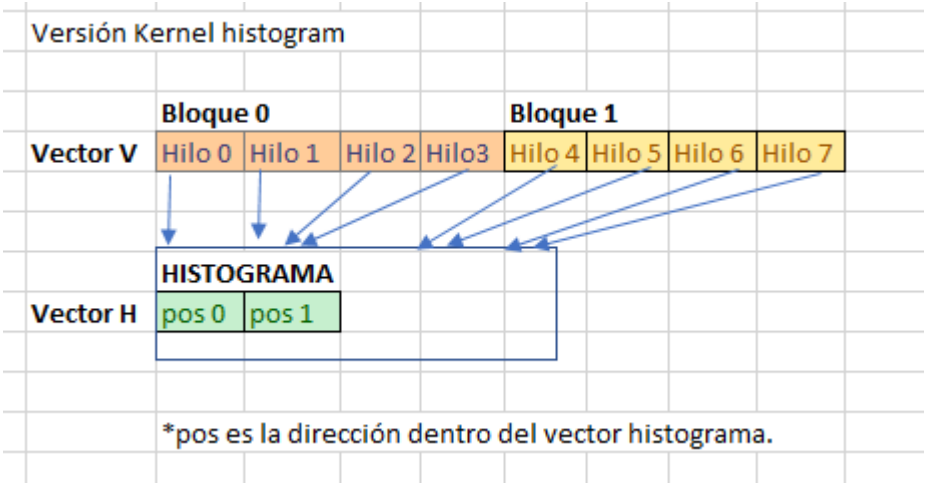
Se adjunta fichero **"histograma.cu"** que contiene toda la programación dividida en las siguientes funciones.

Kernels Cuda

(1) Kernel 1 "histogram": El siguiente Kernel realizará el cálculo del histograma pasado, y lo almacenará en el vector del histograma. Esto se realizará mediante operaciones atómicas por cada hilo de cada bloque, directamente a la memoria global de la GPU donde reside el vector del Histograma.

```
__global__ void histogram(int *V, int * H, int numElementsV, int numElementsH)
```

El siguiente esquema, se muestra como es el funcionamiento suponiendo 2 bloques de ejecución con 4 hilos cada uno, para un histograma de 2 elementos.

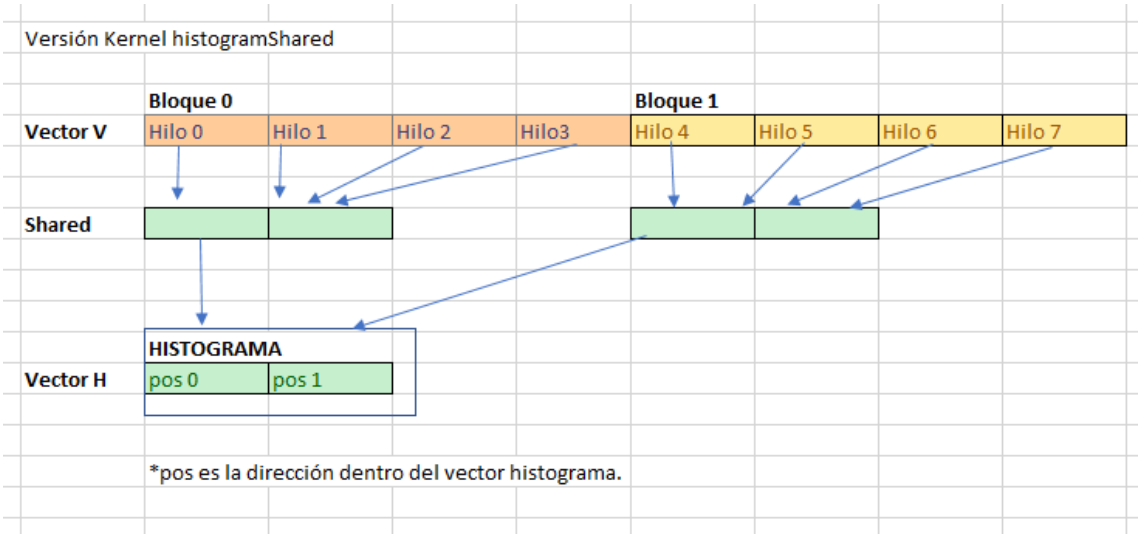


(2) Kernel 2 "histogramShared": Mejora Kernel 1**:** El siguiente Kernel realizará lo mismo que el anterior (Kernel 1) pero con la diferencia de que las operaciones atómicas serán realizadas por los hilos de cada bloque a una memoria compartida **"__shared__"** inicializada a cero por el primer hilo de cada bloque.

