Juan Siverio Rojas

20/01/2021

Proyecto gpu

Contenido

[Objetivo 2](#_Toc62056265)

[Previos 2](#_Toc62056266)

[Descripción 3](#_Toc62056267)

[Kernels Cuda 3](#_Toc62056268)

[(1) Kernel 1 “histogram”: 3](#_Toc62056269)

[(2) Kernel 2 “histogramShared”: Mejora Kernel 1 3](#_Toc62056270)

[(3) Kernel 3 “histogramByBlock” 4](#_Toc62056271)

[(4) Kernel 4 “histogramByBlockShared”: Mejora Kernel 3 5](#_Toc62056272)

[(5) Kernel “sumHistogram” 6](#_Toc62056273)

[Funciones externas. 7](#_Toc62056274)

[(1) Función “calculateHistogramByCpu” 7](#_Toc62056275)

[(2) Función “calculateHistogramByGpu” 7](#_Toc62056276)

[Pruebas 8](#_Toc62056277)

[Con 1 Iteración por algoritmo. 8](#_Toc62056278)

[Análisis de los tiempos obtenidos. 10](#_Toc62056279)

[Tabla Comparativa para 1 iteración 11](#_Toc62056280)

[Con 10 Iteración por algoritmo. 11](#_Toc62056281)

[Tabla Comparativa para 10 iteraciones 11](#_Toc62056282)

[Conclusión 11](#_Toc62056283)

# 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

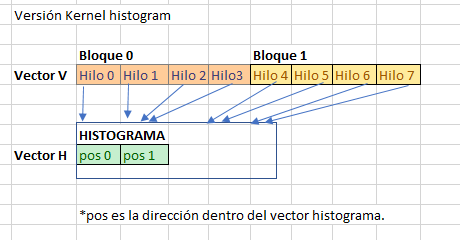
Se adjunta fichero **“histograma.cu”** que contiene toda la programación dividida en las siguientes funciones.

## 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.

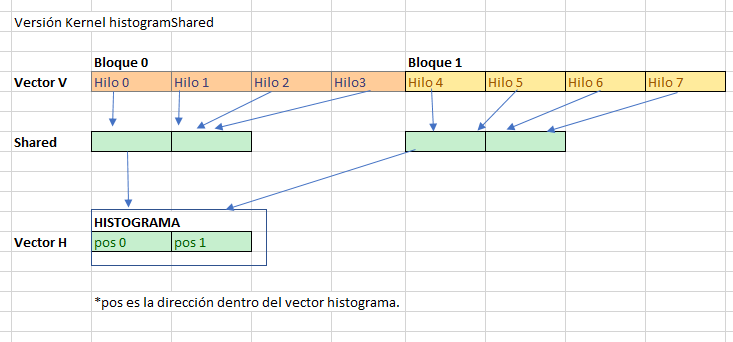


1. Kernel 2 “histogramShared”: Mejora Kernel 1**:** El siguiente Kernel realizará lo mismo que el anterior (Kernel 1) pero con la diferencia de que las operaciones atómicas serán realizadas por los hilos de cada bloque a una memoria compartida **“\_\_shared\_\_”** inicializada a cero por el primer hilo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



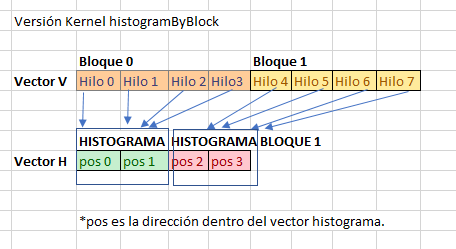
1. Kernel 3 “histogramByBlock”: El siguiente Kernel, funciona mediante la creación y distribución de un histograma por cada bloque. Esto creará tantos vectores histogramas como bloques necesarios para cubrir todo el vector V, por lo que este Kernel proporciona solo una parte de la implementación, puesto 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 numElementsV, int numElementsH)

