
Programación Funcional Paralela en GPUs

Kenny Jesús Flores Huamán

Un Trabajo de Fin de Grado para la obtención del título
Ingeniería Informática - Tecnologías Informáticas



Dirigido por
Miguel Ángel Martínez del Amor

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

Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

Junio 2022

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todos aquellos que me han asistido a lo largo de esta etapa y han colaborado en esta investigación.

En primer lugar, a mi tutor Miguel Ángel Martínez del Amor, por todo el apoyo que me ha brindado tanto a nivel académico como en lo personal.

En segundo lugar, a mis padres, Victor y María Isabel, mi tía Roxana y Ana Ramos por haber confiado en mí en todo momento y dejarme compaginar los estudios con el deporte de alto nivel. Además de mi primo Lucho, por haberme brindado apoyo durante el desarrollo del TFG cuando lo necesitaba.

Me gustaría agradecer también, a mis hermanos del kung-fu, Iván, Mei, Luz, Candela, Max, Ana, Julia y muchos más, por todos esos momentos que hemos vivido entrenando y compitiendo juntos.

Finalmente, pero no menos importante, agradecer a mis amigos de la infancia, Edward y Cristian, además de mis compañeros de informática y del doble grado, Nuru, Gardo, Lourdes, Blackzaren, Chumbi y muchos más, por tantos buenos momentos, reuniones y charlas muy profundas durante una de las etapas más duras de mi vida.

El desarrollo de esta investigación ha sido muy influyente en mi persona y por eso me gustaría agradecer a todas aquellas personas que me han apoyado durante este proceso.

A todos ellos, mil gracias.

Resumen

El uso de la programación paralela sobre GPUs está en todos lados, desde la edición de vídeos, hasta creación de modelos IA para detectar fraudes en tiempo real, pasando por la generación de los gráficos en videojuegos.

En este trabajo, se realizará el estudio, análisis y experimentación del rendimiento, flexibilidad y diseño de algoritmos paralelos sobre CPU y GPU desde el paradigma funcional, en vista de que este paradigma cuenta con una serie de propiedades muy buenas para compaginarse con el paralelismo.

Para poder explorar el empleo de la programación funcional paralela se han diseñado algoritmos paralelos para el procesamiento de imágenes digitales en Repa y Accelerate, dos librerías escritas en Haskell.

Al final del trabajo utilizaremos algunas imágenes y vídeos para probar la eficiencia de estos algoritmos y poder sacar unas conclusiones.

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

Abstract

The use of parallel programming over GPU's is everywhere, from video editing to the creation of IA models to detect frauds in actual time, including graphic generation for video games.

This project will study, analyse and experiment on the performance, flexibility and design of parallel algorithms over CPU and GPU from the functioning paradigm, since this paradigm offers certain properties that are good for adjusting with parallelism.

In order to explore the use of parallel functioning programmation, parallel algorithms have been designed for processing digital images with REPA and Accelerate, two written libraries in Haskell.

In the end of this project, some images and videos will be used to prove the efficiency of these algorithms in order to draw the appropriate conclusions.

Keywords: Parallel computing, CUDA, Image processing, Haskell, Repa, Accelerate

Índice general

1. Introducción	1
1.1. Contexto teórico	1
1.2. Objetivos	2
1.2.1. Objetivos técnicos	2
1.2.2. Objetivos académicos	3
1.3. Estructura de la memoria	3
2. Estado del Arte	4
2.1. Paralelismo y Computación Paralela	4
2.1.1. GPU y CPU	4
2.1.2. Paralelismo y Tipos de Paralelismo	5
2.1.3. Programación Paralela	5
2.1.4. Patrones de diseño paralelo	6
2.1.5. Programación GPU mediante CUDA	12
2.1.6. Programación funcional paralela	13
2.2. Haskell	14
2.2.1. Características principales de Haskell	14
2.3. Programación paralela de datos con REPA	16
2.3.1. Matrices, dimensionalidad e índices	16
2.3.2. Generando arrays	19
2.3.3. Convoluciones mediante Repa Stencil	20
2.3.4. Ejemplo: Producto escalar	22
2.4. Aceleración GPU con Accelerate	24
2.4.1. Características principales de Accelerate:	24
2.4.2. Arrays, dimensionalidad e índices	25
2.4.3. Ejecución de un cálculo acelerado simple	27
2.4.4. Generando arrays	28
2.4.5. Lifting y Unlifting	29
2.4.6. Convoluciones mediante Accelerate Stencil	29
2.4.7. Ejemplo: Producto escalar	31
3. Desarrollo del Proyecto	33
3.1. Lectura de imágenes	33
3.1.1. Lectura de imágenes en Repa	33
3.1.2. Lectura de imágenes en Accelerate	34
3.1.3. Funciones de conversión de tipos de datos	34
3.2. Lectura y escritura de vídeos	34
3.3. Generación de Histogramas	35
3.4. Escala de grises	39
3.5. Filtro Gaussiano	40
3.5.1. Filtro Gaussiano empleando 2 kernels	41
3.5.2. Filtro Gaussiano empleando 1 kernel	43
3.6. Filtro media	44

3.7.	Sobel	45
3.8.	Filtro Laplaciano	48
3.9.	Desarrollo de un módulo para realizar benchmarks	49
4.	Análisis de rendimiento	52
4.1.	Detalles de experimentación	52
4.2.	Validación de los resultados	53
4.2.1.	Generación de Histogramas	53
4.2.2.	Escala de grises	55
4.2.3.	Filtro Gaussiano	56
4.2.4.	Filtro media	57
4.2.5.	Sobel	58
4.2.6.	Filtro Laplaciano	59
4.2.7.	Vídeos de prueba	60
4.3.	Pruebas de rendimiento	60
4.3.1.	Generación de Histogramas	61
4.3.2.	Escala de grises	62
4.3.3.	Filtro Gaussiano	63
4.3.4.	Filtro media	64
4.3.5.	Sobel	65
4.3.6.	Filtro Laplaciano	66
4.3.7.	Tabla resumen	67
4.3.8.	Rendimiento en vídeos	68
5.	Conclusiones y trabajo futuro	69
5.1.	Conclusiones	69
5.2.	Trabajo futuro	70
5.2.1.	Optimización de programas Repa	70
5.2.2.	Uso de otros lenguajes de programación	72
5.2.3.	Pruebas de rendimiento en distinto hardware	72
5.3.	Consejos y buenas prácticas	72
5.3.1.	Accelerate	73
5.3.2.	Repa	73
5.3.3.	JuicyPixels	74
5.3.4.	FFmpeg-light	74
5.3.5.	Criterion	75
A.	Apéndices	77
A.1.	Manual de Usuario	77
A.1.1.	Sistemas Operativos Recomendables	77
A.1.2.	Dependencias	77
A.1.3.	Ejecución del programa	78
B.	Bibliografía	79

Índice de figuras

2.1. Ejecución en serie y en paralelo del patrón <code>map</code>	6
2.2. Ejecución en serie del patrón <code>Reduce</code>	7
2.3. Ejecución en paralelo del patrón <code>Reduce</code> haciendo uso de árboles	8
2.4. Ejecución del patrón <code>mapReduce</code> . Cuando después de un <code>map</code> ocurre una reducción, la combinación puede implementarse de una forma más eficiente dándole el resultado de aplicar el mapa como datos de entrada al patrón <code>reduce</code>	9
2.5. Ejecución en serie del patrón <code>scan</code>	10
2.6. Ejecución en paralelo del patrón <code>scan</code>	10
2.7. Ejecución del patrón stencil	12
3.1. Kernel gaussiano de tamaño 3x3 y 5x5 con $\sigma = 1$	40
3.2. Kernel promedio de tamaño 3x3 para realizar el filtro media	44
3.3. Kernels Gradiente eje X (vertical) y Gradiente eje Y (horizontal)	46
3.4. Dos aproximaciones al uso del filtro laplaciano	48
4.1. Imagen de entrada y su histograma RGB (Accelerate). Este histograma es idéntico que el conseguido con una función histograma escrita en Haskell	53
4.2. Imagen de entrada y su histograma RGB (Repa V3).Este histograma es idéntico que el conseguido con una función histograma escrita en Haskell	54
4.3. Imagen de entrada y resultado transformación RGB a escala de grises (Accelerate). Esta transformación es idéntica que el conseguido con una función grayScale escrita en Haskell	55
4.4. Imagen de entrada y resultado transformación RGB a escala de grises (Repa V2). Esta transformación es idéntica que el conseguido con una función grayScale escrita en Haskell	55
4.5. Imagen de entrada con ruido gaussiano $\sigma = 2$ y resultado tras aplicarle filtro gaussiano (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro gaussiano escrita en Haskell	56
4.6. Imagen de entrada con ruido gaussiano $\sigma = 2$ y resultado tras aplicarle filtro gaussiano (Repa). Esta transformación es idéntica que el conseguido con una función de filtro gaussiano escrita en Haskell	56
4.7. Imagen de entrada con ruido de disparo y resultado tras aplicarle el filtro media (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro media escrita en Haskell	57
4.8. Imagen de entrada con ruido de disparo y resultado tras aplicarle el filtro media (Repa). Esta transformación es idéntica que el conseguido con una función de filtro media escrita en Haskell	57
4.9. Imagen de entrada y resultado tras aplicarle el filtro Sobel (Accelerate). Esta transformación es idéntica que el conseguido con una función Sobel escrita en Haskell	58

