Maestría en Computo Estadístico
Programación y análisis de algoritmos
Reporte Proyecto CUDA
17 de diciembre de 2020
Enrique Santibáñez Cortés

Introducción

El objetivo de este documento es presentar las comparaciones de los tiempos de ejecución de las operaciones matriciales (suma, resta, multiplicación) programada de forma secuencial y de forma paralela considerando distitos tamaños de las matrices en la arquitectura CUDA (cómputo sobre GPU). Los tiempos de ejecución se consideraron apartir que se ejecuta la función a operar, no se contempla los tiempos para generar las matrices ni el tiempo del menu. Se utilizo Google Colab para escribir y ejecutar código en una GPU.

Explicación de las ideas detras del código.

Se describiran los aspectos más importante para entender el código: la definición de la estructura utilizada, el kernel ocupado, y la explicación de las funciones suma (la resta tiene la misma idea que la suma) y multiplicación matricial.

Estructura matriz.

Definimos una estructura llamada 'matriz' con la que trabajamos en todo el programa. Esta constituida por el tamaño de la matriz la cual utilizamos las variables enteras n y m ($n \times m$), además tenemos una variable de tipo apuntador 'df' que contiene la dirección en donde estaran los elementos de la matriz (ver Figura 1).

```
// structura para una matriz de tamano de n x m.
typedef struct {
 int n; // renglones
 int m; // columnas
 int* df; // datos
} matriz;
```

Figura 1: Estructura matriz.

Notemos que solo ocupamos un apuntador y no un apuntador de apuntadores como se vió en clase, esto por el hecho que al ocupar un apuntador de apuntadores la dirección de cada apuntador no era continua entre sí lo que dificultaba un poco trabajar con estas dirección en el device. Entonces para definir una matriz de $n \times m$ es como si tuvieramos un vector de $1 \times nm$, así ya asegurabamos que las direcciones de los elementos de las matriz fueran continuas. Entonces si tenemos una matriz A de tamaño $n \times m$ y queremos acceder al elemento (i, j), entonces podemos acceder $A.df[i + j * n]$ la razón de esto es que cada m elementos de nuestro vector de $1 \times nm$ representa una columna (utilizando modulos es un poco más sencillo de verse).

Configuración de ejecución

El código a ejecutarse sobre la GPU es lanzado como una malla de bloques de hilos. La estructura esta constituida por Grid's, estos por Block's y estos por Thread's. Mi planteamento al definir esta malla fue crearla como si fuera una matriz, es decir, yo definí un kernel bidimensional en donde solo tenemos un Grid el cual estaría definido por un Bloque (un caso más eficiente sería con algún número de M bloques, pero no pude como acomodar la memoria en device cuando no hay un número exacto de divisiones) con (n, m) hilos repartidos en n filas y m columnas. Este enfoque me permitió identificar cada hilo de una manera parecida a la idea de como se accede a los datos en la estructura matriz, solo que en este caso consideramos las variables blockIdx (índice de bloque dentro de la regilla) , blockDim(número de hilos en el bloque), threadIdx (índice del hilo dentro del bloque.) (NVIDIA Corporation 2018). Pero en este caso es como si estuvieramos en nuestra estructura matriz con $n = threadIdx.x, m = blockDim.x$, entonces como estamos en arreglo bidimensional tenemos que el indentificador de cada hilo sería determinado por $i = threadIdx.x + blockIdx.x * blockDim.x$ y $j = threadIdx.y + blockIdx.y * blockDim.y$. Esto nos permite realizar las operaciones de una manera más eficaz.

Suma (resta) matricial en CUDA

Por como definimos la configuración de ejecución (como si fuera una 'matriz') ya tenemos identificado el hilo en nuestra configuración, por lo que cada hilo estaría representaría un elemento de la matriz. Entonces para sumar dos matrices sería equivalente a sumar los elementos de las matrices que tiene cada hilo, es decir, si sabemos que el hilo tiene el identificador (i, j) entonces sumemos los elementos de las matrices como $resultado.df[j + i * n] = A.df[j + i * n] + B.df[j + i * n]$. Es decir, cada hilo estaría obteniendo la suma de un elemento de la matriz.