Este mismo hilo, será el encargado de actualizar el vector histograma de la memoria global, también con operaciones atómicas, tras una sincronización de todos los hilos del bloque.

```
__global__ void histogramShared(int *V, int * H, int num
ElementsV, int numElementsH)
```

El siguiente esquema, se muestra como es el funcionamiento suponiendo 2 bloques de ejecución con 4 hilos cada uno, para un histograma de 2 elementos.



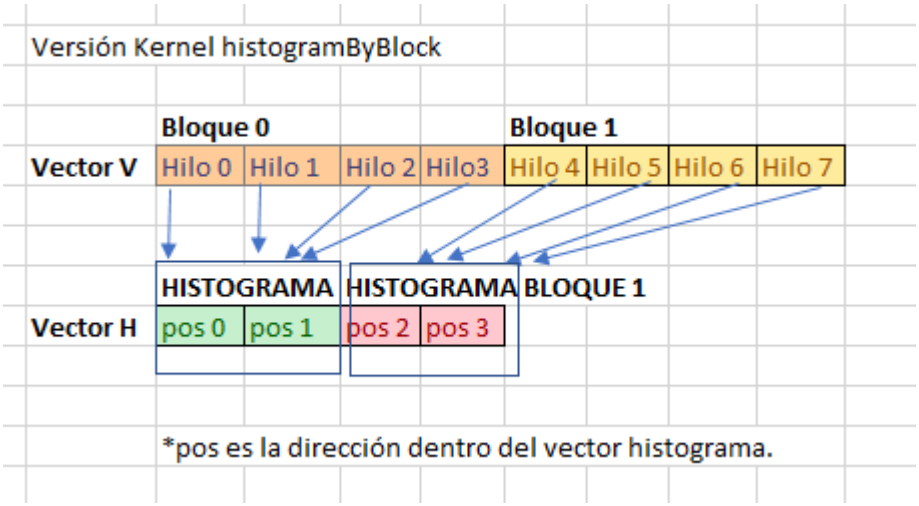
(3) Kernel 3 "histogramByBlock": El siguiente Kernel, funciona mediante la creación y distribución de un histograma por cada bloque. Esto creará tantos vectores histogramas como bloques necesarios para cubrir todo el vector V, por lo que este Kernel proporciona solo una parte de la implementación, puesto que tras la ejecución de este, se debe de llamar a otro Kernel que se encargará de la suma de todos los Histogramas en solo uno.

Antes de la ejecución de este kernel, el vector Histograma será creado teniendo un tamaño en un único vector, equivalente al número de elementos del histograma multiplicado por el número de bloques que ocupa el vector V.

La idea es que cada hilo de un mismo bloque, pueda actualizar su propio histograma utilizando operaciones atómicas directamente a la memoria global de la GPU donde está almacenado el vector histograma por cada bloque. De esa forma se evita que millones de hilos intenten acceder a pocas porciones de memoria y quede distribuida las operaciones atómicas entre un vector mucho más grande.

```
__global__ void histogramByBlock(int *V, int * H, int nu
mElementsV, int numElementsH)
```

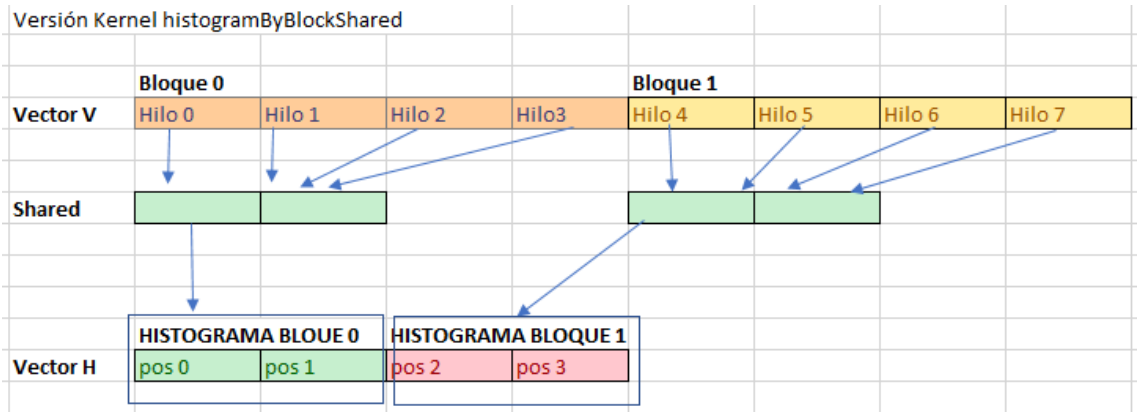
El siguiente esquema, se muestra como es el funcionamiento suponiendo 2 bloques de ejecución con 4 hilos cada uno, para un histograma de 2 elementos.



(4) Kernel 4 "histogramByBlockShared": Mejora Kernel 3**:** Este Kernel realizará lo mismo que el anterior, pero utilizará memoria compartida **__shared__** para cada bloque. Después de forma atómica el hilo 0 de cada bloque, se encargará de actualizar su propio histograma, distribuyendo las operaciones atómicas. Como con el Kernel anterior, se necesita de un segundo kernel encargado de realizar las sumas entre todos los histogramas generados.

```
__global__ void  histogramByBlockShared(int *V, int * H,
 int numElementsV, int numElementsH)
```

El siguiente esquema, se muestra como es el funcionamiento suponiendo 2 bloques de ejecución con 4 hilos cada uno, para un histograma de 2 elementos.



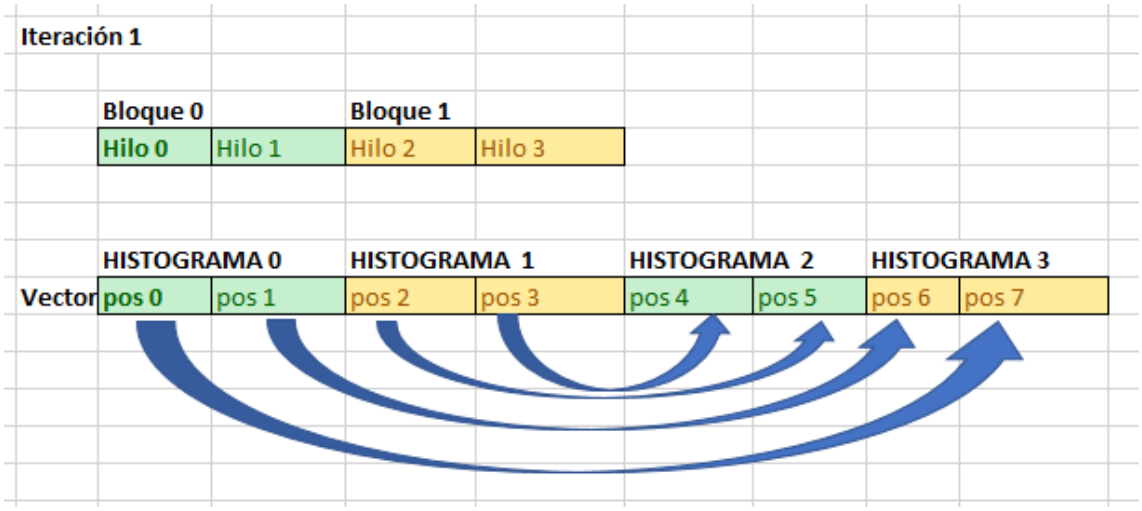
(5) Kernel "sumHistogram": El siguiente Kernel se encarga de realizar una suma de un vector por el método por reducción. Es el Kernel utilizado tras la ejecución de los Kernels 3 ó 4. Devuelve la suma del vector en las primeras posiciones del mismo vector pasado.

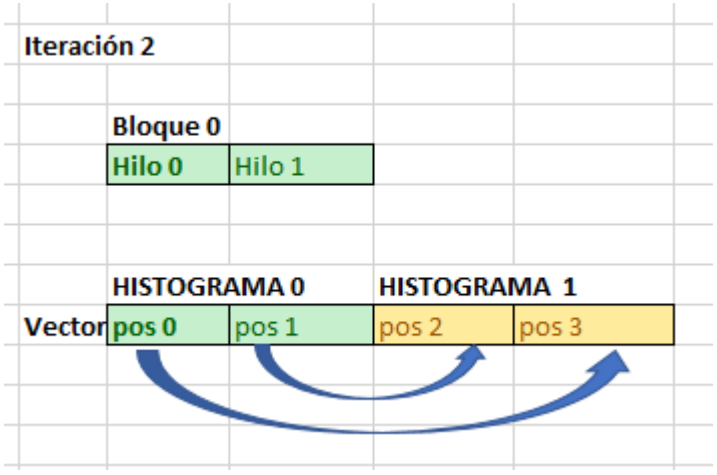
Para el caso concreto de este programa, el vector pasado, será un vector de tamaño **"número elementos histograma"** multiplicado por **"número de bloques que hayan sido necesarios para la ejecución de los kernels 3 ó 4".** Estos múltiples histogramas se encontrarán consecutivos en la memoria.

Este kernel se ejecuta con número de bloques igual al número de bloques(que en este caso equivale al número de histogramas) dividido entre 2, puesto que en el método de reducción, con la mitad de los hilos y sabiendo el tamaño del vector, es suficiente.
El número de hilos por bloque es igual al número de elementos de un histograma.

```
__global__ void sumHistogram(int * h, int blocksPerGrid)
```

Por ejemplo, para el caso de un vector Histograma de 4 histogramas de dos elementos cada uno, la operativa sería como se muestra en el siguiente esquema. En la segunda iteración ya se tendría el resultado en **pos 0 y pos 1**





Funciones externas.

(1) Función "calculateHistogramByCpu" :La siguiente función, realiza el histograma de un vector, utilizando solo la CPU. También se encarga de calcular el tiempo mediante la función de la librería estándar de C, **clock().**

```
int * calculateHistogramByCpu(int * vector, int numEleme
ntsV, int numElementsH)
```

Esta función es con fines comparativos.

(2) Función "calculateHistogramByGpu" La siguiente función realiza el cálculo del histograma de un vector utilizando la GPU. Utiliza diferentes métodos dependiendo de los parámetros.

```
int * calculateHistogramByGpu(int * vector,int numElemen
tsV, int numElementsH, bool byBlock, int threadsPerBlock
, bool shared)
```

Donde:

byblock: true -> Ejecuta kernel por bloques

false -> Ejecuta kernel monolítico.

shared: true -> Ejecuta kernels con memoria compartida.

false -> Ejecuta kernels sin memoria compartida.

Pruebas

Se utilizarán los siguientes valores para la ejecución de las pruebas.

```
#define NUMELEMENTSV 33554432;
#define NUMELEMENTSH 8;
#define THREADSPERBLOCK 1024;
```

Con 1 Iteración por algoritmo.

```
#define LOOPS 1;
```

- Tiempo por CPU

```
Tiempo empleado en calculo por CPU :  0.377256 segundos
Resultado Vector Histograma Calculado por CPU
[4192327]
[4192931]
[4194677]
[4194746]
[4196213]
[4192610]
[4193973]
[4196955]
```

- Ejecución usando Kernel 1 (histogram) Sin bloques y Sin memoria compartida.

```
CUDA kernel -histogram- launch with 32768 blocks of 1024 threads
Resultado Vector Histograma versión Sin bloque y Sin memoria compartida :
[4192327]
[4192931]
[4194677]
[4194746]
[4196213]
[4192610]
[4193973]
[4196955]
```

- Ejecución usando Kernel 2 (histogramShared) Sin bloques y con Memoria Compartida

```
CUDA kernel -histogramShared- launch with 32768 blocks of 1024 threads
Resultado Vector Histograma Versión Sin bloque y Memroria compartida :
[4192327]
[4192931]
[4194677]
[4194746]
[4196213]
[4192610]
[4193973]
[4196955]
```

- Ejecución usando Kernel 3 (histogramByBlock) Con bloques y Sin Memoria Compartida

```
CUDA kernel - histogramByBlock - launch with 32768 blocks of 1024 threads
Resultado Vector Histograma Con bloques y Sin Memoria Compartida :
[4192327]
[4192931]
[4194677]
[4194746]
[4196213]
[4192610]
[4193973]
[4196955]
```

- Ejecución usando Kernel 4 (histogramByBlockShared) Con bloques y Con Memoria Compartida

```
CUDA kernel - histogramByBlockShared - launch with 32768 blocks of 1024 threads
Resultado Vector Histograma Con bloques y Memoria Compartida :
[4192327]
[4192931]
[4194677]
[4194746]
[4196213]
[4192610]
[4193973]
[4196955]
```