4.10. Imagen de entrada y resultado tras aplicarle el filtro Sobel (Accelerate). Esta transformación es idéntica que el conseguido con una función Sobel escrita en Haskell	58
4.11. Imagen de entrada y resultado tras aplicarle el filtro Laplace (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro Laplaciano escrita en Haskell	59
4.12. Imagen de entrada y resultado tras aplicarle el filtro Laplace (Repa). Esta transformación es idéntica que el conseguido con una función de filtro Laplaciano escrita en Haskell	59
4.13. Tiempos de ejecución para las diferentes versiones de generar histogramas. Tenga en cuenta la escala logarítmica en el eje de coordenadas.	61
4.14. Tiempos de ejecución para las diferentes versiones de convertir una imagen RGB a escala de grises.Tenga en cuenta la escala logarítmica en el eje de coordenadas.	62
4.15. Tiempos de ejecución para las diferentes versiones de realizar el filtro media. Tenga en cuenta la escala logarítmica en el eje de coordenadas. .	64
4.16. Tiempos de ejecución para las diferentes versiones de realizar el algoritmo Sobel. Tenga en cuenta la escala logarítmica.	65
4.17. Tiempos de ejecución para las diferentes versiones de realizar el algoritmo Laplace. Tenga en cuenta la escala logarítmica en el eje de coordenadas.	66

Índice de cuadros

4.1.	Tabla de rendimiento en ms - Filtro Gauss (1 kernel 5x5)	63
4.2.	Tabla de rendimiento en ms - Filtro Gauss (2 Kernels 1x5 y 5x1)	63
4.3.	Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Repa y Accelerate Native	67
4.4.	Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Repa y Accelerate PTX	67
4.5.	Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Accelerate Native y PTX	68

Índice de extractos de código

2.1. Ejemplo de uso de la función map	17
2.2. Ejemplo de uso de 2 maps consecutivos	18
2.3. Generación de un Laplace Stencil	21
2.4. Stencils usando QuasiQuotes	21
2.5. Producto escalar de forma secuencial en Haskell	22
2.6. Producto escalar en Repa (paralelizando CPU).	23
2.7. Laplace Stencil en Accelerate	29
2.8. Producto escalar en Accelerate (paralelizando GPU).	31
3.1. Generación de histogramas paralelizando el cálculo de las 3 bandas	37
3.2. Generación de histogramas haciendo el cálculo de cada fila de la imagen en paralelo, además de realizarlo para cada canal en paralelo	38
3.3. Generación de histogramas en Accelerate	38
3.4. Algoritmo escala de grises (Repa V2)	39
3.5. Algoritmo escala de grises (Accelerate)	40
3.6. Algoritmo filtro Gaussiano empleando 2 kernels (Repa)	42
3.7. Algoritmo filtro Gaussiano empleando 2 kernels (Accelerate)	43
3.8. Algoritmo filtro Gaussiano empleando 1 kernel (Repa)	43
3.9. Algoritmo filtro Gaussiano empleando 1 kernel (Accelerate)	44
3.10. Algoritmo Filtro Media (Repa)	45
3.11. Algoritmo Filtro Media (Accelerate)	45
3.12. Algoritmo Sobel (Repa)	47
3.13. Algoritmo Sobel (Accelerate)	47
3.14. Ejemplo de un benchmark usando Criterion	49
A.1. Instalación de Haskell	77
A.2. Instalación de Stack	77
A.3. Instalación de LLVM	78
A.4. Instalación de FFmpeg	78

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[1].

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 cometan 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 desplegar la IA para la detección y prevención de fraudes requieren grandes recursos, muchas corporaciones para conseguir una detección más rápida y una mayor escalabilidad, hacen uso de los servicios escalables basados en la nube y acelerados por GPU que permitan ejecutar bibliotecas, rutinas y algoritmos de IA/ML optimizados[2].

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 horizontal a las relaciones de aspecto adecuadas para las redes sociales, facilitando la producción de contenido en línea[3].

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 a que la programación funcional fue pensado en parte para la combinarlo con la programación paralela, con motivo de 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. Además del enorme interés que le causó dar este estándar en la asignatura de Programación Declarativa del grado de I.I Tecnologías Informáticas en la Universidad de Sevilla.

De manera que, en este trabajo se pretende realizar un estudio del rendimiento de como funciona la programación funcional paralela en GPUs, comparándolo con la programación funcional paralela en CPUs. Para el desarrollo de los algoritmos, se ha decidido elegir Haskell entre todos los lenguajes funcionales en vista de que además

de tener los beneficios de ser un lenguaje funcional (más adelante se verá esto) tiene los suyos propios como promover la seguridad y estabilidad, además de permitirnos construir programas tolerantes a fallas.

Otras ventajas que tiene el lenguaje Haskell es que al utilizar evaluación perezosa (las cosas solo se evalúan si son necesarias) puede producir programas mucho más rápidos. También cuenta con su propio compilador llamado Glasgow Haskell Compiler (GHC)¹ muy parecido al compilador de C gcc, con la diferencia de que el GHC cuenta con un gestor de memoria automático para evitar desbordamientos de memoria.

Aunque Haskell donde realmente es fuerte es en la creación de lenguajes específicos de dominio y en la programación paralela y concurrente [4], cosa que nos viene muy bien para el desarrollo de este proyecto.

1.2. Objetivos

Los lenguajes funcionales son conocidos por tener buenas propiedades para la programación paralela, y existen librerías para el paralelismo sobre CPU y GPU, por lo que 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 módulos orientados a la programación paralela dentro del ecosistema del lenguaje de programación Haskell, mediante la implementación de algoritmos para procesamiento de imágenes.

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.

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

1.2.1. 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 CPU.
- **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.

¹<https://www.haskell.org/ghc/>

- **Objetivo 4:** Poder desempeñar documentos bien estructurados, de una forma clara y entendible utilizando el sistema de composición de textos L^AT_EX
- **Objetivo 5:** Ser capaz de escribir código eficiente y bien documentado utilizando el lenguaje de programación Haskell.

1.2.2. 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 L^AT_EX.

1.3. 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 sobre paralelismo 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 vídeos, además de 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. Además de unos consejos y buenas prácticas que he ido recolectando haciendo uso de esos módulos.

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) [5].

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.

Aunque en el ámbito de la informática de alto rendimiento (**HPC**) el uso de gráficas está entrando con gran fuerza. EL HPC se refiere a la **agregación** de recursos informáticos en **gran escala** para ofrecer un mayor rendimiento, por lo cual, la combinación de GPU con clústeres HPC puede aumentar la potencia de procesamiento en los centros de datos, permitiendo a los investigadores procesar grandes cantidades de datos de forma más rápida, eficiente y económica [6].

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 GPU se compone de numerosos núcleos que pueden manejar miles de subprocessos de manera síncrona [7]. 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)[8].

Tipos de paralelismo:[9]

- **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 de ninguna otra forma. [10].

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. Patrones de diseño paralelo

Los patrones (*patterns* en inglés) recientemente se han vuelto populares como una forma de codificar buenas prácticas en la ingeniería de software. En esta sección, usaremos el término de **patrones de diseño paralelos** para hacer referencia a una combinación recursiva de distribución de los procesos y acceso a los datos para la resolución de un problema en específico. [11]

Map

El patrón `map` consiste aplicar una misma función sobre todos los elementos de una colección de datos. A esta función que se utiliza en el `map` la denominaremos **función elemental**.

Estas funciones elementales utilizadas en el `map` no deben tener efectos secundarios para permitir que todas las instancias puedan ejecutarse en cualquier orden. En consecuencia, esta propiedad nos permite utilizar el patrón `map` de forma paralela sin tener que sincronizar los elementos separados del `map` como podemos observar en la figura 2.1.4, hasta el momento de terminar la ejecución del patrón.

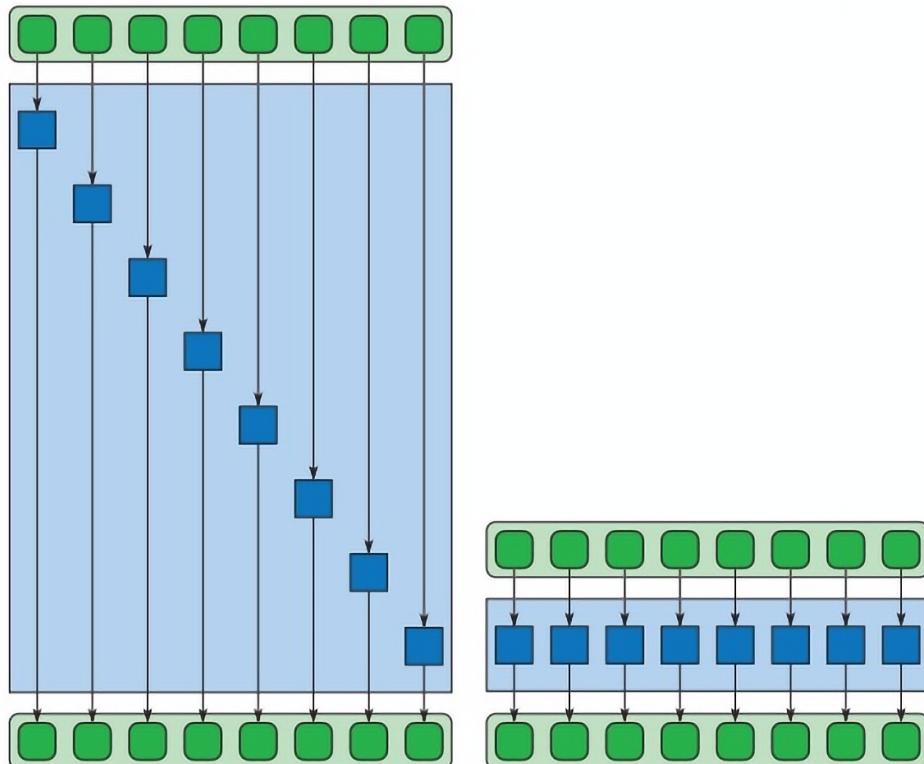


Figura 2.1: Ejecución en serie y en paralelo del patrón `map`

Reduce

El patrón **reduce** consiste en combinar todos los elementos de una colección de datos en un solo resultado mediante una función de reducción asociativa como se puede examinar en la figura 2.1.4. Debido a la asociatividad de la función combinatoria las tareas se pueden distribuir de diferentes formas y si la función es commutativa, habría muchas más posibilidades [11].