Para el caso de la resta es la misma idea.

Multiplicación matricial en CUDA

Tomando una idea parecida a la de la suma, es decir, cada hilo calculará un elemento del producto matricial. Para este caso consideramos que el hilo tendrá qua hacer un bucle para ir sumando cada producto resultante de un elemento de la fila de A por un elemento de la columna de B. Es decir, si el hilo 'esta' en (i, j) este iterará en la fila i de A para multiplicarlo con la columna j de B. El bucle será desde 0 hasta n, y la el producto acumulado se obtiene $suma+ = A.df[k + i * A.n] * B.df[j + k * B.n]$; para que cuando se termine el cuble tengamos el elemento (i, j) de la matriz resultante.

Análisis de los tiempos de ejecución

Suma matricial

Para la comparación de tiempos de ejecución de las funciones las sumas secuencial y paralelizado consideramos matrices cuadradas con tamaños: 100, 1000, ,500, 1500, 5000, 10000, 15000.

Para los distintos tamaños de las matrices anteriores se simulación 2 veces y se obtuvo un tiempo promedio de ejecución. Los resultados son:

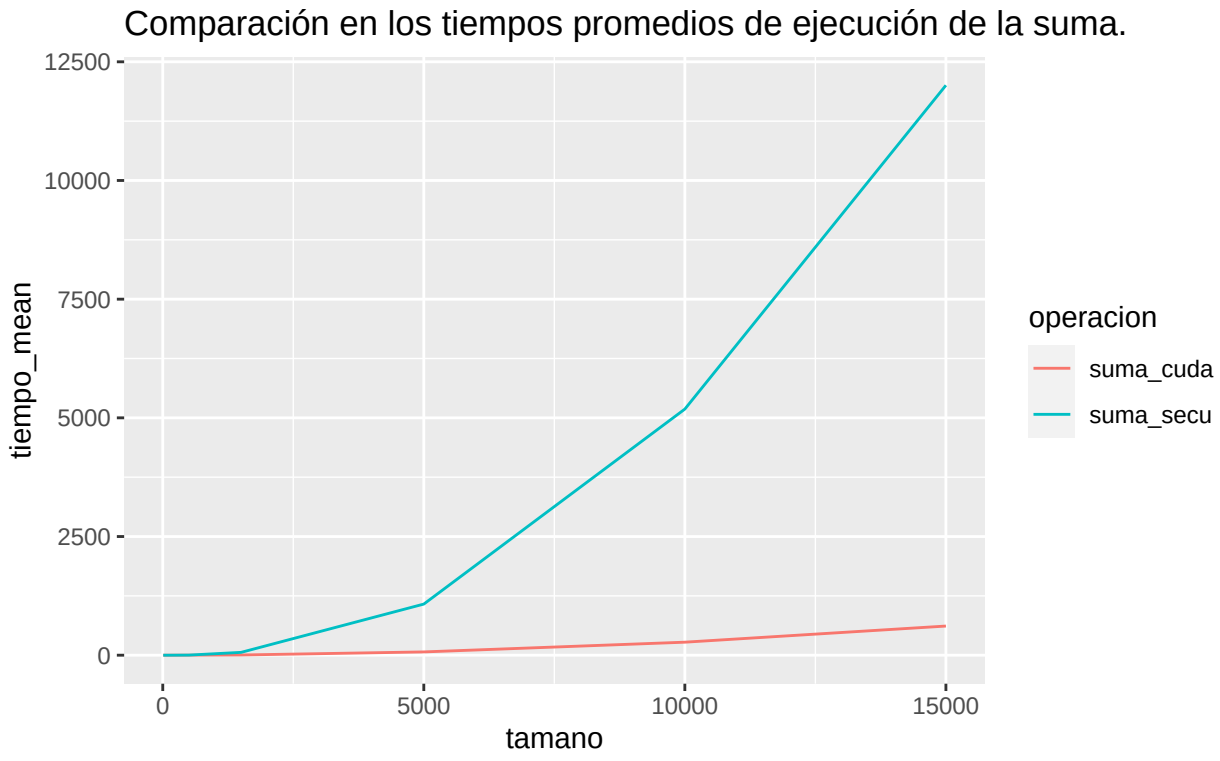
```
library(tidyverse)

datos <- read.csv("simulaciones_cuda .csv")
datos <- datos %>% select(operacion, tamano_matriz_cuadrada, tiempo.ms.,tamano)
datos <- datos[!is.na(datos$tiempo),]
datos$tiempo.ms.<- as.numeric(gsub(",", "", datos$tiempo.ms.))
datos$tamano<- as.numeric(gsub(",", "", datos$tamano))
```

```
datos <- datos %>%
  mutate(tiempo.ms. = ifelse(operacion%in% c("multiplicacion_secu"), (tiempo.ms.)/1000, tiempo.ms

datos_suma <- datos %>%
  group_by(operacion, tamano_matriz_cuadrada, tamano) %>%
  summarise(tiempo_mean=mean(tiempo.ms.)) %>%
  filter(operacion %in% c("suma_secu","suma_cuda"))

datos_suma <- as.data.frame(datos_suma)

datos_suma %>%
  select(operacion, tamano_matriz_cuadrada, tiempo_mean) %>%
  spread(key = "tamano_matriz_cuadrada", value = "tiempo_mean") %>%
  select(operacion, '10x10', '100x100', '500x500', '1500x1500', '5000x5000', '10000x10000', '1500
```

```
##   operacion 10x10 100x100 500x500 1500x1500 5000x5000 10000x10000 15000x15000
## 1 suma_cuda  0.35    0.35    1.05      4.55     69.10      274.50       614.2
## 2 suma_secu  0.00    0.10    2.45     59.60   1076.85     5184.65     12005.5
```

