Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Programación funcional paralela en GPUs

Realizado por
Kenny Jesús Flores Huamán

Para la obtención del título de
Grado en Ingeniería Informática - Tecnologías Informáticas

Dirigido por
Miguel Ángel Martínez del Amor

Realizado en el departamento de
Ciencias de la Computación e Inteligencia Artificial

Convocatoria de Junio, curso 2021/22

Agradecimientos

Resumen

Palabras clave: Computación paralela, CUDA, algoritmos de procesamiento de imágenes, Accelerate, Repa

Índice general

Índice de figuras

1. Introducción

1.1. Contexto teórico

La necesidad de una demanda continua por alcanzar un poder computacional superior como las grandes limitaciones que poseen los sistemas secuenciales, han provocado que la programación paralela haya alcanzado un gran interés dentro del mundo empresarial y en el ámbito de la investigación científica.

Dentro del gigante ecosistema que es la programación paralela uno de los campos más populares es el uso de la aceleración mediante GPU, que consiste en utilizar procesos paralelos para acelerar el trabajo en aplicaciones exigentes, desde IA y análisis de datos hasta simulaciones y visualizaciones[20].

Un ejemplo de uso de la aceleración por GPU es que debido a la gran pérdida de dinero que sufren las instituciones de servicios financieros causado por personas que cometen fraude, estos organismos están utilizando IA y técnicas de aprendizaje automático para detectar fraudes en tiempo real y poder evitar de que ocurran. Como entrenar estos modelos y ejecutar la IA para la detección y prevención de fraudes requiere grandes recursos, muchas corporaciones para conseguir una detección más rápida y una mayor escalabilidad, utilizan los servicios escalables basados en la nube y acelerados por GPU que permitan ejecutar bibliotecas, rutinas y algoritmos de IA/ML optimizados.

Otra aplicación de la aceleración por GPU es en el mundo audiovisual, sobre todo en la suite de Adobe. Por ejemplo, el programa Adobe Premiere Pro utiliza codificación/decodificación acelerada por GPU para acelerar la exportación/reproducción de vídeos, además de el reencuadre automático por IA acelerado por GPU rastrea de forma inteligente objetos y recorta el vídeo horizontar a las relaciones de aspecto adecuadas para las redes sociales, facilitando la producción de contenido en línea[23].

La programación paralela se puede atacar desde distintos paradigmas (imperativo, funcional, entre muchos otros). Pero, en este caso el autor se ha centrado en utilizar el paradigma funcional para atacar el problema de programar de forma paralela. Esto es debido al enorme interés que le causó dar este estándar en la asignatura de Programación Declarativa del grado de I.I Tecnologías Informáticas en la US, además de que la programación funcional fue pensado en parte para la combinarlo con la programación paralela, esto es debido a las enormes propiedades que tienen los lenguajes con este paradigma como por ejemplo, el no producir efectos colaterales y que los datos sean inmutables, ofreciéndonos mucha seguridad, o también la modularidad con la que podemos escribir programas, haciendo que el programa quede mucho más claro, aunque todas estas propiedades las estudiaremos con más detenimiento posteriormente.

De manera que, en este trabajo se pretende realizar un estudio del rendimiento de como funciona la programación paralela en GPUs, comparándolo con la programación paralela en CPUs, utilizando un lenguaje funcional como puede ser en este caso Haskell.

1.2. Motivación

La motivación para llevar a cabo este Trabajo de Fin de Grado ha sido por el enorme interés que tiene el autor por juntar dos de sus grandes pasiones como son el procesamiento de imágenes digitales como el uso de lenguajes de programación funcional, además de ampliar mis conocimientos aprendiendo algo tan importante como es la programación paralela mediante GPU.

Debido a que mis conocimientos sobre programación funcional paralela son nulos, este documento se va a centrar en el estudio de dichos conceptos y se va a dedicar menos tiempo a los algoritmos sobre procesamiento de imágenes, los cuales ya han sido estudiados durante el grado.

1.3. Objetivos

El principal objetivo de este TFG es el estudio, análisis y experimentación del rendimiento, flexibilidad y diseño de algoritmos paralelos desde la programación funcional. En concreto, este trabajo se centrará en las librerías Repa y Accelerate, dos librerías orientadas a la programación paralela dentro del ecosistema del lenguaje de programación Haskell, mediante la implementación de algoritmos para procesamiento de imágenes.

Además de nuestro objetivo principal, tenemos otros objetivos secundarios que nos ayudarán a mejorar como ingeniero informático al que estoy optando ser.

1.3.1. Objetivos técnicos

Estos objetivos están relacionados al aprendizaje de nuevas tecnologías.

- **Objetivo 1:** Conocer y aprender el módulo Repa de Haskell para realizar cálculos de alto rendimiento usando paralelismo en GPU.
- **Objetivo 2:** Conocer y aprender el módulo Accelerate de Haskell para realizar cálculos de alto rendimiento usando paralelismo en GPU.
- **Objetivo 3:** Conocer y aprender los módulos de JuicyPixels Y FFmpeg de Haskell para la lectura y escritura de imágenes y vídeos respectivamente.
- **Objetivo 4:**Poder desempeñar documentos bien estructurados, de una forma clara y entendible utilizando el sistema de composición de textos LaTeX
- **Objetivo 5:** Ser capaz de escribir código eficiente y bien documentado utilizando el lenguaje de programación Haskell.

1.3.2. Objetivos académicos

Los objetivos académicos tienen referencia a los conocimientos teóricos adquiridos.

- **Objetivo 1:** Adquirir un conocimiento sólido sobre la programación paralela.
- **Objetivo 2:** Adquirir un conocimiento sólido sobre procesamiento de imágenes digitales usando programación funcional.
- **Objetivo 3:** Adquirir buenos conocimientos sobre el sistema de composición de textos LaTeX.

1.4. Estructura de la memoria

Esta publicación va a constar de 5 capítulos. Seguidamente, se va a proceder a efectuar una breve descripción de cada uno de las secciones llevadas a cabo en este documento:

1. **Introducción:** En esta primera sección, vamos a dar un contexto al trabajo, expondremos la motivación por la cual se decidió elegir la temática del trabajo, especificaremos los objetivos que se aspiran a lograr y describe los capítulos que forman el documento.
2. **Estado del arte:** Trata los conceptos de una CPU y GPU y computación paralela, además de describir el lenguaje con el que vamos a desarrollar el proyecto como los módulos que hemos utilizado.
3. **Desarrollo del proyecto:** Describe como hemos implementado los diferentes algoritmos de procesamiento de imágenes además de explicar como se han podido leer las imágenes y como se han podido recolectar el rendimiento para poder crear tablas que se expondrán en el capítulo siguiente.
4. **Análisis de rendimiento:** Muestra y discute los resultados obtenidos por las pruebas realizadas durante el desarrollo del proyecto.
5. **Conclusiones y trabajo futuro:** Contiene las conclusiones realizadas durante el trabajo, los problemas que nos hemos encontrado durante el desempeño del trabajo como las líneas en las que se podría seguir trabajando.

Finalmente, vamos a destacar la existencia de un apéndice al final de la memoria, en el cual se presenta un pequeño manual de instalación y de usuario para el perfecto funcionamiento de las pruebas realizadas.

2. Estado del Arte

En este capítulo se hará una breve introducción de todos los conocimientos que hemos utilizado para poder desarrollar el proyecto.

2.1. Paralelismo y Computación Paralela

2.1.1. GPU y CPU

Durante los últimos años ha ido aumentando de manera exponencial la exigencia de carga de trabajo que pretendemos que realicen los ordenadores. Por lo que es importante conocer como funcionan los 2 componentes más relevantes dentro de la mayoría de sistemas actuales: La unidad de procesamiento central (CPU) y la unidad de procesamiento gráfico (GPU)[16].

Una unidad de procesamiento central (CPU) es un componente hardware **esencial** para cualquier sistema computacional moderno debido a que la función que ocupa es interpretar todas las instrucciones de los procesos que necesita el ordenador y el sistema operativo. Su relevancia es tal, que coloquialmente se le conoce como el "cerebro del ordenador".

Por otro lado, la unidad de procesamiento gráfico es un procesador compuesto por muchos núcleos mucho más reducidos y especializados. Este procesador se encarga de realizar el procesamiento de gráficos u operaciones de coma flotante.

Este componente es conocido por el nombre de tarjeta gráfica y su popularidad viene dado por el poder que tiene al renderizar las imágenes de los videojuegos a grandes resoluciones.

Al contrario de lo que piensan las personas que no están metidas dentro de la informática, las tarjetas gráficas no solamente sirven para el mundo de los videojuegos, si no que tiene diversas aplicaciones en muchísimos campos como lo puede ser en el mundo audiovisual, donde las gráficas tienen el poder computacional posible para renderizar vídeos o películas muy pesadas de manera rápida, o también en el terreno de la inteligencia artificial, donde el uso de gráficas nos ayuda a intentar sacar todo el máximo potencial de ejecución de algoritmos de aprendizaje profundo.

Una forma fácil de comprender la diferencia que existen entre los 2 componentes más relevantes de un sistema actual, es comparando cómo se manejan las tareas. De forma arquitectónica, la CPU está compuesta por unos pocos núcleos con mucha memoria caché que están muy bien optimizados para el procesamiento en serie secuencial. Por el contrario, la CPU se compone de numerosos núcleos que pueden manejar miles de subprocesos de manera síncrona[22].

Las GPUs al tener miles de núcleos pueden procesar cargas de trabajos en paralelo de manera muy eficaz.

2.1.2. Paralelismo y Tipos de Paralelismo

El paralelismo es una forma de computo en la que varios cálculos se ejecutan de manera simultánea, siguiendo los principios del algoritmo de Divide y Vencerás (un problema grande se puede dividir en problemas más pequeños, que posteriormente son resueltos simultáneamente)[24].