Para que un operador sea considerado de reducción tiene que poder reducir la colección de datos a un solo valor escalar, y el resultado final debe poder obtenerse a partir de los resultados de las tareas parciales que se crearon.

Algunos operadores que cumplen estos requisitos son la suma, la multiplicación y algunos operadores lógicos.

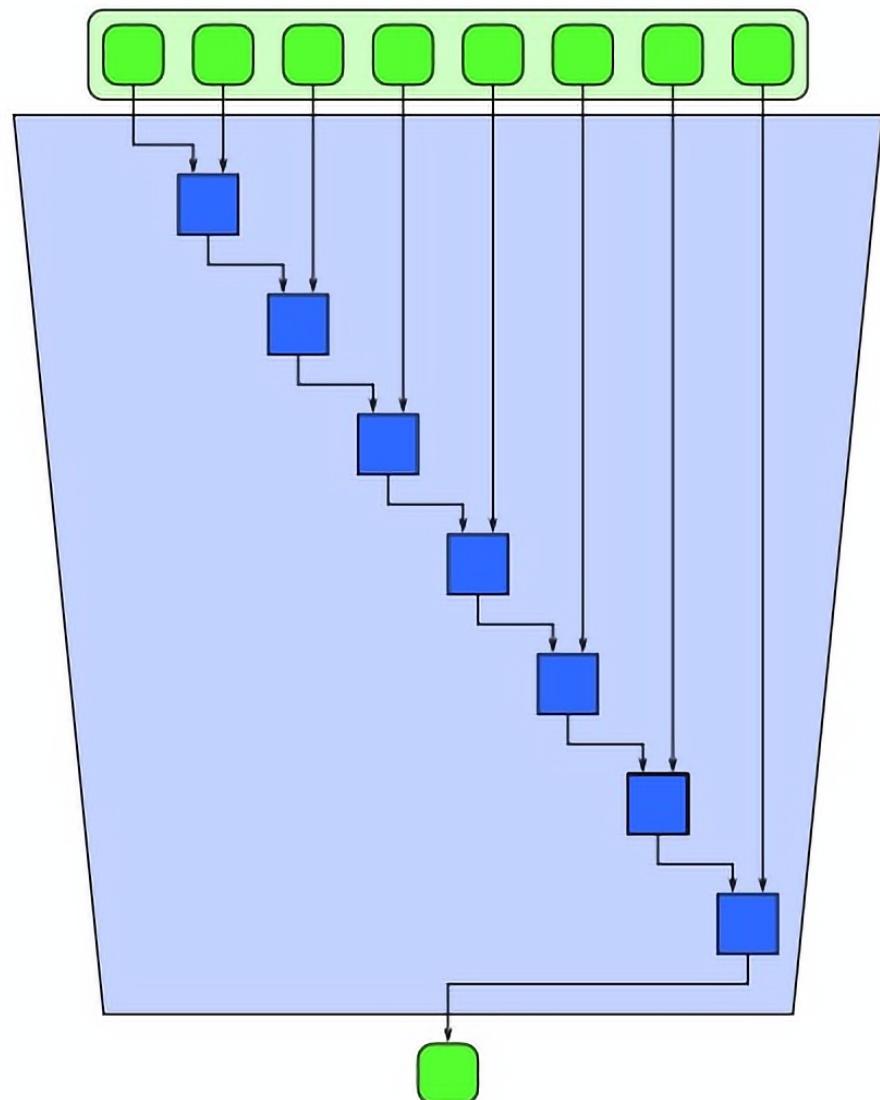


Figura 2.2: Ejecución en serie del patrón Reduce

Este patrón se puede **parallelizar** empleando una **estructura de árbol**, debido a que depende de un reordenamiento de operaciones combinadas por asociatividad. Y para conseguir una implementación eficiente, se pueden hacer uso de árboles poco profundos con grandes *fan-outs* y utilizar un único árbol para combinar los resultados de los trabajadores [11], como se puede observar en la figura 2.1.4.

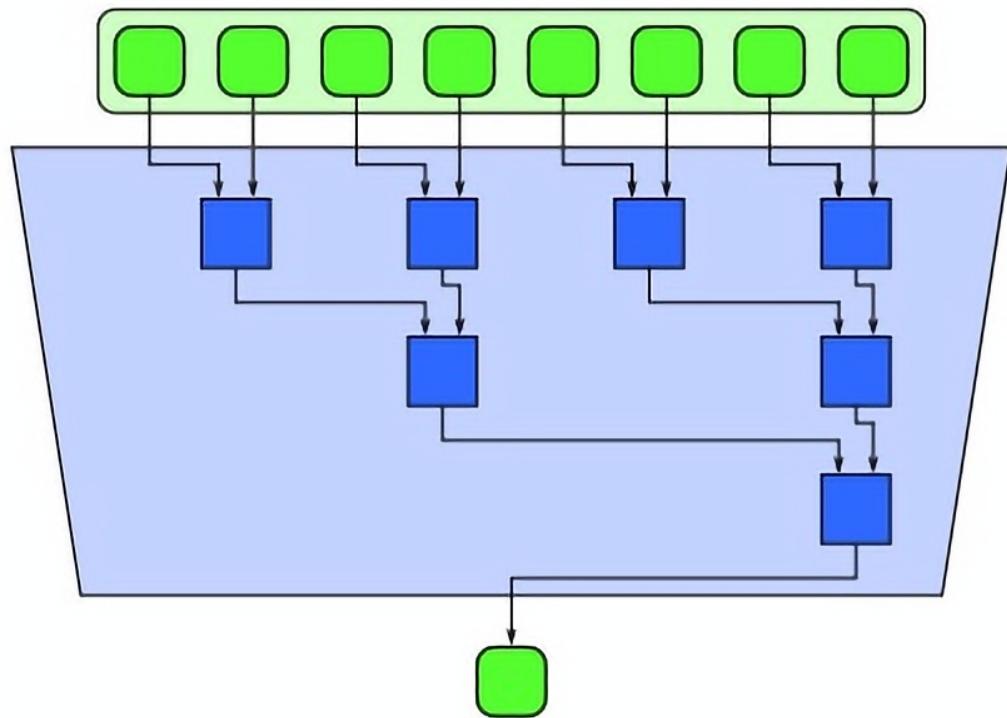


Figura 2.3: Ejecución en paralelo del patrón Reduce haciendo uso de árboles

MapReduce

Un `map` seguido de un `reduce` se puede optimizar fusionando los cálculos del `map` con las etapas iniciales de un cálculo de reducción [12]. Esto se le conoce como MapReduce.

Esta optimización elimina la necesidad de una sincronización después del `map` y también evita la necesidad de escribir la salida del `map` si no se utiliza en otro lugar.

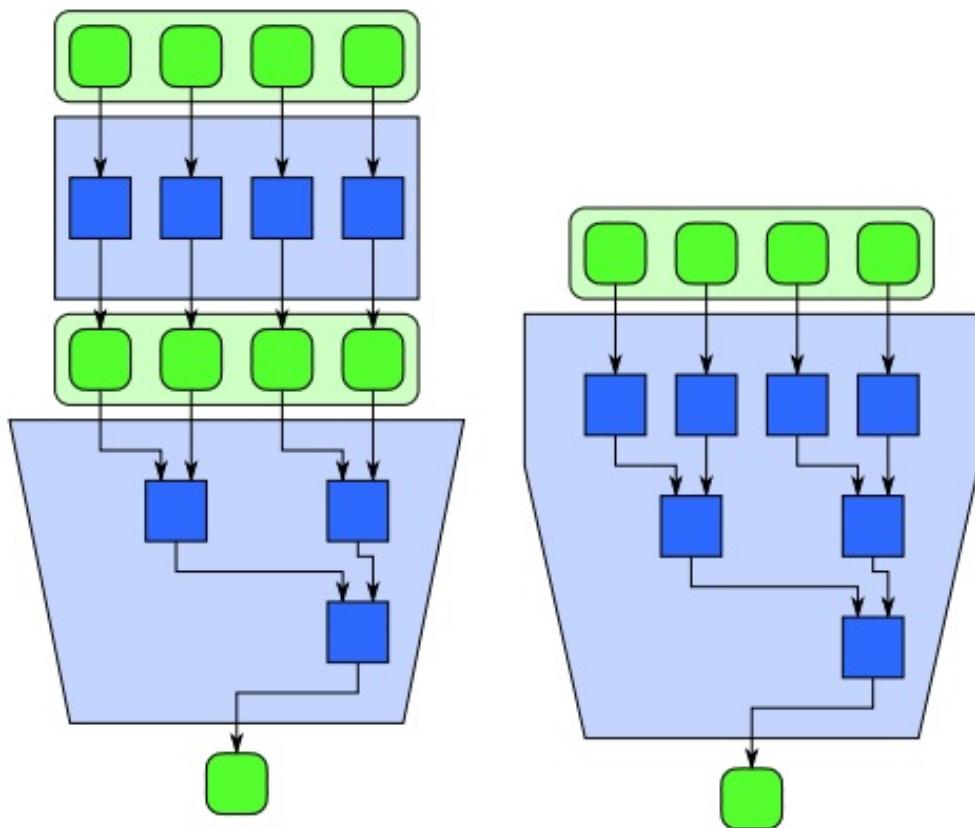


Figura 2.4: Ejecución del patrón `mapReduce`. Cuando después de un `map` ocurre una reducción, la combinación puede implementarse de una forma más eficiente dándole el resultado de aplicar el mapa como datos de entrada al patrón `reduce`.

Scan

Scan calcula todas las reducciones parciales de una colección, es decir, para cada posición de salida, se calcula una reducción de la entrada hasta ese punto.

En general este patrón es un caso especial del patrón en serie llamado `fold`. Cuando utilizamos `fold`, se emplea una función sucesora f para avanzar del estado anterior al estado actual, dada una entrada.

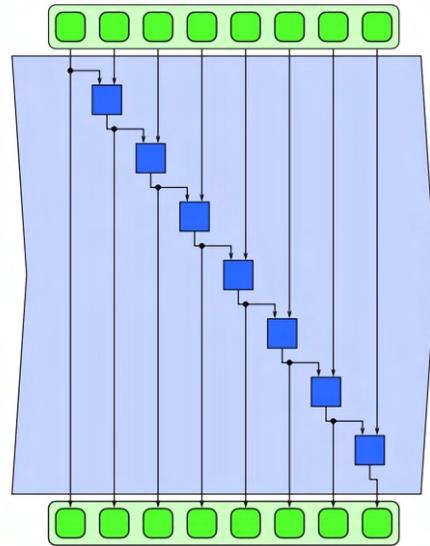


Figura 2.5: Ejecución en serie del patrón `scan`

La función sucesora debe ser asociativa para poder realizar el patrón `scan`, debido a que esto permite reordenar las operaciones para reducir el intervalo y posibilita una implementación paralela.

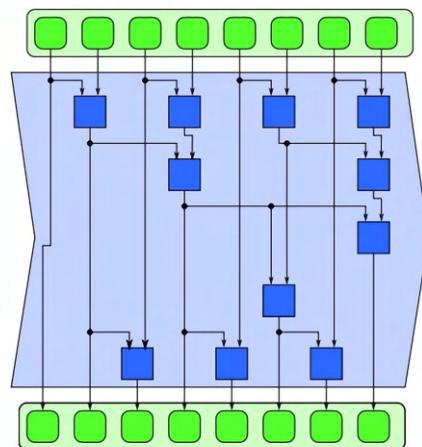


Figura 2.6: Ejecución en paralelo del patrón `scan`

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:

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

Y una matriz K cuyos coeficientes de convolución son:

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

Entonces, la función stencil sería la siguiente:

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

Esto produciría que el valor que devuelve la matriz M sea un escalar.

Viendo la figura 2.1.4, se puede realizar el stencil de manera secuencial, pero como este patrón es un derivado de la función `map`, podemos paralelizarlo de una manera muy eficiente.

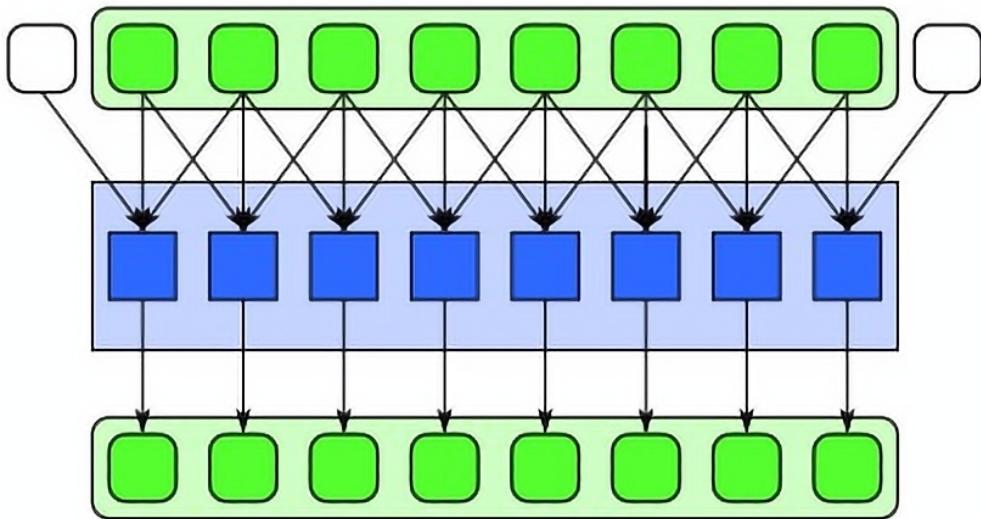


Figura 2.7: Ejecución del patrón stencil

2.1.5. 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. En virtud de las enormes diferencias entre las arquitecturas de una CPU con una GPU anteriormente explicadas, es necesario reformular 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[14].

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 son muy complejas, a diferencia de otras APIs con su misma funcionalidad.

Además, esta generalización que tiene OpenCL con respecto a otras APIs más

enfocadas a un hardware en específico, 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 [15].

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[16] de NVIDIA.

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

2.1.6. Programación funcional paralela

Como se ha visto en la introducción (Capítulo 1), la programación paralela se puede atacar desde distintos paradigmas, pero hay uno de ellos que ha destacado en este ámbito y es el uso de la programación funcional.

El paradigma funcional tiene la filosofía de 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[17].

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 quieras comerte, mientras que los lenguajes declarativos te suministran fotos del pastel preparado.

Por consiguiente, la programación funcional paralela se refiere al uso de la programación funcional junto con el paralelismo para trabajar con la programación declarativa de maneras específicas [18]. El trabajar con el paradigma funcional tiene una serie de propiedades muy interesantes (en las que se discutirá más a fondo en la siguiente sección). Una de las ventajas más importantes de la que nos podemos beneficiar los programadores es que la mayor parte del código que escribamos está hecho a base de funciones puras que operan en objetos inmutables, por lo que la concurrencia no presenta riesgos y el paralelismo es trivial, a diferencia de otros paradigmas como el imperativo, que necesita un estado mutable compartido que requiere una sincronización cuidadosa de bloques críticos y una unión explícita de hilos [19].

Esto hace que la seguridad en la lectura y escritura de datos en memoria sea muy buena, siendo esta una de las mayores preocupaciones en el procesamiento paralelo.

Uno de los lenguajes de programación funcionales más conocidos por hacer uso del paralelismo es **Scala**, que es un lenguaje funcional orientado a objetos, basado en la **JVM** (Java Virtual Machine). Gracias a poseer las características del paradigma funcional, este lenguaje es muy apropiado para su uso en entornos de Big Data y Data Science.

Una de las herramientas más utilizadas en el ámbito del Big Data es **Spark**, un motor implementado en **Scala** para la ejecución de procesos de Machine Learning y Data Science en máquinas de un sólo nodo o en agrupamientos.

Otro lenguaje de programación que debo mencionar es **Futhark**, es un pequeño lenguaje con una sintaxis muy parecida a Haskell diseñado para usar la potencia de cómputo de la GPU a través de **CUDA** y **OpenCL** o código de CPU de subprocesos múltiples para acelerar los cálculos de matrices de datos en paralelo [20].

Finalmente, se va a hablar de Haskell, un lenguaje de programación funcional que además de venir de manera nativa **parallel** con el que se pueden realizar programación paralela, también existen librerías como **Repa**, un módulo especializado en realizar cálculos de matrices paralelizando CPU, y **Accelerate**, un paquete para realizar operaciones sobre matrices empleando paralelismo sobre GPU en gráficas NVIDIA que soporten CUDA. Este lenguaje junto a sus librerías son las que iremos explicando en este documento por las numerosas propiedades que se explicarán a continuación.

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 [21].

2.2.1. Características principales de Haskell

Haskell tiene estas características muy interesantes:[22]

- **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 quieras ¹.

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.

¹Puede ser más engoroso mantener un sistema flexible de tipos para todo

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 [23].

En esta sección se va 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`, aunque la variable `Z` la explicaremos con más detenimiento posteriormente. 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, se guarda 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[23].

Las matrices retrasadas se representan mediante el tipo `Delayed (D)`. Un ejemplo de este tipo de matrices puede ser el siguiente:

Array D (Z :. 10) Int

Transformación entre ambas matrices

Algunas funciones básicas que se pueden realizar con Repa pueden producir una matriz diferida como salida, por lo que si desea tenerlo guardado en memoria, deberemos calcular el valor concreto de cada elemento y transformarlo en un array de manifiesto. Uno de los ejemplos más conocidos es la función `map` del 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 declarada como parámetro. Como siempre va a devolver una matriz retrasada y como hemos explicado aún no es un array, para convertirlo en una matriz, tenemos que llamar a una función que asigna la matriz y calcula el valor de cada elemento. La función `computeS` hace esto por nosotros:

```
computeS :: (Load r1 sh e, Target r2 e)
           => Array r1 sh e
           -> Array r2 sh e

> let arr = fromListUnboxed (Z :. 7) [1..7] :: Array U DIM1 Int
> Repa.computeS $ Repa.map (+1) arr :: Array U DIM1 Int
AUnboxed (Z :. 7) [2,3,4,5,6,7,8]
```

Extracto de código 2.1: Ejemplo de uso de la función map

Esta función lo que realizará es evaluar los elementos del array de manera secuencial, pero puede que se esté preguntando el motivo de que este tipo de funciones devuelvan una matriz retrasada en vez de devolver si mismo tipo de representación.