Observamos claramente que cuando el tamaño de la matriz es pequeño no existe diferencia entre los tiempos secuenciales y paralelos, incluso podemos decir que el secuencial es más rápido pero conforme el tamaño de la matriz se vuelve más grande la diferencia en los tiempos de ejecución son mas notorios. Esto se explica muy sencillo, como tenemos nm hilos realizando la suma de dos matrices, si elejimos cada elemento de la matriz tenemos que cada hilo calcula una suma que representa el elemento (i, j) de la suma matricial.

```
ggplot(datos_suma, aes(tamano, tiempo_mean, group=operacion, color=operacion))+
  geom_line()+
  labs(title="Comparación en los tiempos promedios de ejecución de la suma.")
```



Multiplicación matricial

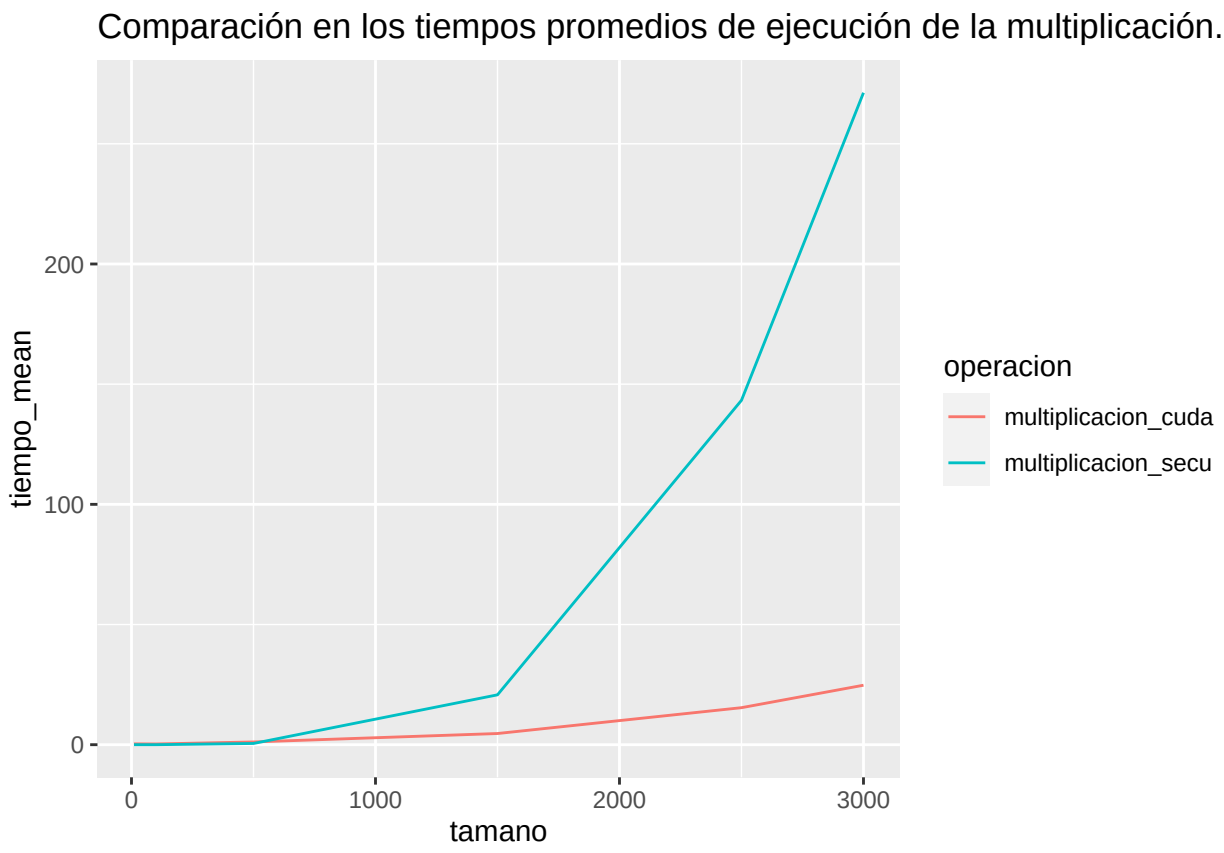
La misma metodología que la suma matricial, tenemos los siguientes resultados:

```
datos_multiplicacion<- datos %>%
  group_by(operacion, tamano_matriz_cuadrada, tamano) %>%
  summarise(tiempo_mean=mean(tiempo.ms.)) %>%
  filter(operacion %in% c("multiplicacion_secu","multiplicacion_cuda"))

datos_multiplicacion<- as.data.frame(datos_multiplicacion)

datos_multiplicacion %>%
  select(operacion, tamano_matriz_cuadrada, tiempo_mean) %>%
  spread(key = "tamano_matriz_cuadrada", value = "tiempo_mean") %>%