Análisis de los tiempos obtenidos.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	69.03%	126.22ms	8	15.777ms	1.4080us	35.060ms	[CUDA memcpy DtoH]
	23.45%	42.886ms	8	5.3607ms	1.4080us	11.236ms	[CUDA memcpy HtoD]
	5.69%	10.413ms	1	10.413ms	10.413ms	10.413ms	histogram(int*, int*, int, int)
	0.88%	1.6018ms	1	1.6018ms	1.6018ms	1.6018ms	histogramByBlock(int*, int*, int, int)
	0.39%	713.92us	1	713.92us	713.92us	713.92us	histogramShared(int*, int*, int, int)
	0.34%	622.98us	1	622.98us	622.98us	622.98us	histogramByBlockShared(int*, int*, int, int)
	0.21%	392.83us	30	13.094us	992ns	170.98us	sumHistogram(int*, int)
API calls:	44.28%	184.87ms	16	11.554ms	12.280us	45.721ms	cudaMemcpy
	42.21%	176.23ms	8	22.028ms	113.32us	168.96ms	cudaMalloc
	12.67%	52.902ms	1	52.902ms	52.902ms	52.902ms	cudaDeviceReset
	0.49%	2.0595ms	8	257.44us	86.402us	326.87us	cudaFree
	0.15%	635.19us	1	635.19us	635.19us	635.19us	cuDeviceTotalMem
	0.11%	451.88us	101	4.4740us	260ns	195.24us	cuDeviceGetAttribute
	0.06%	237.10us	34	6.9730us	3.5500us	26.690us	cudaLaunchKernel
	0.02%	70.922us	1	70.922us	70.922us	70.922us	cuDeviceGetName
	0.00%	11.310us	1	11.310us	11.310us	11.310us	cuDeviceGetPCIBusId
	0.00%	5.5700us	2	2.7850us	290ns	5.2800us	cuDeviceGet
	0.00%	2.4400us	3	813ns	390ns	1.4500us	cuDeviceGetCount
	0.00%	1.0500us	4	262ns	180ns	450ns	cudaGetLastError
	0.00%	420ns	1	420ns	420ns	420ns	cuDeviceGetUuid

- En el caso de las versiones sin bloques, podemos comprobar una enorme mejora del rendimiento solo con utilizar memoria compartida, ya que el **kernel histogram**(versión directa a memoria global), tarda más de 10 ms con respecto a los 713 microsegundos del kernel **histogramShared**, que hace lo mismo pero con acceso a memoria compartida.

Es de alrededor 14 veces más rápido.

- Con respecto a las versiones con bloque, vemos como el **kernel 3 histogramByBlock,** tarda **1.6 milisegundos.**

A este tiempo, en las versiones de bloque, hay que sumar lo que tarda el kernel que se encarga de sumar por reducción de los histogramas resultantes, **sumHistogram,** que además se ejecuta varias veces. En este caso se **ejecuta 15 veces** para este vector, en la imagen aparecen 30 porque realmente se ha llamado desde el **kernel histogramByBlock** y el **kernel histogramByBlockShared.**

Si tenemos en cuenta el tiempo de **392 microsegundos dividido entre 2**, tenemos que sumar **196 microsegundos** a 1.6 milisegundos, y este es el tiempo total empleado por el kernel por bloque.

Como se puede observar, es bastante superior a los 10 milisegundos que consumió el kernel 1, aún así, el kernel 2(histogramShared) sigue obteniendo mejor resultado.

- La duración del **kernel histogramByBlockShared + 15 ejecuciones del kernel sumHistogram** dan un total de 622 +196 = **818 microsegundos** totales.

Vemos que mejora en torno al doble de rendimiento, esta versión de kernel por bloques utilizando memoria compartida con respecto a la que no la usa, pero aún así, sigue siendo más lento que el **kernel 2 de 713 microsegundos.**

Tabla Comparativa para 1 iteración

Kernel	Tiempo	Mejora con respecto a: histogramByBlock	Mejora con respecto a: histogramByBlockShared	Mejora con respecto a: histogramShared	Mejora respecto a: histogram	Mejora respecto a: CPU
Por Cpu	**377 ms**					
histogram	**10.4 ms**					**3600%**
histogramShared	**713 us**				**1300%**	**52700%**
histogramByBlock	**1.6 ms + 196 us**				**400%**	**22000%**
histogramByBlockShared	**622 us + 196 us**	**110%**			**1150%**	**47025%**
sumHistogram	**196 us**					