La contestación a esta pregunta es que al representar el resultado de estas funciones como matrices retrasadas, se puede realizar una secuencia de operaciones evitando construir matrices intermedias (recordamos que las matrices retrasadas no son realmente matrices); esta optimización es llamada fusión y es fundamental para lograr un buen desempeño en Repa [24].

```

> let arr = fromListUnboxed (Z:.7) [7,6..1] :: Array U DIM1 Int
> Repa.computeS . Repa.map (*2)
              $ Repa.map (+1) arr :: Array U DIM1 Int
AUnboxed (Z :. 7) [16,14,12,10,8,6,4]

```

Extracto de código 2.2: Ejemplo de uso de 2 maps consecutivos

Sin embargo, si queremos ejecutar estas funciones de forma paralela, Repa nos proporciona una variante del `computeS`, llamado `computeP`:

```

computeP :: (Load r1 sh e, Target r2 e, Source r2 e, Monad m)
          => Array r1 sh e -> m (Array r2 sh e)

```

Mientras que `computeS` ejecuta la matriz de manera secuencial, `computeP` usa los núcleos disponibles de nuestra CPU para poder realizar los cálculos en paralelo.

El tipo es muy parecido al `computeS`, exceptuando de que `computeP` es una función monádica. No importa que mónada se use (IO, Either,etc) debido a que el propósito de que esté envuelto en una mónada es solamente para asegurar que las operaciones de computación se lleven a cabo en secuencia y no se produzcan anidaciones[24].

Una característica muy importante a tener en cuenta de Repa es que no soporta **paralelismo anidado**, es decir, una llamada a `computeP` no puede hacer referencia a otra matriz calculada con `computeP`, a menos que la evaluación interna ya haya sido evaluada [24], por lo que el requisito de que esté envuelta en una mónada esa función está para evitar el problema

En cambio, si queremos convertir un array de manifiesto en una representación retrasada, lo único que tienes que hacer es usar la función `delay` para ello:

```

delay :: Shape sh
       => Source r e
       => Array r sh e
       -> Array D sh e

```

Esta función envuelve la representación interna para que sea una función de índices de elementos con un tiempo de complejidad de $\mathcal{O}(1)$ [25].

Dimensionalidad e índices

Para construir la dimensionalidad (`Shape`) tenemos dos tipos de constructores, `(Z)` y `(:.)`:

```

data Z = Z
data tail :: head = tail :: head

```

Si escribimos como tamaño `Z`, es la dimensionalidad de un array con cero dimensiones (ejemplo, un escalar). Si añadimos un valor entero `Z :. Int`, nos encontramos con un vector, si agregamos otro valor entero `Z :. Int :. Int`, nos encontramos con una matriz (o un array bidimensional). Así que para ir añadiendo nuevas dimensiones solamente

tenemos que agregar `:: Int` a la derecha. Un ejemplo de matriz bidimensional de tamaño 3x3:

```
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. Podemos utilizar el tipo `Shape` para poder indexar una matriz, por lo que si queremos acceder al primer elemento de este array, se deberá utilizar la siguiente función:

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

Es importante recalcar que cuando se quiere acceder a cualquier elemento de la matriz, el primer elemento que se empieza a contar es desde el cero. por lo cual, si se quiere acceder al primer elemento de una matriz 3x3 deberemos acceder al `(Z :. 0 :. 0)`, por ejemplo:

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

Como se sabe que la forma de guardar internamente una matriz en memoria es independiente de su forma, incluso se puede cambiar la forma sin tener que copiar la matriz haciendo uso de 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 se ha visto en ejemplos anteriores, se puede 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 se especifica el tamaño y dimensionalidad de nuestro array y posteriormente, la lista de valores que tomará nuestro array.

Y si lo que se quiere es crear directamente matrices retrasadas, se puede realizar mediante la función `fromFunction`:

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

Donde el primer argumento toma el tamaño y dimensionalidad que tendrá nuestro array 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 arrays, 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.
- **zip:** Dada dos matrices de manifiesto de tipo `Unboxed`, devuelve un array de pares correspondientes en un tiempo $\mathcal{O}(1)$. Existen versiones de esta función hasta 6 matrices.
- **unzip:** Dado un array de manifiesto de tipo `Unboxed` donde cada elemento es una tupla, transforma esa matriz de pares en una matriz de los primeros componentes y una matriz de los segundos componentes en un tiempo $\mathcal{O}(1)$. Existen estas funciones hasta 6-tuplas.
- **traverse:** Dado un array de entrada, una función que transforma la dimensionalidad del nuevo array y otra función que toma como argumento una función de indexación al original y un índice en la nueva matriz y se encarga de calcular el valor en el índice en la nueva matriz, la salida correspondiente será la generación de una matriz retrasada recorriendo la matriz existente de entrada.
- **foldAllP:** Dado un operador secuencial asociativo y una matriz de rango arbitraria realiza una reducción paralela a un solo valor escalar.

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[26]. 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, se pueden realizar estas operaciones de forma paralela para poder conseguir un mejor rendimiento en nuestros algoritmos ².

²Si no se tienen conocimientos de lo que es un stencil, revisar la sección de patrones de diseño paralelo

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 [27]. Si se quisiera generar el stencil de Laplace mostrado en la siguiente matriz:

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

Se tendría que escribir como el código 2.3::

```
laplace :: Stencil#(Z, Z)
laplace = makeStencil((Z(3) :. 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 2.3: 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 hay que escribir la siguiente función:

```
laplace :: Stencil#(Z, Z)
laplace = [stencil2| 0 1 0
           1 0 1
           0 1 0 |]
```

Extracto de código 2.4: Stencils usando QuasiQuotes

Especificación de condiciones de contorno en Stencils

Cuando se encuentran elementos que se escapan fuera de los límites de nuestro array, se ha de declarar una condición para determinar que valores van a tomar cada elemento fuera del array. En Repa, se encuentra el tipo de datos **Boundary**, que se será muy útil para poder especificar esta condición.

Seguidamente, se va a explicar las diferentes formas de especificar una condición en los Repa stencils:

- **BoundClamp:** otorga a los píxeles del borde los mismos valores que en la imagen original
- **BoundConst:** trata a los puntos fuera de la matriz como si se les hubiera dado un valor constante.

- **BoundFixed**: emplea un valor constante como resultado en cualquier momento en que la convolución dependa de un valor fuera de la matriz.

En conclusión, `BoundClamp` y `BoundConst` no cambiarán la imagen del todo, pero `BoundFixed` dibujaría un borde con un valor `a` en la imagen, por lo que se recomienda utilizar este último solamente cuando se quiera dibujar bordes con un valor constante.

2.3.4. Ejemplo: Producto escalar

Para finalizar la sección, se va a mostrar la adaptación de un algoritmo desarrollado en Haskell hacia el módulo Repa haciendo uso de las funciones de paralelismo sobre CPU. En este caso, se ha decidido hacer uso del algoritmo del **producto escalar**.

El producto escalar es una operación que multiplica por pares 2 vectores y posteriormente suma los resultados. Por ejemplo, dado dos vectores $\mathbf{u} = (u_1, u_2, \dots, u_n)$ y $\mathbf{t} = (t_1, t_2, \dots, t_n)$, su producto escalar se define por:

$$\mathbf{u} \cdot \mathbf{t} = (u_1 \cdot t_1 + u_2 \cdot t_2 + \dots + u_n \cdot t_n)$$

Algoritmo en Haskell (secuencial)

Como se puede observar del código 2.5, la función `zipWith` aplica la función (*) por pares en cada miembro de las dos listas dadas, luego suma el resultado del `zipWith` para que solamente quede un escalar mediante el uso de la función `foldl`.

```
scalarmp :: [Float]
          -> [Float]
          -> Float
scalarmp xs ys = P.foldl (+) 0 (P.zipWith (*) xs ys)
```

Extracto de código 2.5: Producto escalar de forma secuencial en Haskell

Algoritmo en Repa (Paralelizando CPU)

Después de haber realizado el algoritmo secuencial del producto escalar en Haskell puro, es momento de realizar esta tarea en Repa. El algoritmo va a ser el mismo, solamente que se va a cambiar algunas funciones que tiene Repa para realizarlo de forma paralela.

En el código 2.6 se puede observar que en Repa, estamos usando la función `zipWith` de Repa, que tiene el mismo funcionamiento que el de Haskell puro, pero para realizar el `fold` en Repa, se ha hecho uso de la función `sumAllP`, que dado un operador secuencial asociativo y una matriz, realiza una reducción paralela a un solo valor escalar. Además, se puede examinar que esta función devuelve una función monádica como `computeP` para evitar el paralelismo anidado.

```

scalarpRepa :: (Monad m)
              => Array#(U,DIM1,Float)
              -> Array#(U,DIM1,Float)
              -> m#(Float)
scalarpRepa xs ys = sumAllP(R.zipWith(*),xs,ys)

```

Extracto de código 2.6: Producto escalar en Repa (paralelizando CPU).

Rendimiento de ambos algoritmos

En este apartado se ha realizado unas pequeñas pruebas de rendimiento para ver como de eficiente es el adaptar nuestro algoritmo de Haskell puro a Repa.

- **2.000.000 elementos:**

- **Haskell** (secuencial): 2.25 segundos
- **Repa** (paralelizando CPU): 0.88 segundos

- **5.000.000 elementos:**

- **Haskell** (secuencial): 2.42 segundos
- **Repa** (paralelizando CPU): 1.86 segundos

- **10.000.000 elementos:**

- **Haskell** (secuencial): 5.06 segundos
- **Repa** (paralelizando CPU): 3.69 segundos

Viendo las pruebas de rendimiento, se puede llegar a la conclusión que el haber adaptado nuestro algoritmo a Repa, nos ha permitido unos mejores rendimientos, que si se hiciera con Haskell puro.

2.4. Aceleración GPU con Accelerate

Accelerate es un lenguaje específico de dominio incrustado (eDSL) desarrollado principalmente por Trevor L. McDonnell para poder realizar operaciones sobre matriciales multidimensionales de manera eficiente sobre hardware que soporte la parallelización masiva de datos, como una GPU. Esto nos permite hacer uso de la enorme potencia que tienen las tarjetas gráficas escribiendo el código en Haskell y poderlo ejecutar directamente en la GPU sin tener que escribir código CUDA [28].

El módulo Accelerate nos proporciona una infraestructura básica que incluye Data.Array.Accelerate, un módulo para construir cálculos de matrices, donde existen muchas funciones con la misma funcionalidad que Repa, además de Data.Array.Interpreter para poder interpretar los cálculos en Haskell.

Para poder ejecutar realmente una computación en Accelerate sobre 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 quisiéramos ejecutar una computación en Accelerate paralelizando CPU en vez de la tarjeta gráfica, podemos realizarlo gracias al módulo `accelerate-llvm-native`.

2.4.1. Características principales de Accelerate:

Interfaz abstracta

En Accelerate 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.

No soporta paralelismo anidado

Accelerate está restringido a programas que expresan solo paralelismo de datos planos, en los que el cálculo de cada elemento de datos debe ser secuencial. En la práctica, esto limita severamente las aplicaciones de la computación paralela de datos, debido a que no se nos permite construir una matriz de matrices [29].

Optimizaciones

Para conseguir un rendimiento competitivo al código CUDA escrito a mano, Accelerate utiliza una serie de optimizaciones escalares y de matrices, en las que se incluye la fusión de matrices. La fusión de un programa implica combinar bucles sobre una matriz en un solo recorrido, reduciendo el tráfico de memoria y eliminando la creación de matrices intermedias [30].

2.4.2. Arrays, dimensionalidad 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 arrays. 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 array 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 forma 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 **(.:)**:

```
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 arrays 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
```

Arrays en el host y en el dispositivo

Accelerate es un lenguaje DSL integrado (un lenguaje de programación dentro de otro lenguaje) que distingue entre matrices *vanilla* de Haskell y arrays en el lenguaje Accelerate, así como cálculos en ambos tipos de matrices.

Los cálculos de matrices integrados en el lenguaje Accelerate se identifican mediante tipos formados por el constructor de tipo **Acc**, y deben ejecutarse explícitamente antes de que surja efecto. Para poder que una matriz en *Vanilla* Haskell esté disponible para su procesamiento dentro de los cálculos integrados de Accelerate, deberemos utilizar la función **use**:

```
use :: forall arrays. Arrays arrays
=> arrays
-> Acc arrays
```

Aquí tenemos un ejemplo de uso:

```
let vec = fromList (Z :: 5) [0..] :: Vector Int
let vec' = use vec :: Acc (Vector Int)
```

En el contexto de la programación GPU, esta suele tener su propia memoria de alto rendimiento que está separada de la memoria principal de la CPU anfitriona, por lo que los datos deben transferirse a la GPU si se quieren utilizar. En consecuencia, la distinción entre los arrays regulares del tipo `Array sh e` y los arreglos del lenguaje integrado `Acc (Array sh e)` tiene el beneficio de diferenciar entre los cálculos que se ejecutan en el host (la CPU) y los que se ejecutan en la GPU[29].

Una cosa a tener en cuenta es que al tener las memorias separadas, la transferencia de datos tiene un coste computacional, por lo que es mejor mantener los datos en la GPU lo máximo posible.

Cálculos de matrices frente a expresiones escalares

La mayoría de las operaciones que podemos realizar en Accelerate tienen como entrada y salida arrays envueltos en `Acc`. Además del tipo de datos que nos señala los arrays que están en la memoria de la GPU, también nos encontramos con el tipo `Exp`, que define los cálculos escalares dentro de las matrices aceleradas por GPU.

Un ejemplo sería `Exp Float` que representa una expresión integrada en Accelerate que produce un valor de tipo `Int`.

Al igual que los cálculos realizados por `Acc`, los cálculos en `Exp` se ejecutan en el idioma de destino de la GPU.

Para conseguir un lenguaje estratificado, Accelerate distingue entre los tipos de operaciones colectivas y cálculos escalares. Estas operaciones colectivas en `Acc` consisten en muchos cálculos escalares en `Exp` que se ejecutan en paralelo a los datos, sin embargo, los cálculos escalares no pueden contener operaciones colectivas.

Esta estratificación que restringe el lenguaje Accelerate a solo realizar paralelismo de datos planos nos garantiza que los cálculos se puedan asignar de manera eficiente en el hardware.

Índices

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:.)
25

```

En cambio, si se quiere acceder a un elemento de una matriz del lenguaje Accelerate (esté almacenada en la GPU), se debe utilizar el operador (!)

```

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

```

2.4.3. Ejecución de un cálculo acelerado simple

Hasta ahora se han realizado experimentos con matrices en el contexto de código Haskell, sin embargo, en ningún momento se han ejecutado programas en paralelo sobre GPU o CPU.

Por lo cual, para poder realizar un cálculo acelerado debemos emplear la siguiente función:

```

run :: Arrays a
      => Acc a
      -> a

```

Esta función compila y ejecuta una función de una matriz integrada en el lenguaje Accelerate, produciendo un resultado que posteriormente se guardará en Vanilla Haskell.

Existen 3 versiones de la función `run` que podemos exportar:

- `Data.Array.Accelerate.Interpreter` que es usada para experimentaciones y pruebas.
- `Data.Array.Accelerate.LLVM.Native` (del módulo `accelerate-llvm-native`) que ejecuta los cálculos en un procesador multinúcleo.
- `Data.Array.Accelerate.LLVM.PTX` (del paquete `accelerate-llvm-ptx`) llevan a cabo la ejecución de las expresiones paralelizando sobre una GPU NVIDIA.

Probemos un ejemplo muy simple:

Tenemos una matriz de dimensión 3x3 donde cada uno de los elementos es de tipo `Int`, y queremos multiplicar por dos cada elemento del array:

```

> let arr = fromList (Z:.3:.3) [1..] :: Array DIM2 Int
> run $ Accelerate.map (*2) (use arr)
 $\text{Array } (Z \text{ :. } 3 \text{ :. } 3) [2, 4, 6, 8, 10, 12, 14, 16, 18]$ 

```

Desglosando esto, en primer lugar podemos ver que llamamos a la función `map` de `Accelerate`:

```

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

```

Donde se aplica la función dada por elementos a una matriz que está integrada dentro de los cálculos integrados de `Accelerate`. Para conseguir la integración de una matriz recordemos que la función `use` se encarga de ello.

Y para que esto se realice de forma paralela sobre GPU, la función `run` del paquete `accelerate-llvm-ptx` se encargará de llevar a cabo esa tarea y de devolvernos el resultado en Vanilla Haskell.

2.4.4. Generando arrays

Al igual que en la librería `Repa`, existen funciones para construir arrays 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, el querer utilizar estas funciones y luego integrarlas en el lenguaje `Accelerate` empleando la función `use` está bien, pero es una manera muy ineficiente de crear matrices, debido a que tenemos que copiar la matriz en la memoria de la GPU. Por lo que lo mejor es crear arrays dentro del lenguaje `Accelerate`, la más genérica de todas es hacer uso de 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 array, y el segundo parámetro le indicamos la función para que calcule cada elemento del array. Para entenderlo mejor daremos el siguiente ejemplo:

```
> run $ generate (I1 10) (\(I1 i) -> i + 1) :: Vector Int
Vector (Z :: 10) [1,2,3,4,5,6,7,8,9,10]
```

2.4.5. Lifting y Unlifting

Un valor de tipo `Int` es un valor simple de Haskell, en cambio, `Exp Int` es un valor elevado, es decir, un entero elevado al dominio de las expresiones integradas de Accelerate.

Por lo que, al existir diferencias entre tipos, cuando se quiera trabajar con índices o tuplas puede ser conveniente conocer estas dos funciones:

- **unlift**: esta función de Accelerate nos permite desenvolver un valor estructurado dentro de un `Exp`.
- **lift**: envuelve un valor estructurado dentro de un `Exp`.

Para entender mejor como funcionan estas funciones, lo veremos en un ejemplo:

```
> let sh = constant (Z :: 3 :: 3) :: Exp DIM2
> let Z :: x :: y = unlift sh      :: Z :: Exp Int :: Exp Int
> let u = lift (x,y)            :: Exp (Int, Int)
```

2.4.6. 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 cálculos.

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 bidimensional con código [2.7](#):

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

Extracto de código 2.7: 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

Al igual que en Repa, Accelerate tiene el tipo de datos `Boundary` para poder especificar una condición para determinar que valor va a tomar cada elemento fuera de la matriz. 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 píxel 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 extensión de nuestro array como el reflejo de la matriz.

Para entenderlo, podemos tener como ejemplo un stencil 5x3, en la cual el elemento fuera del array 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.

$a \ bc$ <pre>+-----+ +-----+ d ef ef d g hi hi g bc a +-----+ +-----+</pre>	\rightarrow	$d $ $g $ $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.

2.4.7. Ejemplo: Producto escalar

Para finalizar el capítulo, se va a exponer la evolución de un algoritmo realizado en Haskell puro al lenguaje **Accelerate**, llevando a cabo el paralelismo sobre GPU.

Para realizar esta tarea, se ha decidido hacer el mismo ejemplo que se empleó en la sección de Repa, el producto escalar.

El algoritmo realizado en Haskell puro es el mismo que el realizado en Repa con código [2.5](#).

Examinando el algoritmo realizado en Accelerate con código [2.8](#), se puede observar que el `zipWith` hace la misma función que realiza en Haskell puro, pero el funcionamiento del `fold` es distinto. En este caso no realizamos un plegado por la derecha o izquierda del vector, si no que esto se realiza en paralelo, empleando árboles de forma interna como explicamos en la sección de diseño de patrones paralelos.