  select(operacion, '10x10', '100x100', '1500x1500', '2500x2500', '3000x3000')
```

```
##             operacion 10x10 100x100 1500x1500 2500x2500 3000x3000
## 1 multiplicacion_cuda  0.35 0.30000   4.65000   15.4000     24.75
## 2 multiplicacion_secu  0.00 0.00335  20.75085  143.2944    271.25
```

De igual manera observamos que cuando se ejecuta en CUDA el tiempo de ejecución en menor que cuando se ejecuta en el CPU.

```
ggplot(datos_multiplicacion, aes(tamano, tiempo_mean, group=operacion, color=operacion))+
  geom_line()+
  labs(title="Comparación en los tiempos promedios de ejecución de la multiplicación.")
```



Conclusiones

Se observa claramente la mejoría en los tiempos de ejecución de una GPU al CPU. Cabe aclarar que los tiempos de ejecución comparados no es la manera más eficiente, ya que si observamos nosotros estamos utilizando demasiados hilos tantos como elementos de las matrices introducidas, por lo que si el número de elementos de las matrices es muy grande tanto que el número de hilos de nuestra GPU es menor nuestro programa es inútil. Por lo que recomendaría realizar más particiones en los bloques y no solo ocupar uno como fue el caso.

Bibliografía

NVIDIA Corporation. 2018. "NVIDIA CUDA C Programming Guide." https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf.