Implementación en GPU del método de Jacobi para la restauración de imágenes ruidosas

Jacobo Hernández Varela
Victor Alfredo Valle Hernández,
Daniel Jaime Acosta

Departamento de Matemáticas, Universidad de Guanajuato
CIMAT

Mayo 2020

1 Introducción

El filtrado de imágenes para la eliminación de ruido es elimina aquellos píxeles cuyo nivel de intensidad es muy diferente al de sus vecinos y cuyo origen puede estar tanto en el proceso de adquisición de la imagen como en el de transmisión. Diversos tipos de técnicas pueden ser utilizadas con ese proposito, en el trabajo presente se plantea un problema de optimización que permite suavizar la imagen encontrando una imagen que se aproxime a la anterior al mismo tiempo que minimiza la diferencia entre un pixel y una vecindad dada. Este problema de optimización cudratica es replanteado en un sistema de ecuacion y se utiliza el método de Jacobi para resolverlo. Se presenta una implementación secuencial y una implementacio paralela usando CUDA, finalmente se hace un análisis que compara el rendimiento y la aceleración de las implementaciones.

2 Eliminación de ruido como un problema de optimización

La eliminación del ruido en imágenes puede abordarse mediante varias estrategias, una de las estrategias más conocidas es abordar el problema de buscar una imágen sin ruido como un problema de optimización. Dada una imagen ruidosa g en forma de vector, se plantea el siguiente problema de optimización:

$$\setminus \arg\min_x f(x) = \frac{1}{2}\sum_{i\in\Omega}((x_i - g_i)^2 - \lambda \sum_{j\in N(i)}(x_i - x_j)^2)$$

Donde Ω son los píxeles de la imagen g, $N(i)$ son los vecinos del píxel i y λ un parámetro de regularización. Notemos como el primer término de nuestra

función objetivo es la que conduce la búsqueda a encontrar una imágen muy parecida a g, al mismo tiempo que el segundo miembro penaliza que la imágen x tenga píxeles cuyos vecinos difieran mucho, el parámetro λ tiene un papel importante en la regularización, un mayor valor de λ permite una imagen más difuminada. La función del problema anterior está escrito en forma explicita, una forma de ver dicha función es en su forma matricial:

$$f(x) = xA^T x + gx^t + gg^T$$

donde A se puede ver como una matriz de adyacencia sparse donde los únicos terminos distintos de 0 del renglon i es el término de la diagonal $A_{ii} = 1 + k\lambda$, con k igual al ńúmero de vecinos y $A_{ij} = \lambda$ siempre que i y j sean vecinos. La imágen1 muestra la definición de vecindad para 3 distintos tipos de píxeles caracterizados por el número de vecinos.

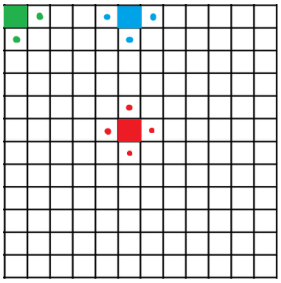

Figure 1: Ejemplos de tipos de píxeles según sus vecinos. Rojos píxeles del interior (4 vecinos), azul píxeles del marco no esquina] (3 vecinos) y verde píxeles de la esquina (2 vecinos)

Para minimmizar f debemos derivar e igual a 0, al hacerlo el problema de optimización se convierte a la resolución de un sistema lineal de ecuaciones dado por:

$$Ax = g$$

Existen diversos algoritmos para resolver un sistema lineal de ecuaciones, entre los más famosos se encuentra Gauss-Seidel y el métode de Jacobi. En el siguiente prufindizaremos más en este método.

3 El método de Jacobi

El método de Jacobi es un método iterativo para resolver un sistema lineal de ecuaciones que utiliza la descomposición LDU de una matriz A. Una condición suficiente para la convergencia del método es que la matriz A sea diagonalmente dominante. En nuestro caso la naturaleza del problema resulta en una matriz

diagonalmente dominante. El método de Jacobi parte de la descomoposición LDU de la matriz A. Dicha factorización permite escribir la matriz A como:

$$A = L + D + U$$

donde L es una matriz triangular inferior, D una matriz diagonal y U una triangular superior. La actualización iterativa del problema está dada por:

$$x_{k+1} = D^{-1}(b - (L + U)x^k)$$

desarrollando esto, nos queda:

$$x_i^{k+1} = \frac{b_i - \sum_j a_{ij} x_j^k}{a_{ii}}$$

En particular para nuestro problema, la regla de actualización estará dada por:

$$x_i^{k+1} = \frac{g_i - \lambda \sum_{j \in N_i} x_j^k}{1 + |N_i \lambda}$$

Notemos como el cálculo de la aproximación en la iteración $k + 1$ depende únicamente de la aproximación en la iteración k, esto facilitara la implementación paralela.

4 Implementaciones

En esta sección describimos brevemente las estrategías seguidas para la implementación serial y paralela del problema descrito en las seciones anteriores. La implementación en secuencial del método es únicamente seguida por la formula iterativa:

$$x_i^{k+1} = \frac{g_i - \lambda \sum_{j \in N_i} x_j^k}{1 + |N_i \lambda}$$

La actualización de la solución tiene una complejidad O(n) en el número de píxeles, ya que la suma que se presenta dentro de cada píxel, al tener cada uno a lo más 5 vecnos permite el calculo eficiente de esta suma, por lo tanto el método puede llegar a comportar de forma $O(n^2)$ hasta converger.
Basándonos en la observación de que cada actualización depende únicamente de la actualización pasada la paralelización del calculo de actualización es fácil de realizar, haciendo que las iteraciones sean hechas secuencialmente. Así pues, en nuestra implementación realizada con CUDA nuestra función kernel se encarga de realizar la actualización de un píxel en particular dentro de la iteración k+1. Para ello dividimos en una malla en bloques de $32 * 32$ hilos que ejecutan una parte de la imagen vista en forma matricial. Al terminar cada iteración, el llamado de la función kernel alterna para que no se hagan copiados innecesario de memoria. Habíamos puesto una función que sincroniza cada llamada a la

función kernel, esto porque pensabamos que las llamadas a la función kernel se hacían de forma asincrona. Sin embargo, en comentarios posteriores a la exposición se nos comentó que dicha llamada no es necesaria, ya que debido a que las llamadas a la función kernel están dentro de un for, el compilador nos las detecta hasta que está dentro de la ejecución por lo que en realidad había una sincronización implicita al terminar cada ciclo.

5 Experimentos

Los resultados obtenidos de las implementaciones realizadas son presentados en esta sección. Primero presentamos el efecto de aplicar el suavizado sobre una imagen borrosa. La imagen original y sus resultados para distintos valores de λ es mostrada en la figura 2. Es claro como el efecto de suavizado se incrementa conforme λ. Estos resultados se hicieron en número maximo de iteraciones de 1000 o hasta convergencia.

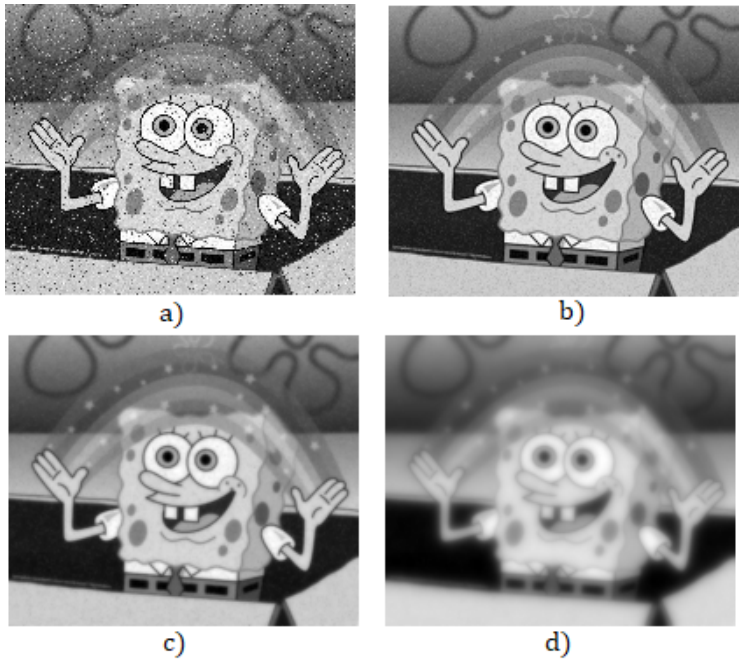

Figure 2: Resultados comparando el efecto de λ. a) Imagen original $\lambda = 0$, b) $\lambda = 1$, c) $\lambda = 10$y d) $\lambda = 100$

Ahora mostraremos distintas graficas que comparan el rendimiento de las distintas implementaciones realizadas. En la figura 3a mostramos una gráfica que compara el tamaño de píxeles de la imágen vs el tiempo, para las dos distintas implementaciones. Observamos como el tiempo del secuencial es mucho

mas grande que el paralelo y cumple con nuestras cotas que dimos para su complejidad. Sin embargo esta gráfica no permite analizar el tiempo del programa en paralelo debido a las enormes diferencias, es por ello que en la Figura 3b mostramos la misma grafica solamente que el tiempo lo ponemos en una escala logaritmica. Vemos como las matrices pequeñas llegan a hacer mas eficientes que la version paralela, pero conforme aumenta el número de datos la velocidad del programa paralelo supera por creces a la versión secuencial.

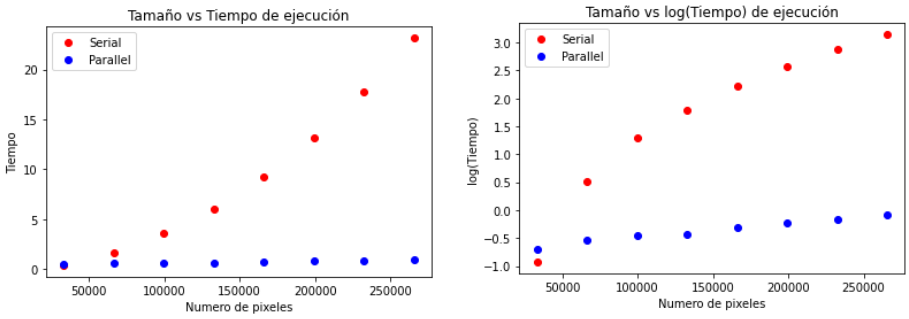

Figure 3: Comparación de tiempo entre implementaciones

Por último mostramos la grafica que muestra el speedup obtenido para distintos tamaños, vemos como claramente se matiene un speedup lineal conforme aumentamos el tamaño de la imagen.

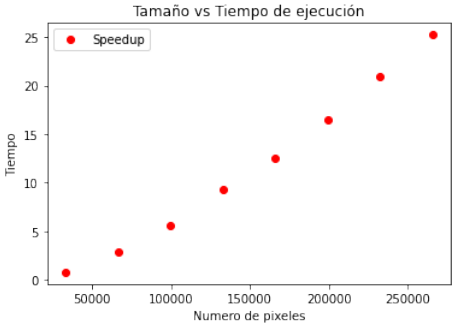

Figure 4: Tamaño vs Speedup)

6 Conclusiones

La implementación de esta estrategía de suavizado mejora enormemente con una implementación paralela en GPU usando CUDA. Obsevamos que en general entre mayor sea el tamaño del problema mas eficiencia se logra de parte del programa en CUDA. Queda como pendiente una ampliación de este algoritmo

a colores y también el uso de memoria compartida entre bloques de hilos que permita un mas rápido acceso a la memoria que probalemente se traduzca en una aceleración.