```
scalarmpAcc :: Acc (Vector Float)
  -> Acc (Vector Float)
  -> Acc (Scalar Float)
scalarmpAcc xs ys = fold (+) 0 $ A.zipWith (*) xs ys
```

Extracto de código 2.8: Producto escalar en Accelerate (paralelizando GPU).

Rendimiento de ambos algoritmos

En esta sección se ha realizado unas pruebas de rendimiento para mostrar la eficiencia de adaptar nuestros algoritmos de Haskell al lenguaje Accelerate.

- **2.000.000 elementos:**
 - **Haskell** (secuencial): 2.25 segundos
 - **Accelerate** (paralelizando GPU): 0.43 segundos
- **5.000.000 elementos:**
 - **Haskell** (secuencial): 2.42 segundos
 - **Accelerate** (paralelizando GPU): 0.34 segundos
- **10.000.000 elementos:**
 - **Haskell** (secuencial): 5.06 segundos
 - **Accelerate** (paralelizando GPU): 0.46 segundos

Viendo el rendimiento que tiene Accelerate sobre Haskell puro, se puede decir que el haber transformado nuestro algoritmo a Accelerate, nos ha hecho conseguir unos mejores rendimientos que usando el lenguaje Haskell.

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 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`.

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). Después, se llevó a cabo el desarrollo de unas funciones que nos permitieran leer y escribir vídeos. Seguidamente, 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.

Hay que tener en cuenta de que los algoritmos que se van a desarrollar en esta sección, están implementadas para realizarse en imágenes en escala de grises, por lo que si queremos realizar algunos en imágenes RGB, lo único que tenemos que hacer es repetir esta función por cada banda de la imagen (en este caso 3 veces).

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.

Las imágenes RGB en Repa es una colección de bandas donde cada una de ellas es un array bidimensional.

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.

Las imágenes RGB en Accelerate las tratamos como una matriz bidimensional en la cual cada elemento es una terna con los 3 valores RGB.

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. Lectura y escritura de vídeos

Debido a que trabajar con *codecs* de vídeo es complicado en el lenguaje Haskell, esto puede explicar que Accelerate y Repa no tengan disponibles módulos para leer y escribir vídeos en sus librerías, por lo que para conseguir nuestro objetivo de leer vídeos, tuvimos que recurrir a la librería [ffmpeg-light](#), que tiene los enlaces mínimos a la biblioteca FFmpeg, una colección de software que permite grabar y convertir vídeos a otros formatos.

Se ha hecho uso de este módulo debido a que el resto de soluciones que existen en Haskell están incompletas o funcionan muy mal. Además, que esta librería transforma los fotogramas a JuicyPixels, cosa que nos puede ayudar para evitar transformaciones extras si tuviera su tipo de datos específico.

En primer lugar, se realizó la instalación de la biblioteca FFmpeg en nuestro sistema debido a que es una dependencia externa de ffmpeg-light. Sin embargo, no podíamos instalar el módulo debido a que en sistemas derivados de Ubuntu y Debian según la documentación oficial, la biblioteca FFmpeg no era compatible con el módulo, por lo que nos proporcionaban un *script* en Bash para instalarlo.

No obstante, seguía dando errores, por lo que después de hablar con el creador oficial de este módulo ¹, lo que me recomendó es mirar los paquetes que instala la integración continua del módulo ² para realizar una instalación correcta. Después de probar esto, se pudo instalar el módulo correctamente.

Para el desarrollo de funciones de lectura y escritura de vídeos en nuestro proyecto tuvimos que investigar en distintas fuentes de información, esto es debido a que la documentación existente de la librería es muy escasa. Por lo que, después de encontrar unas funciones muy parecidas a lo que queríamos hacer ³ ⁴, con los nuevos conocimientos se adaptaron esos algoritmos para que funcionen con el tipo de imágenes que estábamos trabajando.

Cuando se desarrolló la función de escribir un vídeo a partir de imágenes JuicyPixels, solamente soportaba el esquema de color RGB8, por lo que se decidió modificar la función de grabar vídeos para que pueda soportar todos los tipos de JuicyPixels que sea capaz de leer FFmpeg: Pixel8 (sinónimo de Word8), PixelRGB8 y PixelRGBA8.

3.3. 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[31].

Esto significa que el histograma de una imagen en escala de grises cuenta todos los píxeles de una imagen de acuerdo con los distintos colores de grises posibles (256 distintos niveles de grises en una profundidad de 8 bits) y cuenta la frecuencia con la que aparece cada distinto color.

Para entender mejor los conceptos de generación del histograma se pondrá un ejemplo a continuación:

Una imagen en escala de grises se compone únicamente de 7 píxeles [1,1,2,0,1,0,3], el histograma asociado a esta imagen sería:

```
> imagen = [1,1,2,0,1,0,3]
> histograma(imagen)
[2,3,1,1,0,0,0....0]
```

Donde el píxel 0 aparece 2 veces, el píxel 1 se manifiesta 3 veces y si nos damos cuenta, a partir del valor 4 ya todos son 0, debido a que no aparecen en la imagen.

¹<https://github.com/acowley/ffmpeg-light/issues/66>

²<https://github.com/acowley/ffmpeg-light/blob/master/.github/workflows/ci.yml>

³<https://stackoverflow.com/questions/46896784/>

⁴<https://stackoverflow.com/questions/46942579/>

Este vector lo podemos representar como una gráfica donde cada posición es el tono de gris que representa y el valor que hay sería el número de veces que se repite ese gris en la imagen.

Debido a que estamos considerando imágenes RGB de 8 bits de profundidad cada banda, la generación de un histograma RGB sería una colección en la cual cada elemento sería un histograma en escala de grises generado por la banda del color correspondiente.

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 ⁵, 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 píxel, 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 súper ineficiente. Esto hizo que se descartase esta función.

Después de leer a varios desarrolladores, debates en internet, entre otros con el mismo problema ⁶, a la hora de realizar histogramas en Repa y de intentos fallidos de intentar agregar este algoritmo a este módulo, decidí hacer caso a la comunidad y utilizar una estructura de datos diferente a los matrices 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 ⁷:

- **Sequence:** es una estructura de datos que se basa internamente en *finger trees* ⁸ (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[32]), 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[33].

- **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 [34].

Además de que sus indexaciones son de tiempo $\mathcal{O}(1)$, la razón por la que se probó esta librería como estructura interna es que es la base de muchas otras librerías de

⁵<https://gist.github.com/hirschenberger/4079202>

⁶<https://github.com/TomMD/JuicyPixels-repa/issues/2#issuecomment-10357726>

⁷<https://stackoverflow.com/questions/9611904/>

⁸https://en.wikipedia.org/wiki/Finger_tree

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 $\mathcal{O}(1)$, por lo que me pareció interesante esa funcionalidad.

En total, descartando la versión alfa que se desarrolló anteriormente por motivos de rendimiento, se decidió realizar 3 distintas versiones donde dos de ellas se realizarán con el uso de la estructura de datos `Sequence` y la restante con `Vector`.

En esta nueva primera versión de generación de histogramas, se decide utilizar como acumulador la estructura de datos `Sequence`, y los cálculos se realizaron de manera secuencial.

Posteriormente, se decidió realizar una segunda versión con código 3.1 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. Para conseguir que este algoritmo vaya en paralelo, hemos recurrido al uso del módulo `Control.Parallel.Strategies` [35].

Mirando la documentación del paquete se puede percibir uno que la función `using` evaluará cada elemento de la lista en paralelo vía una estrategia, en este caso la estrategia que vamos a emplear es la de `rdeepseq`, que evalúa completamente la función que le pasemos.

```
generateHistogramsV2 :: ImgRGB Int -> Histograms
generateHistogramsV2 xs =
    let bs = Prelude.map generateHistogram xs
        cs = bs `using` parList rdeepseq
    in cs
```

Extracto de código 3.1: Generación de histogramas paralelizando el cálculo de las 3 bandas

Para terminar, se desarrolla una tercera versión con código 3.2, en el cual se hace uso del módulo `Vector` como acumulador de datos.

Para la ejecución de este algoritmo en primer lugar 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`.

Para paralelizar el cálculo de los 3 canales en paralelo junto a la generación de histogramas de forma simultánea, se hizo uso de la misma estrategia que en la versión 2.

Pero, para poder fusionar todos los histogramas generados por las filas de un canal en paralelo, se empleó un `fold` paralelo al que le denominamos `pfold`⁹.

⁹<https://stackoverflow.com/questions/19117922/parallel-folding-in-haskell>

```

generateHistogramsV3 :: ImgRGB Int -> [HVector]
generateHistogramsV3 xs =
  let bs = Prelude.map generateHist xs
      cs = bs `using` parList rdeepseq
  in cs

generateHist :: Channel Int -> HVector
generateHist band = pfold (V.zipWith (+)) (generateRows band)

generateRows :: Channel Int -> [HVector]
generateRows band =
  let (Z :: w :: h) = R.extent band
      bs = Prelude.map (generateRow band) [0..w-1]
      cs = bs `using` parList rdeepseq
  in cs

```

Extracto de código 3.2: Generación de histogramas haciendo el cálculo de cada fila de la imagen en paralelo, además de realizarlo para cada canal en paralelo

Histogramas en Accelerate

El algoritmo realizado en el código 3.3 ha empleado la función `histogram` para generar un histograma por cada banda de la imagen. Normalmente si lo ejecutamos en Accelerate Native o PTX debería realizar los histogramas de forma paralela, pero al no forzar la evaluación de las matrices, produce que las operaciones se fusionen y los tres bucles de ejecución paralelos se ejecuten secuencialmente en un solo kernel.

Por lo que, para forzar la evaluación de una expresión de un array, existe la función `compute` de Accelerate que evita la fusión entre las operaciones, permitiendo que la generación de histogramas se realice simultáneamente, reduciendo el tiempo de ejecución que si no usáramos esta función.