Tipos de paralelismo:[29]

- **Paralelismo a nivel de bit:** Esto significaba aumentar el tamaño de la palabra del procesador. Este aumento reducía el número de instrucciones que tiene que ejecutar nuestro procesador. Este método se utilizaba mucho pero quedó "estancado" desde la llegada de las nuevas arquitecturas de 32 y 64 bits.
- **Paralelismo a nivel de instrucción:** como un programa es una secuencia de instrucciones ejecutadas por un procesador, esta técnica consiste en reordenar esas instrucciones combinándolos en grupos que posteriormente serán ejecutados en paralelo, sin que cambie el resultado del programa. Esto es posible gracias a que las instrucciones están divididas en diferentes etapas y los procesadores actuales tienen un "pipeline" donde permiten seguir ejecutando una instrucción en una etapa mientras otra instrucción se encuentra en otra fase en un mismo ciclo de reloj.
- **Paralelismo a nivel de datos:** consiste en dividir un conjunto independiente de datos en diferentes subconjuntos, de modo que a cada procesador le corresponda un subconjunto de esos datos.
- **Paralelismo a nivel de tareas:** se caracteriza en la que cálculos completamente diferentes se puedan realizar en cualquier conjunto igual o diferente de datos.

2.1.3. Programación Paralela

Anteriormente hemos visto como se componen los 2 elementos más importantes de nuestros sistemas actuales y también, ya hemos visto la filosofía que sigue el paralelismo. Por lo que la programación paralela es un **paradigma de computación** en la cual usamos de manera simultánea **múltiples recursos** computacionales para poder resolver un problema.

La mayor ventaja de utilizar este paradigma es que podemos resolver problemas que no podríamos resolver en un tiempo razonable. Aunque existen otras virtudes como ejecutar código de una forma más rápida, ofrecernos un mejor balance de rendimiento y costo que la computación secuencial, o ejecutar problemas de orden y complejidad mayor que no podrían realizarse en [21].

Aunque este paradigma suene maravilloso, también tiene sus puntos negativos como ofrecernos un mayor consumo energético, una mayor dificultad para escribir código, y demás.

2.1.4. Programación GPU mediante CUDA

El paradigma de la programación paralela está implementado en diversas librerías y APIs, en la que la mayoría de lenguajes de programación se apoyan para también introducir el concepto de la paralelización. Matrices En virtud de las enormes diferencias entre las arquitecturas de una CPU con una GPU anteriormente explicadas, es necesaria la reformulación de los algoritmos en CPU en el lenguaje de las tarjetas gráficas.

Por fortuna, ahora es mucho más sencillo realizar programación paralela por GPU gracias a APIs como pueden ser OpenCL, CUDA, entre otras plataformas.

OpenCL (Open Computing Language)

OpenCL es una API estándar abierta diseñada para efectuar aplicaciones paralelas de propósito general en diversas unidades de procesamiento como pueden ser CPUs, GPUS, e incluso más específicos, como puede ser una FPGA [13]. Al principio esta API fue creada por Apple, pero posteriormente, le propuso al Grupo Khronos convertirlo en un estándar abierto y libre que no dependa de un hardware de un determinado fabricante[27].

OpenCL especifica un lenguaje de programación basado en C99 que tiene extensiones que son apropiadas para ejecutar códigos de datos paralelos en varios dispositivos.

Gracias a esta API, podemos aprovechar el enorme potencial que tiene la computación paralela en diversas unidades de procesamiento siendo compatible con diversos fabricantes de gráficas.Pero, debido a que está implementada para distintos tipos de dispositivos, la configuración y su interfaz se hacen demasiado compleja que otras APIs con su misma funcionalidad.

Además, esta generalización que tiene OpenCL con respecto a otras APIs más enfocadas a un hardware en epecífico, produce que tenga menos rendimiento que esas librerías. Un caso en especial es el de CUDA, en la que para todos los tamaños de problemas, el rendimiento del kernel de OpenCL es entre un 13 % y un 63 % más lento que el de CUDA[17].

CUDA (Computer Unified Device Architecture)

CUDA es una plataforma de computación paralela desarrollada por NVIDIA para la computación general en unidades de procesamiento gráfico (GPUs). Mediante el uso de CUDA, podremos acelerar nuestros algoritmos aprovechando toda la potencia que tiene una GPU[4].

Para las aplicaciones aceleradas por intermedio de una GPU, la parte secuencial de la carga de trabajo va a ser ejecutada en la CPU, en cambio, las partes de la aplicación donde demandemos un uso intensivo de cómputo, se va a ejecutar de manera simultánea en los cientos de núcleos que tiene una GPU de NVIDIA.

CUDA nos permite codificar algoritmos en GPUs de Nvidia por medio de una variación del lenguaje de programación C (CUDA C), aunque por medio de wrappers

podemos utilizar utiliar lenguajes de programación populares como puede ser Python, Julia, Fortran y Java en vez de usar el lenguaje propio de CUDA.

Dentro del ecosistema de los lenguajes de programación funcional, Haskell cuenta con Accelerate, una de las librerías más agradables para el procesamiento acelerado de matrices, que puede ejecutarse en CUDA.

2.1.5. Aplicaciones de programación paralela en GPUs

2.2. Haskell

Haskell es un lenguaje de programación funcional perezoso. El proyecto empezó a desarrollarse a finales de los años 80 por un comité de investigadores cuyo objetivo era definir un estándar abierto para los lenguajes funcionales. Los resultados del comité dieron su fruto cuando a finales del 97 se desarrolló **Haskell 98**, el estándar del lenguaje Haskell, y durante estos años el lenguaje ha ido evolucionando hasta producir **Haskell 2010**, que es el estándar actual de Haskell[14].

Como hemos visto anteriormente, Haskell es considerado un lenguaje funcional, esto significa que en vez de obtener resultados dándole al ordenador una sucesión de tareas que posteriormente ejecutará como realizan los lenguajes imperativos, en este tipo de paradigma describe qué/cuál es la solución del problema[19].

Para entender esto de una manera más clara vamos a poner un ejemplo gastronómico: los lenguajes imperativos proporcionan la receta que debes seguir para preparar el rico pastel de chocolate con helado que quieres comerte, mientras que los lenguajes declarativos te suministran fotos del pastel preparado.

2.2.1. Características principales de Haskell

Haskell tiene estas características muy interesantes:[28]

- **Puro:** En Haskell las variables son inmutables, es decir, no modifican su valor. Al existir esta inmutabilidad o integridad referencial, produce que no puedan existir efectos colaterales, en consecuencia, los programas funcionales son propensos a tener muchos menos errores que los imperativos.
- **Perezoso:** Esto significa que no se ejecutarán funciones ni se van a calcular resultados mientras no seas necesario. Esto hace que podamos construir listas infinitas de elementos sin caer en cálculos infinitos que bloqueen la máquina.
- **Fuertemente tipado:** en este lenguaje es imposible convertir, si no es explícitamente con funciones de conversión, entre diferentes tipos de datos, por lo que al no convertir sin querer un Int en un Float, producirá que a la larga se produzcan menos errores. Esta característica puede parecer tediosa por las operaciones adicionales que debes hacer para convertirlo al tipo de dato que quieres, pero, en

opinión del autor puede ser más engorroso mantener un sistema flexible de tipos para todo.

Haskell nos proporciona un sistema de tipos muy elaborado que nos puede ayudar a deducir inteligentemente como se usan las variables e inferir que tipo de dato debe tener la variable.

- **Modular:** La programación modular consiste en dividir un programa en módulos con el fin de hacer el programa más legible, más sencillo para encontrar errores y permitirnos reutilizar módulos sin tener que crearlos de nuevo. Haskell nos permite utilizar este paradigma debido a que todo programa escrito en Haskell, es un conjunto de módulos.

2.3. Programación paralela de datos con REPA

Repa es una librería escrita en Haskell que nos permite realizar operaciones sobre matrices utilizando paralelismo sobre CPU. Repa significa "REgular PArallel arrays" y el significado de matriz regular viene de que los arrays son densos, rectangulares, y cada elemento de una matriz Repa tiene que ser del mismo tipo.

En esta sección vamos a explicar cómo funciona Repa por dentro, las funciones principales predefinidas por la librería y ver como funcionan las convoluciones dentro de Repa

2.3.1. Matrices, Dimensionalidad e índices

Representación abstracta de una matriz

Las matrices en Repa tienen la siguiente estructura:

Array r sh e

Donde el parámetro r representa el tipo de representación que tiene la matriz, sh representa el tamaño y la dimensionalidad que tiene la matriz y el tipo e significa el tipo de dato que se guarda en cada elemento de la matriz. Un ejemplo para entender esto sería la siguiente matriz:

Array U (Z :. 3 :. 3) Int

En este caso nos encontramos con una matriz bidimensional de tamaño 3x3, donde cada elemento de la matriz tiene que ser de tipo Int. La U viene de unboxed, y significa que cada valor de la matriz va a tener que ser evaluada y guardada. El tipo U es la representación más eficiente si estamos trabajando con datos numéricos.

Matrices de manifiesto vs Matrices retrasadas

Para usar de manera eficiente esta librería es importante entender la diferencia que tienen las matrices de manifiesto (**manifest arrays**) y las matrices retrasadas (**delayed arrays**).

Las matrices de manifiesto tienen la característica de que el valor concreto de cada elemento ha sido calculado y guardado en memoria. Un ejemplo de este tipo de matrices es el tipo **Unboxed** o **U**:

Array U (Z :. 10) Int

Una cosa a tener en cuenta es que cuando el array sea de la dimensión que sea se guarde en memoria, va a ser guardado como un vector.

En cambio, una matriz retrasada se representa mediante funciones que se calculan cada vez que se necesita acceder a un elemento. En consecuencia, al no tener que estar generando una matriz real cada vez que realizamos una operación, puede ser buena idea transformar nuestra matriz de manifiesto a una matriz retrasada.

Las matrices retrasadas se representan mediante el tipo Delayed o **D**. Un ejemplo de este tipo de matrices puede ser el siguiente:

Array D (Z :. 10) Int

Para poder transformar nuestros arreglos retrasados a matrices de manifiesto, Repa nos proporciona las funciones monádicas de computación **computeP** para evaluar los elementos del array de manera paralela y **computeS** para evaluarlos de manera secuencial.

Algunas funciones básicas que nos permite realizar Repa pueden generar una matriz retrasada como salida, por lo cual habrá que convertirlo posteriormente a un arreglo de manifiesto cuando queramos utilizarlo. Uno de los ejemplos más conocidos sería la función map, que tiene el siguiente tipo:

```
map :: (Shape sh, Source r a) => (a -> b) -> Array r sh a ->
   Array D sh b
```

La función map aplica a cada elemento de la matriz una función dicha por parámetros. Pero lo más importante es que da igual el tipo de matriz que le pasemos, siempre va a devolver una matriz retrasada.

Dimensionalidad e índices

En Repa podemos utilizar el tipo Shape para poder indexar una matriz. Por ejemplo si consideramos una matriz bidimensional de tamaño 3x3 la matriz podría ser de la siguiente forma:

Array U (Z :. 3 :. 3)

Donde Z representa la dimensión 0, el primer 3 representa la longitud de la primera dimensión y el siguiente 3 representa la longitud de la segunda dimensión.

Si queremos acceder al primer elemento de esta matriz, deberemos utilizar la siguiente función:

```
(!) :: Shape sh => Source r e => Array r sh e -> sh -> e
```

Es importante recalcar que cuando queremos acceder a cualquier elemento de la matriz, el primer elemento que se empieza a contar es desde el cero. por lo cual, si

queremos acceder al primer elemento de una matriz 3x3 deberemos acceder al (Z :. 0 :. 0):

```
> let arr = fromListUnboxed (Z :. 3 :. 3) [1..9] :: Array U
   DIM2 Int
> arr ! (Z :. 0 :. 0)
1
```

Como sabemos que como se guarda internamente una matriz en memoria es independiente de su forma, incluso podemos cambiar la forma sin tener que copiar la matriz utilizando la función reshape:

```
reshape :: (Shape sh1, Shape sh2, Source r1 e) => sh2 ->
   Array r1 sh1 e -> Array D sh2 e
```

2.3.2. Generando arrays

Como hemos visto en ejemplos anteriores, podemos generar matrices de manifiesto a partir de listas gracias a la función **fromListUnboxed**:

```
arr = fromListUnboxed (Z :. 9) [1..9] :: Array U DIM1 Int
```

En el primer argumento especificamos el tamaño y dimensionalidad de nuestro arreglo y posteriormente, la lista de valores que tomará nuestro arreglo

Y si lo que queremos es crear directamente matrices retrasadas, podemos realizarlo mediante la función **fromFunction:**

```
arr = fromFunction (Z:.5) (\(Z:.i) -> i*2 ::Int)
```

El primer argumento toma el tamaño y dimensionalidad que queremos que tenga nuestro arreglo y el segundo parámetro es la función con la que va a crear cada elemento de nuestra matriz retrasada.

Funciones básicas

Repa contiene muchas funciones para realizar operaciones con arreglos, por lo que en este apartado recapitularemos las funciones a considerar por el autor:

- **map:** Dados una función y una matriz de repa, construye una matriz retrasada aplicando la función a cada elemento del array.
- **zipWith:** Combina 2 matrices elemento a elemento, aplicando una función a la combinación de esos 2 elementos. Si la extensión de las 2 matrices difiere, la extensión de la nueva matriz será su intersección.

2.3.3. Convoluciones mediante Repa Stencil

El papel de los stencils es fundamental en la realización de muchas aplicaciones. Los stencils pueden servir para resolver ecuaciones diferenciales parciales (PDEs) sobre cuadrículas regulares hasta el cálculo de convoluciones en el campo del procesamiento de imágenes. Esto ha llevado a que en Repa se añadieran nuevos módulos como nuevos tipos de representaciones para poder permitir el uso de estos cálculos.

Como los stencils solamente dependen de un pequeño subconjunto de valores de una matriz, podemos realizar estas operaciones de forma paralela para poder conseguir un mejor rendimiento en nuestros algoritmos.

Pero antes que nada tenemos que saber exactamente que es un stencil y como funciona.

¿Qúe es un stencil?

Un stencil es una función map en la que el valor correspondiente y sus vecinos se multiplican por unos coeficientes de convolución y luego todos esos valores se suman produciendo un valor escalar por cada valor de entrada.

Para un mejor entendimiento pondremos el siguiente ejemplo:

Si consideramos una matriz de entrada M de dimensión 3x3:

$$\mathbf{M} = \begin{bmatrix} t1 & t2 & t3 \\ l & c & r \\ b1 & b2 & b3 \end{bmatrix} \tag{2.1}$$

Y una matriz K cuyos coeficientes de convolución son:

$$\mathbf{K} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{2.2}$$

Entonces, la función stencil sería la siguiente:

$stencil(M, k) = t1 \cdot a + t2 \cdot b + t3 \cdot c + l \cdot d + c \cdot e + r \cdot f + b1 \cdot g + b2 \cdot h + b3 \cdot i$

Como ya hemos podido comprobar con el ejemplo anterior, los stencils son multidimensionales, como también también existen diferentes tipos de vecindarios.

A continuación, veremos como se pueden llevar todos estos conceptos al ecosistema de Repa.

Especificando stencils

Simon Peyton Jones junto 2 investigadores de la Universidad de Nueva Gales del Sur a la hora de implementar la convolución paralela de stencils en Repa, decidieron definir una función que mapee un índice relativo al coeficiente que pertenece a ese índice.

Si quisieramos generar el stencil de laplace mostrado en la figura 2.3 , tendríamos que escribirlo de la forma escrita en esta 2.1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.3}$$

```
laplace :: Stencil sh a
laplace = makeStencil (Z :. 3 :. 3)
$ \ix -> case ix of
Z :. 0 :. 1 -> Just 1
Z :. 0 :. -1 -> Just 1
Z :. 1 :. 0 -> Just 1
Z :. -1 :. 0 -> Just 1
_ -> Nothing
```

Extracto de código 2.1: Generación de un Laplace Stencil

Pero, otra forma más sencilla de escribir esta sintaxis es utilizando el pragma QuasiQuotes. Por lo que para poder generar el mismo stencil anterior, solamente tienes que escribir la siguiente función:

```
laplace :: Stencil sh a
laplace = [stencil2|  0 1 0
                      1 0 1
                      0 1 0 |]
```

Extracto de código 2.2: Stencils usando QuadiQuotes

2.4. Programación paralela con GPUs mediante Accelerate

Accelerate es un lenguaje específico de dominio incrustado (eDSL) desarrollado principalmente por Trevor L. McDonnell para poder realizar operaciones matriciales empleando paralelismo sobre GPU. Esto nos permite hacer uso de la enorme potencia que tienen las GPU escribiendo el código en Haskell y poderlo ejecutar directamente en la GPU sin tener que escribir código CUDA[1].

El módulo Accelerate nos proporciona una infraestructura básica que incluye Data.Array.Accelerate, un módulo para construir cálculos de matrices como un Data.Array.Interpreter para poder interpretar los cálculos en Haskell.

Para poder ejecutar realmente una computación en Accelerate en una GPU, en primer lugar necesitamos una GPU compatible con la tecnología CUDA de NVIDIA, un sistema operativo compatible con el compilador LLVM, y el módulo accelerate-llvm-ptx.

Si quisieramos ejecutar una computación en Accelerate paralelizando CPU en vez de la tarjeta gráfica, podemos realizarlo gracias al módulo accelerate-llvm-native.

Características principales de Accelerate:

- **Interfaz abstracta**: Los tipos que representan los cálculos de matrices solo se exportan de manera abstracta, es decir, el usuario puede ejecutar cálculos de matrices, pero no puede inspeccionar esos cálculos. Esto es así para permitir mayor flexibilidad para futuras extensiones del módulo.
- **Lenguaje estratificado**: Este módulo distingue los tipos de operaciones colectivas **Acc** del tipo de operación escalar **Exp** para obtener un lenguaje estratificado. Las operaciones colectivas comprenden muchos cálculos escalares que se ejecutan en paralelo, pero a la inversa no puede ocurrir. Esta separación excluye estáticamente el paralelismo de datos anidados e irregulares, en cambio, Accelerate se limita solamente al paralelismo de datos que involucra solo matrices multidimensionales regulares.
- **Optimizaciones**: Accelerate utiliza una serie de optimizaciones escalares y de matrices, incluida la fusión de matrices para mejorar el rendimiento de los programas. La fusión de un programa implica combinar bucles sobre una matriz en un solo recorrido, lo que reduce el tráfico de memoria y elimina matrices intermedias.

2.4.1. Arrays e índices

Tanto la librería Repa como Accelerate, tienen la característica de que son librerías especializadas para programar con matrices. Por lo que una computación en Accelerate toma uno o más matrices y entrega uno o varios arreglos. Sin embargo, la diferencia con la librería Repa es que su estructura de datos tiene solamente 2 parámetros:

Array sh e

Donde el tipo sh, representa el tamaño y la dimensionalidad que tiene el arreglo y el parámetro e es el tipo de dato que se almacena en cada elemento de la matriz. A diferencia de la librería Repa, las matrices no están etiquetadas explícitamente con un tipo de representación, aunque internamente existan las matrices retrasadas y las operaciones se fusionan de la misma manera que en Repa.

La dimensionalidad y los índices usan el mismo tipo de datos que la librería Repa, es decir, que son construidos como listas utilizando solamente Z y $(Z : .)$:

```
data Z = Z
data tail :. head =  tail :. head
```

Y a continuación algunos sinónimos de las dimensiones más comunes dentro de la librería Accelerate:

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

```
type DIM3 = DIM2 :. Int
```

Debido a que los arreglos de dimensión cero, uno y dos son muy comunes dentro del módulo, podemos encontrar algunos sinónimos muy usados:

```
type Scalar = Array DIM0
type Vector = Array DIM1
type Matrix = Array DIM2
```

Para poder acceder a un elemento de una matriz, deberemos utilizar una función u otra dependiendo de que tipo de matriz tengamos. Si nos encontramos con una matriz sin acelerar, debemos utilizar el siguiente método:

```
indexArray :: (Shape sh, Elt e) => Array sh e -> sh -> e
```

Al igual que en Repa, cuando queremos acceder al primer elemento de una matriz, el primer elemento que se empieza a contar es desde el cero. Por lo cual, si queremos acceder al último elemento de una matriz 5x5 deberemos acceder al $(Z : ,4 : ,4)$:

```
> let arr =  fromList (Z :. 5 :. 5) [1..] :: Array DIM2 Int
> arr
> Matrix (Z :. 5 :. 5)
  [  1,  2,  3,  4,  5,
     6,  7,  8,  9, 10,
    11, 12, 13, 14, 15,
    16, 17, 18, 19, 20,
    21, 22, 23, 24, 25]
> indexArray arr (Z:.4:.4)
  25
```

En cambio, si queremos acceder un elemento de un arreglo en el contexto de cálculo de Accelerate, deberemos usar el operador (!):

```
(!) :: forall sh e. (Shape sh, Elt e) => Acc (Array sh e) ->
   Exp sh -> Exp e
```

2.4.2. Generando arrays

Al igual que en la librería Repa, existen funciones para construir arreglos sin acelerar a partir de listas, a partir de otra función o incluso puedes generarlos a partir de otras estructuras de datos como puede ser una matriz Repa o una imagen BMP.

A continuación vamos a enseñar las funciones que nos permiten generar arrays a partir de listas o a partir de una función:

```
fromFunction :: (Shape sh, Elt e) => sh -> (sh -> e) -> Array
    sh e

fromList :: forall sh e. (Shape sh, Elt e) => sh -> [e] ->
   Array sh e
```

En cambio, si quisieramos generar un arreglo en el contexto de cálculo de Accelerate, podemos utilizar la función generate:

```
generate :: forall sh a. (Shape sh, Elt a) => Exp sh -> (Exp
   sh -> Exp a) -> Acc (Array sh a)
```

Donde el primer parámetro indicamos el tamaño que va a tener nuestro arreglo, y el segundo parámetro le indicamos la función para que calcule cada elemento del arreglo.

2.4.3. Convoluciones mediante Accelerate Stencil

De la misma forma que Repa puede hacer convoluciones de stencils de forma paralela, Accelerate también tiene su propia implementación para poder realizar este tipo de calculos.

Para aquellas posiciones de la matriz en la que la vencidad se extiende más allá de los límites de la matriz de origen, declararemos una condición para determinar el contenido de las posiciones de vecindad fuera de los límites.

Especificando stencils

A diferencia de la librería Repa, los vecindarios del stencil se especifican a través de n-tuplas anidadas, donde la profundidad del anidamiento es igual a la dimensionalidad de la matriz.

Un ejemplo de esto podría ser la creación de un stencil laplace de dimensión 3x3 para una matriz bidimensional2.3:

```
laplace :: Stencil3x3 a -> Exp a
laplace ((_,t,_)
        ,(l,c,r)
        ,(_,b,_)) = t+l+r+b
```

Extracto de código 2.3: Laplace Stencil en Accelerate

Siendo c el punto focal del stencil, y el resto de valores son su vecindario, por lo que para crear el kernel laplace, solamente hay que sumar los valores que necesitemos. Esto hace que se haga una forma muy fácil de desarrollar sin tener que usar pragmas.

Especificación de condiciones de contorno en Stencils

Cuando nos encontramos con elementos que se escapan fuera de los límites de nuestro arreglo, tenemos que declarar una condición para determinar que valores van a tomar cada elemento fuera del array. En Accelerate, nos encontramos con el tipo de datos ***Boundary*** que nos va ayudar a poder especificar esa condición de manera muy simple.

A continuación, vamos a explicar las diferentes formas de especificar una condición en los Accelerate stencils:

- **Clamp:** Trata a los puntos fuera de la matriz con el mismo valor que el pixel borde.

  Por ejemplo, si nos encontramos con un stencil de dimensión 3x3, el elemento que está fuera del contorno *b*, va a tener el valor de la posición *c*.

```
  +------------+
  |a           |
 b|cd          |
  |e           |
  +------------+
```

- **Mirror:** Esta especificación trata a los puntos más allá de la entensión de nuestro arreglo como el reflejo de la matriz.

  Para entenderlo, podemos tener como ejemplo un stencil 5x3, en la cual el elemento fuera del arreglo *c* va a tener el valor de *d*, y el valor fuera del contorno *b* va a tener el valor de la posición de *e*:

```
  +------------+
  |a           |
bc|def         |
  |g           |
  +------------+
```

- **Wrap:** Este tipo de condición trata a los puntos que se van de la extensión de nuestra matriz, como valores envueltos de la matriz.

  Para dejarlo más claro, pondremos un ejemplo de un stencil 3x3, donde los elementos fuera de los límites se van a leer como el patrón realizado a la derecha.

```
 a bc
  +------------+       +------------+
 d|ef          |       |ef          d|
 g|hi          |  ->   |hi          g|
  |            |       |bc          a|
  +------------+       +------------+
```

- **Function:** Especificación que trata a cada elemento fuera del contorno como el resultado de aplicar una función al índice fuera de los límites, permitiendo poder especificar diferentes condiciones de contorno en cada lado.

3. Desarrollo del Proyecto

En este capítulo describiremos como se han implementado los distintos algoritmos realizados en los 2 módulos que hemos utilizado.

En primer lugar, se ha decidido que vamos a trabajar únicamente con algoritmos de procesamiento de imágenes digitales con esquema de color RGB de 8 bits cada canal. Además de que exceptuando el cálculo de histogramas, el resto de algoritmos deben recibir como parámetro de entrada, al menos, una matriz de tipo *coma flotante* o *float*.

También, se decidió utilizar Repa como contrintante de Accelerate porque es la librería fundamental dentro de los módulos de paralelismo por CPU.

El desarrollo de los algoritmos ha consistido en primer lugar, en implementar un método para poder leer imágenes tanto en Repa como en Accelerate desde distintos tipos de formato (jpg,png,entre otros), esto es debido a que ambas librerías por defecto, solamente pueden leer formatos bmp (Windows bitmap). Posteriormente, se implementó cada uno de los algoritmos de procesamiento de imágenes.

Por otra parte, se ha desarrollado un sistema para poder recolectar la información del tiempo de ejecución de cada algoritmo para posteriormente en el siguiente capítulo, poder analizar las pruebas de rendimiento de los distintos cálculos.

3.1. Lectura de imágenes

Por más que Accelerate Y Repa cuenten con funciones de lectura de imágenes, éstas solamente aceptan las que tengan el tipo de formato BMP, por lo cual, para poder leer cualquier tipo de formato de imagen, se ha recurrido al uso de la librería JuicyPixels para realizar estas tareas.

Se ha acudido al uso a esta librería entre las diversas que existen debido a que es muy sencilla de usar, está íntegramente escrita en Haskell, por lo que no tenemos que preocuparnos de instalar herramientas externas,permite leer y escribir imágenes desde diversos formatos, y también que existen wrappers para transformar imágenes de JuicyPixels a otras librerías de manera simple.

3.1.1. Lectura de imágenes en Repa

Para la realización de conversión entre JuicyPixels y Repa se realizó un módulo para tener un mayor control de las estructuras que quería a fin de la realización de los algoritmos, además de aprender como funcionaba internamente Repa.

3.1.2. Lectura de imágenes en Accelerate

En primer lugar, se pensó utilizar el módulo *accelerate-io-JuicyPixels* para poder realizar la transformación de JuicyPixels a Accelerate y viceversa, pero, como el tipo de datos que nos devolvía a Accelerate era muy complejo de utilizar y los algoritmos estaban implementados para datos de coma flotante, se decidió hacer la función de conversión a Accelerate de forma manual.

No obstante, para poder transformar nuestras matrices aceleradas a imágenes Juicy con cualquier tipo de formato, nos ayudamos del módulo anteriormente mencionado, en virtud de que las matrices en Accelerate no son del tipo *Word*, si no que son *Exp Word*, produciendo muchos problemas al intentarse realizar de manera manual.

3.1.3. Funciones de conversión de tipos de datos

Cuando se transforman las imágenes de JuicyPixels a alguno de los diferentes módulos que hay, los valores de la matriz se cargan con un tipo de datos llamado Pixel8, que es un sinónimo del Word8.

Sin embargo, todos los algoritmos de procesamiento de imágenes se han realizado con tipos de datos Int o Float, por lo que se ha requerido desarrollar funciones para convertir de Pixel8 a un tipo de datos numérico que acepte nuestro algoritmo y viceversa.

Además, también se decidió mantener el tipo de datos Pixel8 debido a que JuicyPixels solo acepta ese tipo de datos para construir sus imágenes RGB. Si queremos construir una imagen en escala de grises solamente necesitamos una matriz de tipo float.

3.2. Generación de Histogramas

En el contexto de procesamiento de imágenes, el histograma de una imagen es una distribución de valores en escala de grises que muestra la reiteración de ocurrencia de cada valor de nivel de gris[8].

Como hemos considerado imágenes RGB8, el histograma de una imagen RGB va a ser la terna de 3 histogramas (uno por cada banda) donde cada una de ellas va a almacenar píxeles que varían de 0 a 255.

Histogramas en Repa

La versión inicial de la generación de histogramas en Repa fue desarrollada de forma secuencial inspirándome en algoritmos desarrollados por la comunidad, a causa de que el autor todavía estaba iniciando con este módulo.

Esta versión de prueba funcionaba con algunas imágenes, pero en otras con un rango de colores más grande tardaba una eternidad. Esto es debido a que para la creación del algoritmo, tenía que realizar la función traverse 1 vez por cada pixel, y como tenemos 3 bandas (debido a que estamos trabajando en el modelo de color RGB), pues tendremos que aplicar esa misma función 3 veces por cada píxel, lo que conduce a que sea un código super ineficiente.

Después de leer a varias personas con el mismo problema a la hora de realizar histogramas en Repa y de intentos fallidos de intentar integrar este algoritmo a este módulo, decidí hacer caso a la comunidad y utilizar una estructura de datos diferente a los arreglos de Repa como acumulador, que posteriormente si hace falta, se puedan transformar de nuevo a Repa.

Para la realización de estas nuevas versiones de histogramas, decidí usar como acumulador 2 estructuras de datos distintas:

- **Sequence:** es una estructura de datos que se basa internamente en *finger trees* (una estructura de datos puramente funcional, que nos permite construir de manera eficiente otras estructuras de datos funcionales como secuencias, árboles de búsqueda entre muchos otros[15]), lo que significa que es un tipo de datos puramente funcional.

  Las características que me hicieron querer probar esta estructura de datos fue que las secuencias admiten una indexación rápida, además de al querer editar esta estructura de datos, generalmente evita copiar la secuencia entera, reproduciendo solamente la parte que han cambiado[6].

- **Vector:** es un módulo para matrices polimórficas capaces de contener cualquier valor de Haskell. Este paquete admite una rica interfaz de operaciones tipo lista y operaciones masivas de matrices.[18]

  Además de que sus indexaciones son de tiempo O(1), lo que me hizo decantarme por probar esta librería como estructura interna es que es la base de muchas otras librerías de matrices, debido a que sus funciones están bien documentadas y también, porque existe un módulo donde se pueden convertir vectores del paquete *Vector* a vectores Repa con un tiempo 0(1), por lo que me pareció interesante esa funcionalidad.

Esta nueva primera versión de generación de histogramas, decidimos utilizar como acumulador la estructura de datos *Sequence*, y los cálculos se realizaron de manera secuencial. Posteriormente, se decidió realizar una segunda versión usando la misma estructura de datos, pero haciendo el cálculo del histograma de cada banda en paralelo, por lo cual esto solo llevaría a realizarlo 3 veces más rápido.

Por otro lado, desarrollé esta tercera versión utilizando como acumulador de datos el módulo *Vector*. Para la ejecución de este algoritmo primero se hace el cálculo de cada fila de la imagen en paralelo, además de realizarlo para cada canal en paralelo. De esta forma, si nos centramos en un canal, obtenemos tantos histogramas como filas tenga la imagen, después quedaría realizar la suma de cada entrada de los histogramas para obtener un histograma final por cada banda. La operación suma de histogramas es paralelizable, debido a que es un reduce.

Histogramas en Accelerate

La implementación de generación de histogramas en accelerate se compone de 2 funciones, una de ellas genera un histograma por cada banda en paralelo, y la otra función genera un histograma a partir de una banda en donde los elementos del arreglo los realiza el histograma paralelizando los elementos de la matriz.

3.3. Escala de grises

Una imagen en escala de grises es aquella en el valor que que contiene cada píxel de la imagen, es la representación de su luminancia, en una escala que se extiende entre blanco y negro. El tono más oscuro posible es el color negro, representando la ausencia total de luz transmitida o reflejada y el tono más claro posible es el blanco, la reflexión total de la luz en todas las magnitudes de onda visibles[2].

Debido a que las imágenes RGB son un conjunto de bandas de las cuales cada una de ellas es una imagen en escala de grises, podemos realizar una conversión de RGB a escala de grises utilizando la siguiente fórmula:

$0,299 \cdot Rojo + 0{,}587 \cdot Verde + 0,114 \cdot Azul$ [26]

Esta fórmula representa la percepción relativa de la luminosidad promedio del rojo, verde y azul.

A continuación, mostraremos como se ha implementado estos algoritmos en cada librería.

Escala de grises en Repa

La primera versión para extraer la luminosidad de una imagen RGB se desarrolló paralelizando el cálculo entre 2 de las 3 bandas, ayudándonos de la función zipwith para realizar la suma de la luminosidad de esas componentes. Posteriormente, realizamos el mismo cálculo con la banda resultante de la suma de los dos canales anteriores y la banda que queda.

Al final descartamos esta versión, debido a que conseguimos realizarla de manera mucho más eficiente utilizando la función zip3, que fusiona las tres bandas produciendo que cada elemento de una matriz sea una terna (este método de fusión se realiza en un tiempo de complejidad 0(1)), pudiendo realizar una función map paralela calculando la luminosidad de cada pixel, siendo más eficiente que la versión anterior.

Escala de grises en Accelerate

Para la realización de este algoritmo fue paralelizando la función map donde cada elemento de la matriz se calculaba la luminosidad. No hubo que usar ningún zip para fusionar los arrays porque las imágenes RGB de Accelerate, por defecto lo hemos

construido con una matriz bidimensional donde cada elemento del arreglo es una terna con los componentes RGB de esa posición.

3.4. Filtro Gaussiano

El filtro gaussiano es un operador de convolución que se emplea en la eliminación de ruido en imágenes y vídeos. Desde una perspectiva matemática, el proceso de desenfoque gaussiano en una imagen es la convolución de una imagen y su distribución normal[7]. Esta técnica se denomina desenfoque gaussiano porque la distribución normal también se le conoce como distribución gaussiana.

Podemos definir el tamaño del núcleo de acuerdo con los requisitos que usted prefiera, pero la desviación estándar de la distribución gaussiana en la dirección X e Y debe elegirse con cuidado teniendo en cuenta el tamaño del kernel, de manera que los bordes del kernel estén cerca de cero. A continuación se va a mostrar el kernel 3x3 y 5x5 para un valor $\sigma = 1$.

$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad \frac{1}{273} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figura 3.1: Kernel gaussian de tamaño 3x3 y 5x5 con $\sigma = 1$

Para la realización de ambos algoritmos en primer lugar, se decidió utilizar una matriz gaussiana con desvíación estándar de $\sigma = 1$ y dimensionalidad 5x5 como stencils, que posteriormente se utilizó para realizar primero, la convolución con la imagen de forma paralela, y posteriormente, lo dividimos por 273 de forma paralela todos los píxeles de la matriz para conseguir el desenfoque que queremos.

Investigando más adelante, intentamos desarrollar una versión donde aprovechemos la propiedad separable que tiene el filtro gaussiano de dividir el proceso en dos pasos, donde en la primera pasada se usa uno de los kernels para desenfocar la imagen en una dirección (horizontal o vertical) y posteriormente en la segunda pasada se aplicará el stencil en la dirección restante.

El efecto resultante de realizar la convolución de una pasada con una matriz bidimensional o 2 pasadas con vectores unidimensionales es el mismo, con la ventaja que requerimos menos cálculos al tener kernels más pequeños, consiguiendo así mucha mejor eficiencia.

Por lo que para la realización de esta segunda versión, hemos utilizado dos arreglos gaussianos con la misma desviación estándar $\sigma = 1$ con dimensionalidad 1x5 y 5x1 respectivamente como stencils, que posteriormente aplicaremos a las bandas de nuestra

imagen de forma paralela para poder conseguir el mismo desenfoque, pero con menos cálculos.

3.5. Filtro media

El filtro de la media es el más simple, intuitivo y fácil de ejecutar para suavizar imágenes, es decir, reducir la cantidad de variaciones de intensidad entre píxeles vecinos[10].

La idea de este filtro es visitar cada píxel de la imagen y remplazarla por la media de sus píxeles vecinos, incluido él mismo, esto tiene el efecto de eliminar los valores de píxeles que no son representativos en su entorno.

Al igual que otros filtros, se basa en una convolución donde se utiliza un kernel, que representa la forma y el tamaño del vecindario que se utilizará para calcular la media. Con frecuencia se utiliza un kernel de tamaño 3x3 como el de la figura 3.2, aunque se pueden utilizar núcleos más grandes, como por ejemplo uno de 5x5 para suavizados más severos.

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figura 3.2: *Kernel* promedio de tamaño 3x3 para realizar el filtro media

En ambos módulos, la implementación del filtro media a una imagen se efectúo aplicando la función stencil utilizando el kernel mostrado en la figura 3.2.

Como en la librería Repa no acepta que los stencils sean números decimales, en primer lugar se hizo la utilización del stencil usando el kernel de unos, y posteriormente se utilizó una función map en paralelo donde a cada elemento de la matriz se le dividió por 9.

3.6. Sobel

El operador Sobel es utilizado obtener la magnitud del gradiente correspondiente a una imagen y así, enfatizamos las regiones de alta frecuencia espacial que corresponden a los bordes[11].

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figura 3.3: *Kernels* Gradiente eje X y Gradiente eje Y

En teoría, el operador Sobel, consta de un par de núcleos de convolución de tamaño 3x3 como se muestra en la figura 3.3 Cada uno es correspondiente al gradiente vertical y horizontal de la imagen.

Siendo Gx el gradiente horizontal y Gy el eje vertical, para sacar la magnitud del gradiente de la imagen hay que realizar la siguiente fórmula:

$|G| = \sqrt{G_x^2 + G_y^2}$

Para la realización del algoritmo en ambos módulos, en primer lugar se calculó los gradientes de la imagen, para posteriormente calcular de forma paralela la magnitud de la imagen.

Los gradientes fueron calculados utilizando *kernels* de dimensión 3x3 para sacar las convoluciones de los gradientes.

3.7. Filtro Laplaciano

En el procesamiento de imágenes digitales, el filtro Laplaciano es un detector de bordes que se emplea para calcular las segundas derivadas de una imagen, midiendo la velocidad a la que cambian las primeras derivadas[30].

Esto determina si un cambio en los valores de píxeles adyacentes se deben a un borde o a una progresión contínua.

El laplaciano L(x,y) de una imagen con valores de intensidad I(x,y) viene dado por:

$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$ [9]

No obstante, para imágenes de dos dimensiones, el operador discreto de laplace se puede calcular como si fuera un filtro de convolución. Los 2 *kernels* bidimensionales de uso común se encuentran en la figura 3.4.

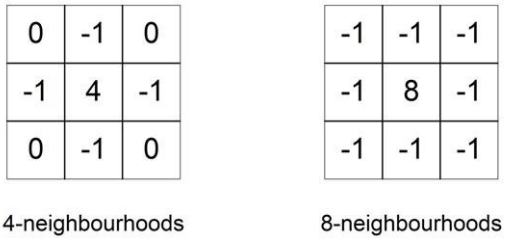

Figura 3.4: Dos aproximaciones al uso del filtro laplaciano

La implementación en ambos módulos se realizó utilizando la primera imagen de los kernels de laplace 3.4 y realizando la convolución como se ha estado realizando anteriormente.

3.8. Desarrollo de un módulo para realizar benchmarks

Comparar el tiempo que tarda en realizarse un algoritmo es un indicador del rendimiento en el mundo real, por lo que se han investigado diferentes librerías para crear *benchmarks* en Haskell.

Entre las diversas librerías que existen, la evaluación del rendimiento de código Haskell ha sido dominada por el uso del módulo **Criterion**[25], en consecuencia, se ha utilizado este paquete como base para realizar nuestro propio módulo de pruebas de rendimiento de nuestros cálculos.

Para desarrollar benchmarks mediante el uso de criterion, deberemos utilizar sus funciones predefinidas para ello. Un ejemplo sería este:

```
-- benchmark.hs
import Criterion.Main

-- Funcion que vamos a sacar su rendimiento
fib m | m < 0     = error "negative!"
      | otherwise = go m
  where go 0 = 0
        go 1 = 1
        go n = go (n-1) + go (n-2)

-- funcion main que nos realiza los benchmark
main = defaultMain [
  bgroup "fib" [ bench "1"  $ whnf fib 1
               , bench "5"  $ whnf fib 5
               , bench "11" $ whnf fib 11
               ]
  ]
```

Extracto de código 3.1: Ejemplo de un benchmark usando Criterion

Como construir un conjunto de test

Un conjunto de test en el módulo Criterion consiste en una colección de valores del tipo Benchmark.

```
main = defaultMain [
  bgroup "fib" [ bench "1"  $ whnf fib 1
               , bench "5"  $ whnf fib 5
               , bench "11" $ whnf fib 11
               ]
  ]
```

Donde, la función defaultMain, toma una lista del tipo Benchmark y donde evalua el rendimiento a cada una de las funciones que le pasamos.

Para poder agrupar pruebas de rendimiento por categorías, podemos utilizar la función **bgroup**. El primer argumento es el nombre que va a tener el grupo de pruebas de rendimiento y el segundo parámetro es la lista de las pruebas que se van a realizar para ese grupo.

```
bgroup :: String -> [Benchmark] -> Benchmark
```

Ahora que hemos visto como agrupar los benchmarks habrá que ver como construir una medida rendimiento para una función, y eso se realiza gracias a la función **bench**.

Esta función toma como argumentos el nombre de la actividad que estamos realizando y un tipo de datos llamado *Benchmarkable* que es un contenedor de código al que se le puede realizar pruebas de rendimiento.

Por defecto, criterion nos permite realizar pruebas de rendimiento de código puro y cualquier acción de la mónada IO.

La mayoría de las acciones de la mónada IO pueden efectuarse sus pruebas de rendimiento utilizando una de las siguientes funciones:

```
nfIO   :: NFData a => IO a -> Benchmarkable
whnfIO ::                IO a -> Benchmarkable
```

Aunque para las pruebas de rendimiento que hemos realizado, hemos usado siempre la función nfIO, debido a que así nos aseguramos que se evalúe por completo la función.

En cambio, si queremos realizar pruebas de rendimiento con funciones puras, deberemos utilizar las siguientes funciones para evitar la evaluación perezosa:

```
nf :: NFData b => (a -> b) -> a -> Benchmarkable
whnf :: (a -> b) -> a -> Benchmarkable
```

La función nf acepta dos parámetros, una función casi saturada que queremos medir el rendimiento y el segundo es el argumento final para darle.

La función whnf evalúa el resultado de una acción lo suficientemente profundo como para que se conozca el constructor más externo, esto se le conoce como forma normal de cabeza débil.

Creación de un módulo Backend

Debido a que el módulo Accelerate nos permite ejecutar el mismo algoritmo entre sus diferentes *backends* (intérprete, paralelizando CPU o CPU) sin tener que cambiar el código, por lo cual, para poder ejecutar en un mismo conjunto de pruebas algoritmos con los distintos tipos de *backends*, se decidió realizar un pequeño módulo llamado **Backend**, inspirado en como se realizan las pruebas de rendimiento en el módulo *accelerate-examples* pero sin tanta complejidad.

Problemas con Repa

Cuando estuvimos desarrollando el módulo, cuando teníamos que realizar las pruebas de rendimiento de algoritmos relacionados con la librería Repa, existía un error diciéndonos que no existía ninguna instancia de NFData de la librería deepseq, por lo que investigando cuando creamos nuestros propios tipos, tenemos que crear instancias de NFData para que se asegure que toda la estructura de la matriz Repa se evalúe por completo

Por lo que como consejo para personas que usen estructuras de datos propias o fuera del base, asegurarse de que vengan con instancias creadas de NFData o si no, crearlas ustedes mismos.

Finalmente, con todos estos conocimientos expuestos se pudo desarrollar de manera eficiente el módulo con las funciones necesarias para crear el conjunto de *benchmarks*.

4. Análisis de rendimiento

4.1. Plataforma utilizada

Todos los resultados que se van a mostrar en este capítulo pertenecen a ejecuciones efectuadas sobre el mismo sistema, compuesto por una CPU Intel Core i7-7700HQ de 4 núcleos y a una frecuencia básica de 2,8 GHz. La GPU del sistema es una NVIDIA GeForce GTX 1050 con 2GB de memoria y 640 núcleos CUDA.

Se intentó recurrir al uso de los miniclusters de GPU proporcionados por el grupo de investigación en Computación Natural de la Universidad de Sevilla para conseguir unos resultados mucho más grandes, pero debido a los diferentes conflictos que se desarrollaron al montar el proyecto en esos clusters, se tuvo que descartar la idea por falta de tiempo. En consecuencia, los diferentes test realizados en esta sección, llegaron a lo que pudiera aguantar el sistema del autor.

A la hora de realizar el análisis de rendimiento de los distintos algoritmos, solamente hemos tenido en cuenta el tiempo que tarda el algoritmo en ejecutarse, eliminando el tiempo de transformación de la imagen a esa librería o la conversión a imagen después de haber aplicado la función. Además, para probar la eficiencia de cada algoritmo, se ha probado con imágenes con una cantidad de píxeles distinta.

La media de los tiempos de ejecución se ha medido gracias al desarrollo de un módulo para realizar Benchmarks explicado en el capítulo anterior.

Después de haber obtenido el tiempo medio que tarda cada algoritmo con distintas imágenes, se decidió desarrollar unas tablas, para posteriormente ser leídas con la librería Pandas del lenguaje Python y crear gráficos gracias a seaborn.

A continuación vamos a ver los resultados de los diferentes algoritmos implementados.