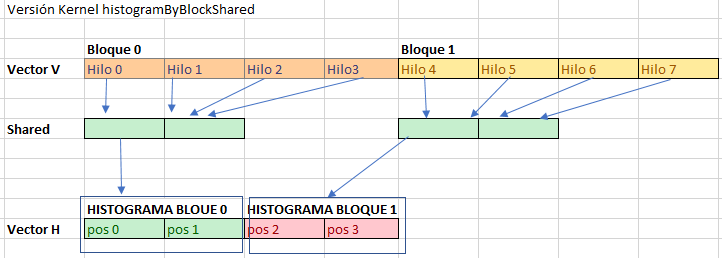
El siguiente esquema, se muestra como es el funcionamiento suponiendo 2 bloques de ejecución con 4 hilos cada uno, para un histograma de 2 elementos.



1. 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.



1. 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.

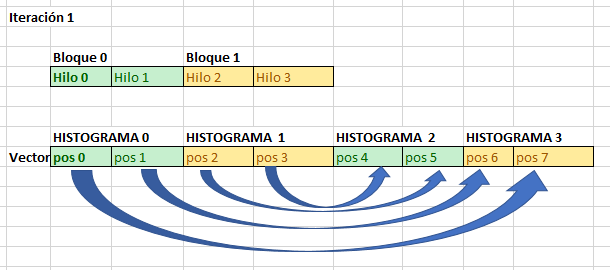
Para el caso concreto de este programa, el vector pasado, será un vector de tamaño **“número elementos histograma”** multiplicado por **“número de bloques que hayan sido necesarios para la ejecución de los kernels 3 ó 4”.**  Estos múltiples histogramas se encontrarán consecutivos en la memoria.

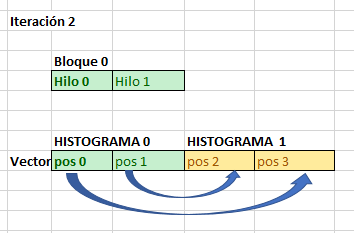
Este kernel se ejecuta con número de bloques igual al número de bloques(que en este caso equivale al número de histogramas) dividido entre 2, puesto que en el método de reducción, con la mitad de los hilos y sabiendo el tamaño del vector, es suficiente.

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 numElementsV, int numElementsH)

Esta función es con fines comparativos.

1. 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 numElementsV, 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;

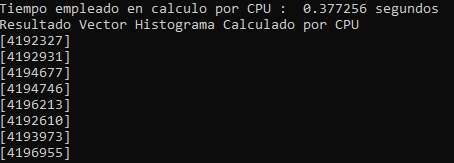
#define NUMELEMENTSH 8;

#define THREADSPERBLOCK 1024;

## Con 1 Iteración por algoritmo.

#define LOOPS 1;

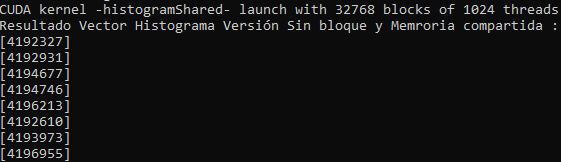
* Tiempo por CPU



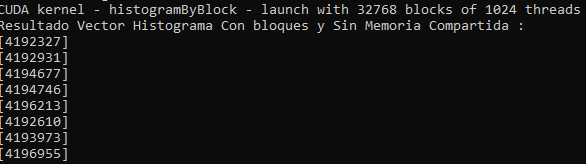
* Ejecución usando Kernel 1 (histogram) Sin bloques y Sin memoria compartida.