Con 10 Iteración por algoritmo.

```
#define LOOPS 10;
```

Resultado de tiempos.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   69.77%  1.24745s        80  15.593ms  1.1520us  34.910ms  [CUDA memcpy DtoH]
                   23.73%  424.33ms        80  5.3042ms  1.1520us  11.254ms  [CUDA memcpy HtoD]
                    4.95%  88.507ms        10  8.8507ms  7.3069ms  10.432ms  histogram(int*, int*, int, int)
                    0.76%  13.572ms        10  1.3572ms  1.3500ms  1.3659ms  histogramByBlock(int*, int*, int, int)
                    0.29%  5.2589ms        10  525.89us  525.66us  526.11us  histogramByBlockShared(int*, int*, int, int)
                    0.29%  5.2469ms        10  524.69us  524.48us  525.12us  histogramShared(int*, int*, int, int)
                    0.20%  3.6126ms       300  12.042us     800ns  171.04us  sumHistogram(int*, int)
      API calls:   85.12%  1.80940s       160  11.309ms  10.280us  42.989ms  cudaMemcpy
                   11.07%  235.27ms        80  2.9409ms  111.66us  156.10ms  cudaMalloc
                    2.58%  54.740ms         1  54.740ms  54.740ms  54.740ms  cudaDeviceReset
                    1.07%  22.782ms        80  284.77us  219.83us  347.09us  cudaFree
                    0.11%  2.2917ms       340  6.7400us  3.3100us  33.831us  cudaLaunchKernel
                    0.03%  696.45us         1  696.45us  696.45us  696.45us  cuDeviceTotalMem
                    0.02%  468.27us       101  4.6360us     280ns  202.39us  cuDeviceGetAttribute
                    0.00%  65.742us         1  65.742us  65.742us  65.742us  cuDeviceGetName
                    0.00%  20.911us        40     522ns     160ns  6.3710us  cudaGetLastError
                    0.00%  10.860us         1  10.860us  10.860us  10.860us  cuDeviceGetPCIBusId
                    0.00%  2.4700us         3     823ns     410ns  1.4800us  cuDeviceGetCount
                    0.00%  1.4300us         2     715ns     300ns  1.1300us  cuDeviceGet
                    0.00%     460ns         1     460ns     460ns     460ns  cuDeviceGetUuid
```

- En este caso, parece que en las versiones

Tabla Comparativa para 10 iteraciones

Se puede observar como mejoran todos los tiempos con respecto a lo esperado.

Kernel	Tiempo	Debería tardar Tiempo de 1 iteración * 10	Mejora con respecto a 1 iteración
histogram	**88.5 ms**	104 ms	**18%**
histogramShared	**5.2 ms**	**7.13 ms**	**37%**
histogramByBlock	**13.5 ms + 1.8 ms**	**17 ms**	**11%**
histogramByBlockShared	**5.2 ms + 1.8 ms**	**7 ms**	**11%**
sumHistogram	**1.8 ms**		

Conclusión

De los datos obtenidos se puede deducir claramente varias cuestiones:

- Que cualquier aplicación del algoritmo mediante la GPU, es como mínimo un 3.700 % más rápida que por CPU.

- Que la distribución de las escrituras atómicas entre más zonas de memoria(vector histograma más grande) realizadas por las versiones de bloque, también mejora del orden de un 400% con respecto a las versiones de un solo histograma. Esto es así incluso con la penalización de tener que sumar después todos los histogramas. Parece que es debido a la optimización de lectura/escritura de palabra grandes , de alrededor de 128 bytes, utilizadas por las GPU.

- Y lo más característico a destacar, es que la memoria compartida, es muchísimo más rápida que la global, hasta el punto de que incluso teniendo que realizar más escrituras atómicas (ya que a la de actualización de los histogramas de cada bloque con su propia memoria compartida, hay que sumarle la actualización de la memoria global realizada posteriormente por un hilo de cada bloque) y teniendo que realizar una sincronización de hilos, aun así, compensa la mejora de rendimiento obtenida con la memoria compartida, convirtiéndose el algoritmo del kernel **histogramShared** en el más óptimo de los cuatro.

- La realidad es que el algoritmo **histogramByBlockShared** es el más rápido de todos, y además es lógico, ya que combina la distribución de las escrituras con la rapidez de la memoria compartida, pero como hay que sumarle el tiempo de las sumas por reducción, finalmente en tiempos absolutos el kernel **histogramShared se queda como el más rápido.**

- Otra conclusión interesante, se obtiene del hecho de que todos los algoritmos mejoran sus tiempos en iteraciones repetitivas, lo que demuestra que parece que hay una optimización y uso de memorias caches.

  Cabe destacar, que la versión de memoria compartida, mejora aún más los tiempos en varias repeticiones, tal vez debido a una mejor optimización en las caches de memoria compartida que la global.