4.2. Rendimiento : Generación de Histogramas

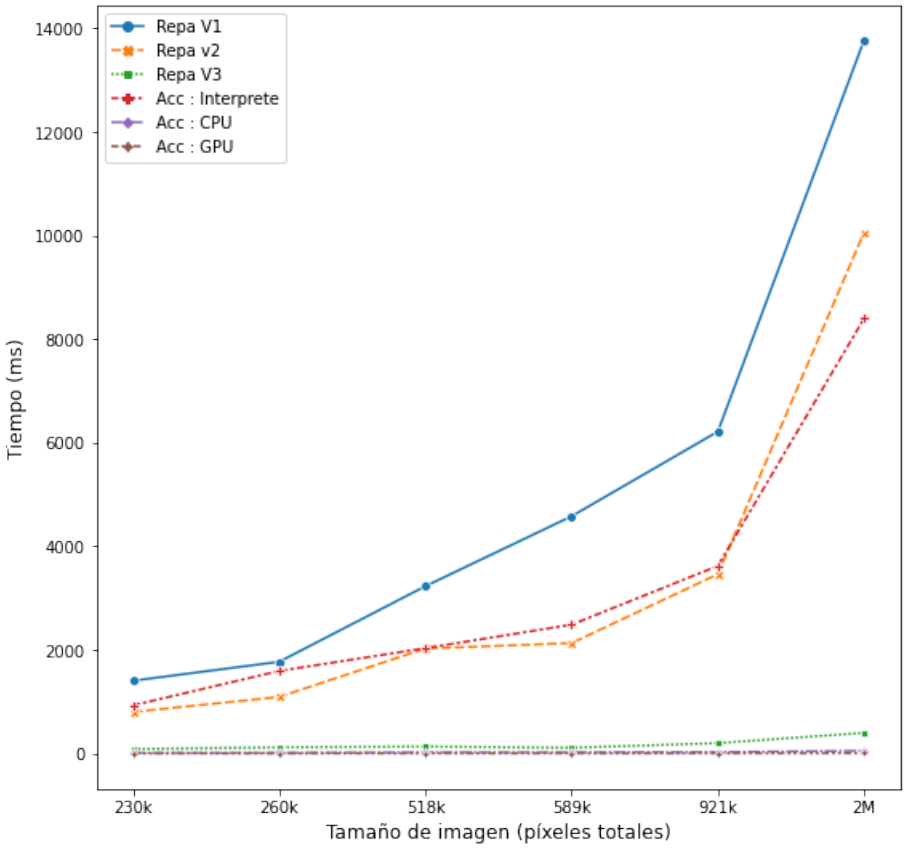

Figura 4.1: Cambiar caption

4.3. Escala de grises

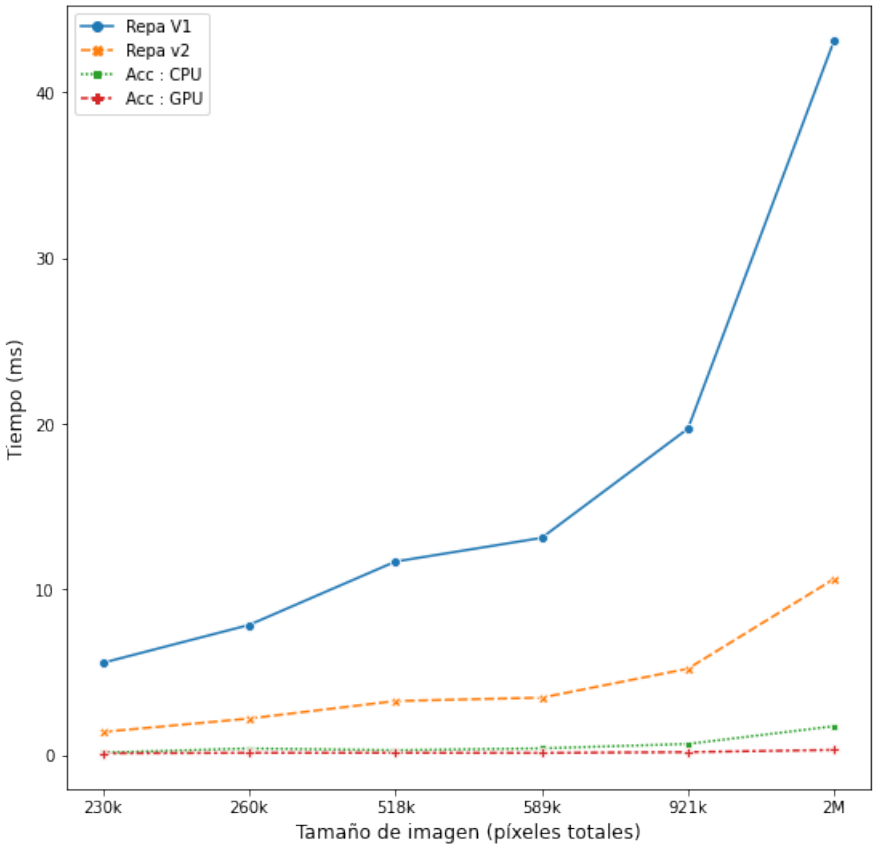

Figura 4.2: Cambiar caption

4.4. Filtro Gaussiano

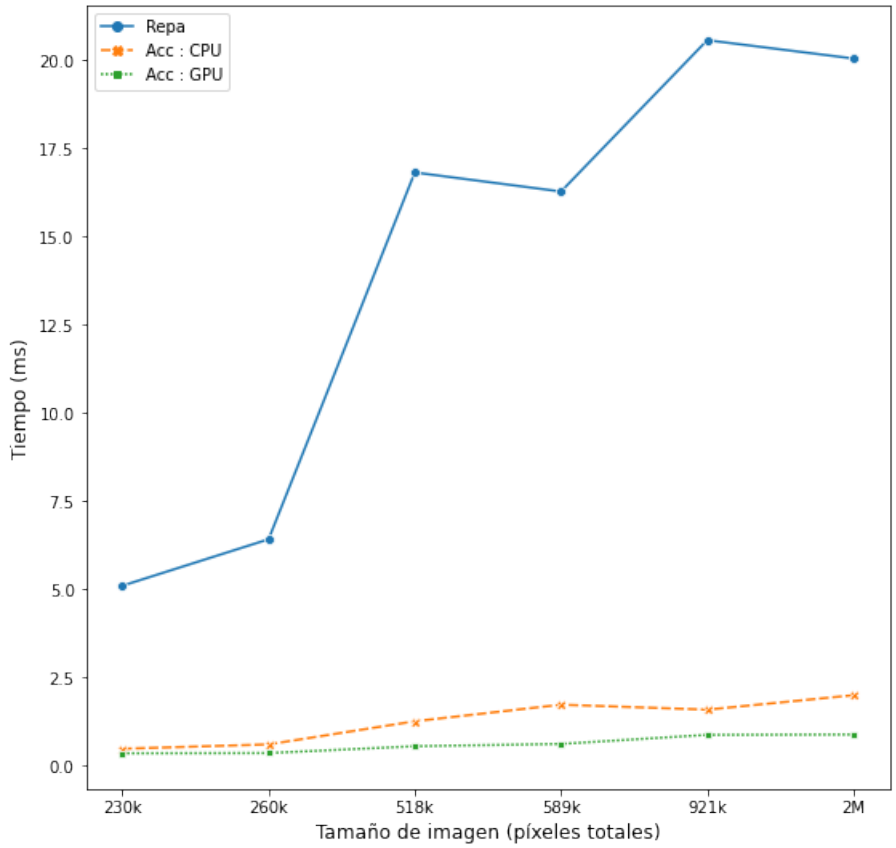

Figura 4.3: Cambiar caption

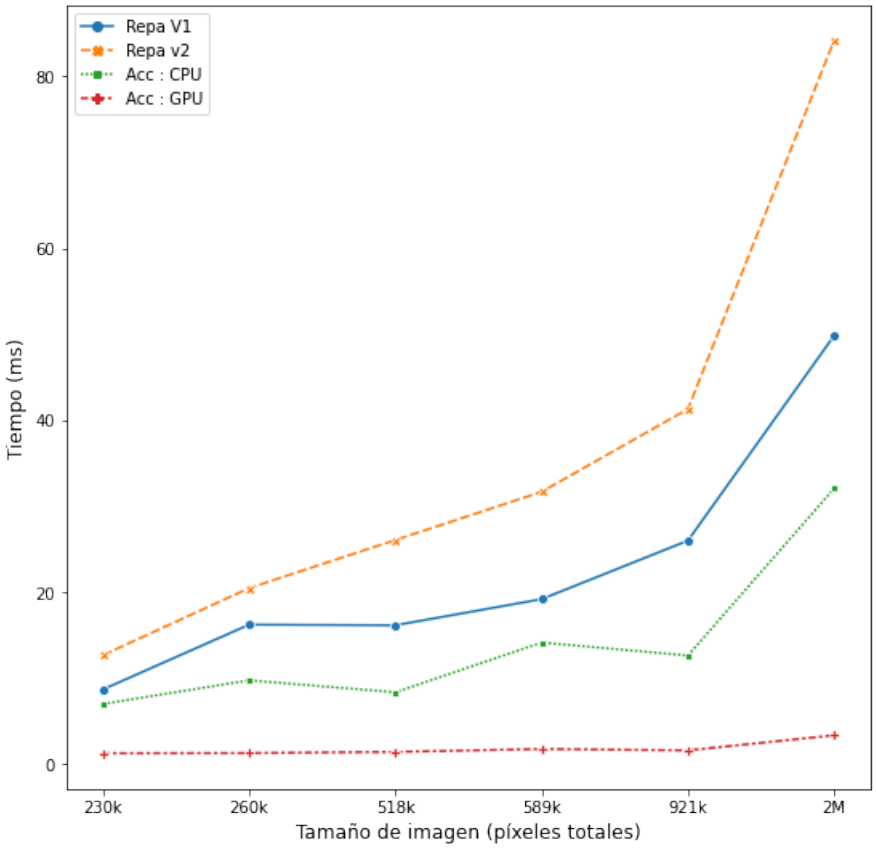

Figura 4.4: Cambiar caption

4.5. Filtro media

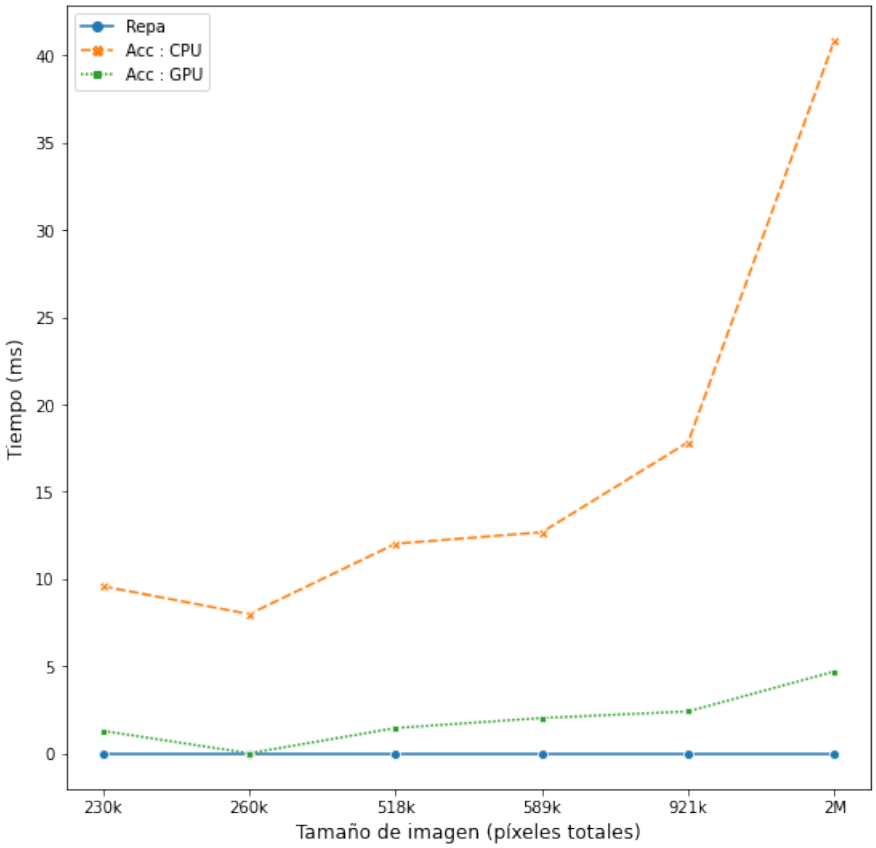

Figura 4.5: Cambiar caption

4.6. Sobel

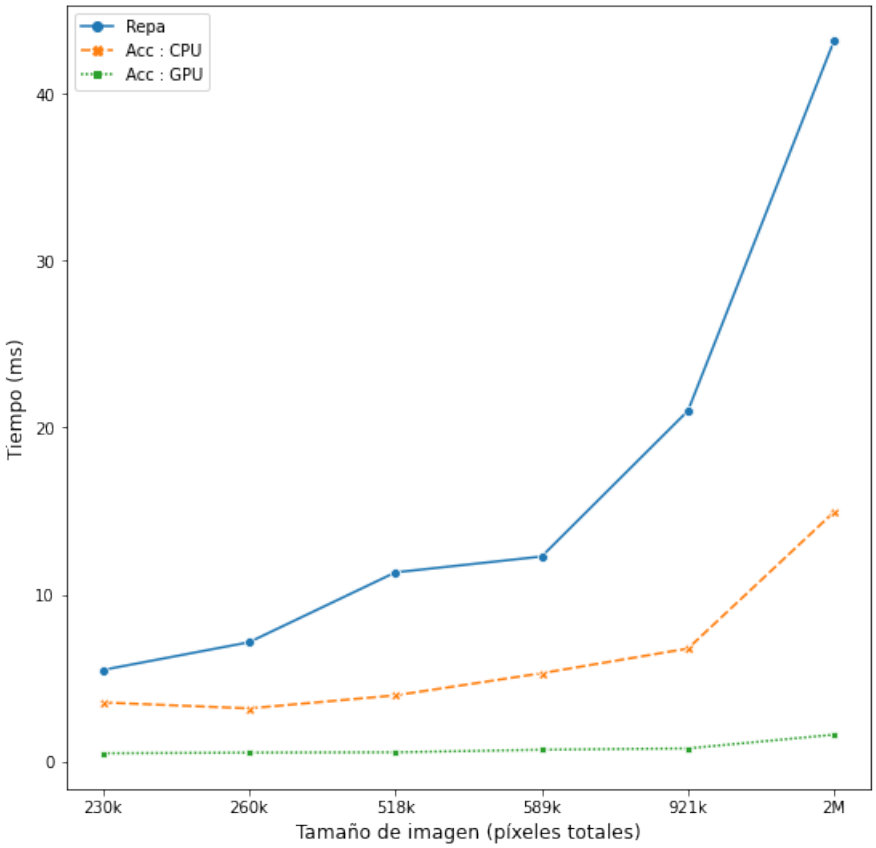

Figura 4.6: Cambiar caption

4.7. Filtro Laplaciano

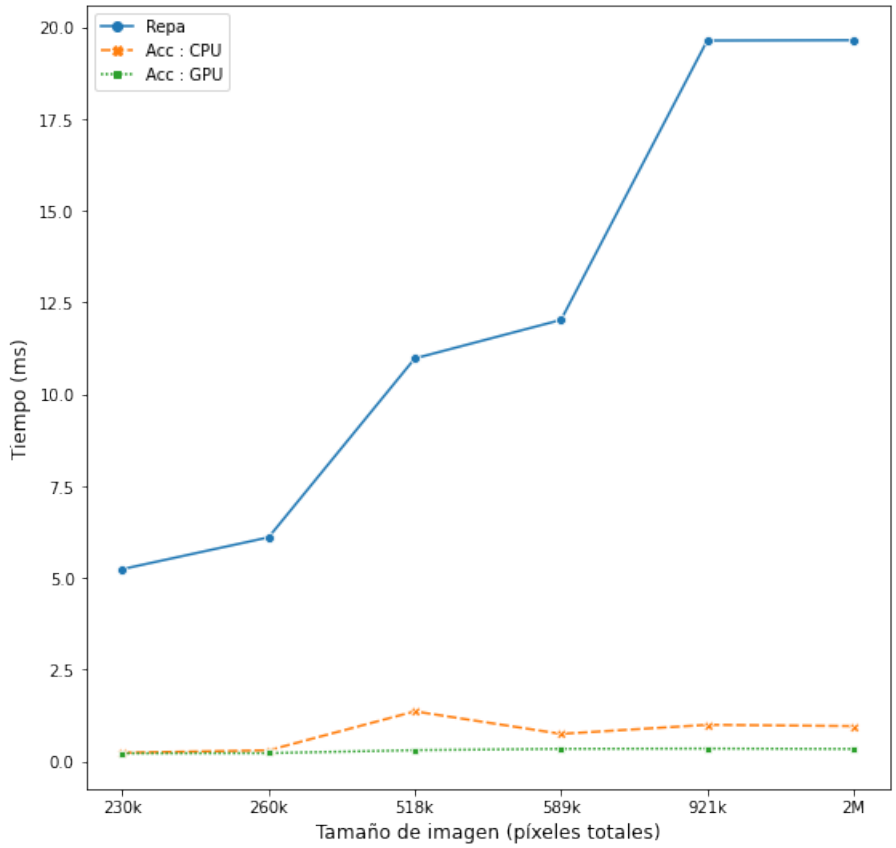

Figura 4.7: Cambiar caption

5. Conclusiones y trabajo futuro

5.1. Conclusiones

5.2. Trabajo futuro

6. Apéndices

6.1. Manual de Usuario

A continuación, se procede a describir los sistemas operativos recomendables, así como las dependencias necesarias para poder ejecutar las pruebas desarrolladas por el autor.

6.1.1. Sistemas Operativos Recomendables

El proyecto ha sido desarrollado utilizando el lenguaje de programación Haskell, en su estándar más reciente, Haskell 2010. Y aunque no se ha haya realizado ningún código en específico del sistema operativo, el desarrollo de todo el proyecto ha sido realizado sobre el sistema operativo POP!_OS 21.10, derivada de Ubuntu.

Por consiguiente, todos las dependencias necesarias para su instalación se podrán usar para las distribuciones derivadas de Ubuntu/Debian. Para el uso de las dependencias en otros sistemas operativos, buscar referencias en la web.

Aún así, se recomienda el uso de sistemas operativos GNU/Linux o OSX debido a que son los únicos sistemas operativos de los que existe documentación en Accelerate.

Dado a que este proyecto hace uso de CUDA, es necesario que la máquina sobre la que se ejecute el proyecto tenga una tarjeta gráfica NVIDIA que sea compatible con esta tecnología [3].

6.1.2. Dependencias

Además de tener instalado el lenguaje de programación Haskell, deberemos de instalar **Stack** [12], una herramienta empleada en la creación de proyectos Haskell y de administración de las dependencias dentro del proyecto. Si no se tuviera en el sistema, ejecutar los siguientes comandos:

```
sudo apt-get install haskell-platform
```

Extracto de código 6.1: Instalación de Haskell

```
wget -qO- https://get.haskellstack.org/ | sh
```

Extracto de código 6.2: Instalación de Stack

LLVM

Para poder ejecutar programas en paralelo sobre CPU/GPU en Accelerate es necesario instalar LLVM, un compilador de optimización maduro que está enfocado a varias arquitecturas. Por lo que se requerirán las siguientes librerías externas:

- LLVM
- libFFI

Ejemplo de instalación de estas librerías en Debian/Ubuntu:

```
apt-get install llvm-9-dev freeglut3-dev libfftw3-dev
```

Extracto de código 6.3: Instalación de LLVM

CUDA

Para el correcto funcionamiento de este proyecto, también es necesario, además de tener una tarjeta compatible con CUDA, instalar las herramientas CUDA en nuestro sistema. Puede encontrarlo en la página oficial de NVIDIA.[5]

6.1.3. Ejecución del programa

El código del proyecto puede ser extraído a través del repositorio alojado en `https://github.com/kennyfh/TFG-kenflohua.git`.

Una vez instaladas las dependencias anteriores y el proyecto descargado, los análisis realizados pueden ser ejecutados haciendo uso del siguiente comando desde la carpeta raíz del proyecto:

```
stack run
```

7. Bibliografía

[1] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, January 2011.

[2] TechTarget Contributor. What is grayscale?, 2010. URL https://www.techtarget.com/whatis/definition/grayscale.

[3] NVIDIA Developer. CUDA GPUs - Compute Capability, June 2012. URL https://developer.nvidia.com/cuda-gpus.

[4] NVIDIA Developer. Cuda zone - library of resources, July 2017. URL https://developer.nvidia.com/cuda-zone.

[5] NVIDIA Developer. CUDA Toolkit 11.7 Downloads, April 2021. URL https://developer.nvidia.com/cuda-downloads.

[6] Bertram Felgenhauer, David Feuer, Ross Paterson, and Milan Straka. Data.sequence, 2014. URL https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-Sequence.html.

[7] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Spatial filters - gaussian smoothing, 2003. URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm.

[8] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Image analysis - intensity histogram, 2003. URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/histgram.htm.

[9] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Spatial filters - laplacian / laplacian of gaussian, 2003. URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm.

[10] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Spatial filters - mean filter, 2003. URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm.

[11] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Feature detectors - sobel edge detector, 2003. URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm.

[12] Commercial Haskell Group. The haskell tool stack, 2015. URL https://docs.haskellstack.org/en/stable/README/.

[13] Khronos Group. The OpenCL Specification, July 2019. URL https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf. publisher: Khronos Group.

[14] HaskellWiki. Language and library specification, 2020. URL https://wiki.haskell.org/Language_and_library_specification.

[15] RALF HINZE and ROSS PATERSON. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006. doi: 10.1017/S0956796805005769.

[16] Intel. CPU vs GPU: ¿cuál es la diferencia?, 2022. URL https://www.intel.com/content/www/es/es/products/docs/processors/cpu-vs-gpu.html.

[17] Kamran Karimi, Neil Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *Computing Research Repository - CORR*, arXiv:1005.2581, 05 2010.

[18] Roman Leshchinskiy. Data.Vector, 2010. URL https://hackage.haskell.org/package/vector-0.12.3.1/docs/Data-Vector.html.

[19] Miran Lipovača. Introduction - so what is haskell, 2011. URL http://learnyouahaskell.com/introduction#so-whats-haskell.

[20] Rick Merritt. What Is Accelerated Computing?, September 2021. URL https://blogs.nvidia.com/blog/2021/09/01/what-is-accelerated-computing/.

[21] Camilo Mosquera and Santiago Peña. Programación paralela, 2020. URL http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#four.

[22] NVIDIA. ¿GPU vs. CPU? ¿Qué es la computación por GPU? | NVIDIA, 2022. URL https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/.

[23] NVIDIA. Tus aplicaciones creativas favoritas aceleradas con GPU NVIDIA, 2022. URL https://www.nvidia.com/es-es/studio/software/.

[24] Sergio Orts Escolano and Vicente Morell Giménez. Introducción a la computación paralela con GPUs, May 2011. URL https://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf.

[25] Bryan O'Sullivan. criterion: a haskell microbenchmarking library, 2014. URL http://www.serpentine.com/criterion/.

[26] Charles Poynton. Color FAQ - Frequently Asked Questions Color, 2006. URL http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToC11.

[27] Elizabeth Riegel. Khronos launches heterogeneous computing initiative, June 2008. URL https://web.archive.org/web/20080620123431/http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/.

[28] Fernando Sancho Caparrini. Haskell: el lenguaje funcional, 2016. URL http://www.cs.us.es/~fsancho/?e=110.

[29] Jesús Adrian Toro Sanchez. Paralelismo, tipos, July 2016. URL https://arqcompingener.wordpress.com/2016/07/21/paralelismo-tipos/.

[30] Instituto de Radioastronomía y Astrofísica UNAM. Apply laplacian filters, 2012. URL https://www.irya.unam.mx/computo/sites/manuales/IDL/Content/GuideMe/ImageProcessing/LaplacianFilters.html.