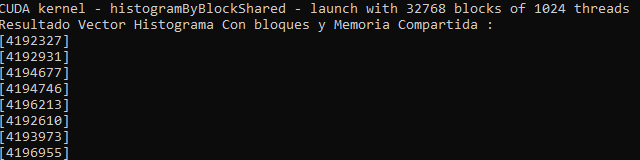
* Ejecución usando Kernel 2 (histogramShared) Sin bloques y con Memoria Compartida



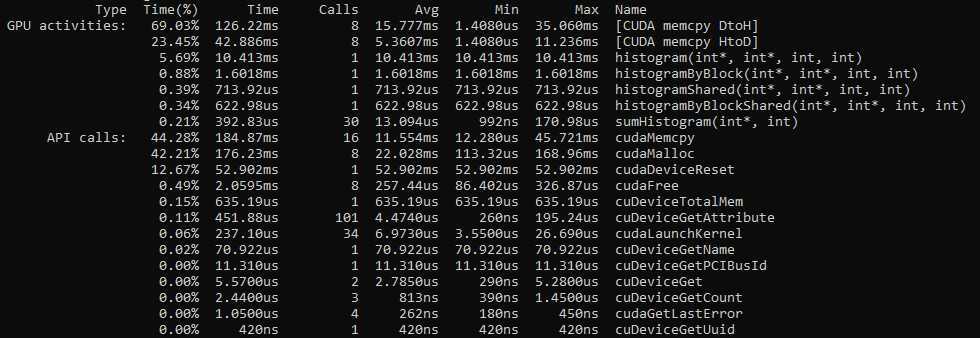
* Ejecución usando Kernel 3 (histogramByBlock) Con bloques y Sin Memoria Compartida



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



### Análisis de los tiempos obtenidos.



* 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.**

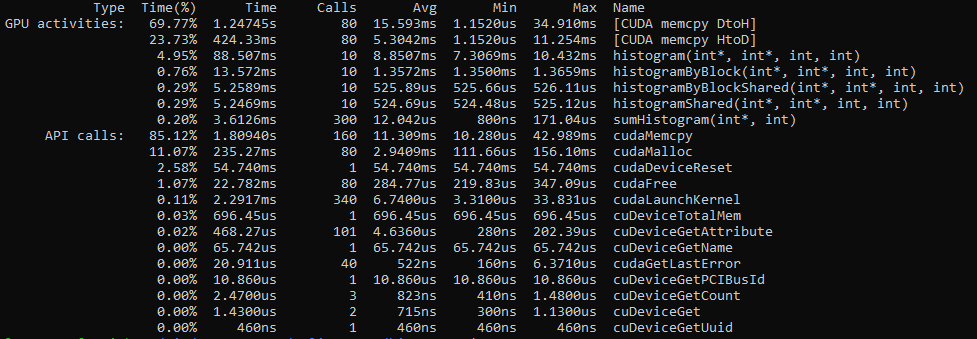
### Tabla Comparativa para 1 iteración

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Kernel** | **Tiempo** | **Mejora con respecto a:**  **histogramByBlock** | **Mejora con respecto a:**  **histogramByBlockShared** | **Mejora con respecto a:**  **histogramShared** |
| **histogram** | **10.4 ms** |  |  |  |
| **histogramShared** | **713 us** |  |  |  |
| **histogramByBlock** | **1.6 ms + 196 us** |  |  |  |
| **histogramByBlockShared** | **622 us + 196 us** |  |  |  |
| **sumHistogram** | **196 us** |  |  |  |

## Con 10 Iteración por algoritmo.

#define LOOPS 10;

Resultado de tiempos.



* En este caso, parece que en las versiones

### Tabla Comparativa para 10 iteraciones

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Kernel** | **Tiempo** | **Mejora con respecto a:**  **histogramByBlock** | **Mejora con respecto a:**  **histogramByBlockShared** | **Mejora con respecto a:**  **histogramShared** |
| **histogram** | **88.5 ms** |  |  |  |
| **histogramShared** | **5.2 ms** |  |  |  |
| **histogramByBlock** | **13.5 ms + 1.8 ms** |  |  |  |
| **histogramByBlockShared** | **5.2 ms + 1.8 ms** |  |  |  |
| **sumHistogram** | **1.8 ms** |  |  |  |

# Conclusión