```

gHistogram :: Acc (Matrix (Pixel8, Pixel8, Pixel8))
-> Acc (Vector (Int, Int, Int))
gHistogram arr =
  let (r,g,b) = A.unzip3 arr
  in A.zip3 (compute $ histogram $ flatten $ promoteInt r)
             (compute $ histogram $ flatten $ promoteInt g)
             (compute $ histogram $ flatten $ promoteInt b)

```

Extracto de código 3.3: Generación de histogramas en Accelerate

3.4. Escala de grises

Una imagen en escala de grises es aquella en el valor 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[36].

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:

$$0,299 \cdot Rojo + 0,587 \cdot Verde + 0,114 \cdot Azul [37]$$

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 obtener la luminosidad de una imagen RGB, se desarrolla haciendo uso de la función `zipWith` de Repa para combinar la percepción de luz promedio del canal rojo y del canal verde. Posteriormente, se emplea la misma función para fusionar el canal resultante con el canal azul dando como resultado una imagen en escala de grises.

Al final se decidió descartar esta versión debido a que se puede conseguir mejores resultados si se realiza el cálculo de la luminosidad de todas las bandas a la vez, además, para que la función auxiliar fuera genérica se tenía que hacer uso de indexaciones de lista para obtener la constante con la que debe multiplicarse ese píxel, cosa que hacía muy ineficiente el algoritmo.

La segunda versión del algoritmo con código 3.4 se emplea el uso de la función `zip3` de Repa, que realiza una matriz de ternas, donde cada una de las ternas contiene elementos de las 3 bandas que ocurren en la misma posición. Gracias a esta propiedad de fusionar bandas, pudimos realizar una función `map` de forma paralela, calculando la luminosidad de cada terna, siendo más eficiente que la versión anterior.

```
toGrayScaleV2 :: ImgRGB Pixel8 -> IO (Channel Float)
toGrayScaleV2 [r,g,b] = R.computeP
    . R.map luminance
    $ U.zip3 r g b
toGrayScaleV2 _ = error "No hay bandas suficientes"
{-# NOINLINE toGrayScaleV2 #-}
```

Extracto de código 3.4: Algoritmo escala de grises (Repa V2)

Escala de grises en Accelerate

El algoritmo realizado en el código 3.5 ha empleado la función `map` nativo de Accelerate al que si lo ejecutamos con Accelerate Native (parallelismo CPU) o Accelerate PTX (paralelización mediante GPU) se va a realizar de forma paralela el cambio de RGB a escala de grises por cada píxel de la imagen.

```
grayScale :: Acc (Matrix (Pixel8,Pixel8,Pixel8))
            -> Acc (Matrix Float)
grayScale = Accelerate.map (\ pix ->
  let (r,g,b) = unlift pix
    r'        = A.fromIntegral r / 255
    g'        = A.fromIntegral g / 255
    b'        = A.fromIntegral b / 255
    in r' * 0.2989 + g' * 0.5870 + b' * 0.1140)
```

Extracto de código 3.5: Algoritmo escala de grises (Accelerate)

En vista de que por defecto la estructura que tienen las imágenes RGB en accelerate es de una matriz bidimensional donde cada elemento del array es una terna con los componentes RGB de esa posición, no se ha tenido que emplear ninguna función para fusionar los arrays como se tuvo que hacer en Repa.

3.5. 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 [38]. 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. En la figura 3.1 se puede ver los kernels de dimensión 3x3 y 5x5 con $\sigma = 1$.

$$\frac{1}{16} \times \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \quad \frac{1}{273} \times \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 7 & 4 & 1 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 7 & 26 & 41 & 26 & 7 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 1 & 4 & 7 & 4 & 1 \\ \hline \end{array}$$

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

En el desarrollo del algoritmo en ambas librerías, se decidió emplear el kernel 5x5 de la figura 3.1 para siempre intentar realizar pruebas con operaciones grandes.

El desenfoque gaussiano se puede aplicar a una matriz bidimensional como dos operaciones unidimensionales separables, es decir, el mismo efecto de emplear un kernel bidimensional (por ejemplo el 5x5 que se va a emplear), se puede lograr aplicando un kernel unidimensional para desenfocar la imagen en una dirección (horizontal o vertical) y posteriormente repetir el proceso en la dirección restante.

El efecto de la aplicación de ambas formas es la misma, con la diferencia que al emplear 2 vectores unidimensionales, vamos a requerir menos cálculos al hacer uso de kernels más pequeños, consiguiendo una mejor eficacia.

Para comprobar que el rendimiento cuando aplicas un kernel que es separable es verdadero, se ha decidido realizar 2 versiones, la primera empleando la propiedad separable del filtro gaussiano, y la segunda haciendo uso del kernel 5x5.

3.5.1. Filtro Gaussiano empleando 2 kernels

En el desarrollo de este algoritmo se va a emplear 2 kernels, el primero nos va a permitir desenfocar la imagen en la dirección horizontal como se muestra en la siguiente ecuación:

$$gaussHorizontal = [0,06136 \quad 0,24477 \quad 0,38774 \quad 0,24477 \quad 0,06136]$$

El kernel llamado `gaussVertical` si lo empleamos en una convolución va a producir el desenfoque de una imagen en la dirección vertical, siendo el valor de este nuevo kernel $(gaussHorizontal)^T$.

Algoritmo en Repa

El algoritmo en Repa con código 3.6 se ideó de la siguiente forma:

En primer lugar se realizó una función auxiliar que me permitiera desenfocar la imagen en el eje horizontal. Cuando se intentó aplicar la función `forStencil2` en la imagen haciendo uso del kernel mostrado en la ecuación 3.5.1, se produjeron fallos debido a que Repa no soporta la creación de stencils con números fraccionarios, por lo cual, se recurrió a la siguiente aproximación:

$$[1 \quad 4 \quad 6 \quad 4 \quad 1] \frac{1}{16}$$

Esto produjo que antes que nada, se realizara la función `forStencil2` con el kernel aproximado a números enteros, y posteriormente realizar una función `smap` (un map para matrices producidas por las funciones de stencils) donde dividiremos cada elemento de la matriz por 16. La función `computeP` ayuda a que en el momento de ejecución, la convolución como el map se realicen de manera paralela.

Posteriormente, realizaremos otra función auxiliar parecida a la anterior con la diferencia que el kernel que vamos a emplear es la transpuesta del kernel adaptado.

En la función principal, primero se realiza el desenfoque en dirección horizontal y posteriormente al resultado de aplicar el filtro se le aplica la otra función para aplicar el kernel en la dirección vertical.

```
blurX :: Channel Float -> IO (Channel Float)
blurX img = R.computeP
    $ R.smap (/ 16)
    $ forStencil2 BoundClamp img
        [stencil2| 1 4 6 4 1 |]
{-# NOINLINE blurX #-}

blurY :: Channel Float -> IO (Channel Float)
blurY img = R.computeP
    $ R.smap (/16)
    $ forStencil2 BoundClamp img
        [stencil2| 1
                    4
                    6
                    4
                    1 |]
{-# NOINLINE blurY #-}

blurV1 :: Channel Float -> IO (Channel Float)
blurV1 img = (blurX >=> blurY) img
{-# NOINLINE blurV1 #-}
```

Extracto de código 3.6: Algoritmo filtro Gaussiano empleando 2 kernels (Repa)

Si se observa la función principal del código 3.6 existe un operador ($>=>$). A este se le conoce como **operador pez derecho** y su utilidad es que si existe una función monádica ($a \rightarrow b$) como puede ser IO o Maybe, devuelve un valor a y otra función monádica ($b \rightarrow c$) acepta el valor de a como entrada, podemos componer esas dos funciones en una sola función monádica [39].

Un ejemplo de uso de este operador es que si nosotros tenemos una imagen img a la que se quiere aplicar la función monádica $f1$ y posteriormente, el resultado de esa aplicación queremos aplicarle la función monádica $f2$ sería:

```
(f1 >=> f2) img
```

Que puede ser expresada con la expresión **do** como:

```
do res <- f1 img
   f2 res
```

Algoritmo en Accelerate

El algoritmo realizado es sacado de los ejemplos de la documentación de Accelerate y lo que hace es primero realizar el desenfoque en la dirección horizontal y posteriormente, al resultado le aplicas el desenfoque en el eje vertical. En este caso, como Accelerate tiene una mayor flexibilidad a la hora de desarrollar stencils, hemos podido emplear kernels con números reales.

```
blur :: Acc (Matrix Float) -> Acc (Matrix Float)
blur = stencil (convolve5x1 gaussian) clamp
    . stencil (convolve1x5 gaussian) clamp
where gaussian = P.map A.constant
[0.06136, 0.24477, 0.38774, 0.24477, 0.06136]
```

Extracto de código 3.7: Algoritmo filtro Gaussiano empleando 2 kernels (Accelerate)

3.5.2. Filtro Gaussiano empleando 1 kernel

Algoritmo en Repa

Para la realización del algoritmo en Repa con código 3.8 en primer lugar se realizó las convoluciones de la imagen haciendo uso del segundo kernel de la figura 3.1 en la función `forStencil2`, y posteriormente lo dividimos por 273 a todos los píxeles de la imagen empleando la función `map` para conseguir el desenfoque que queremos. Recordando que para que ambas funciones funcionen de forma paralela, no olvidarnos de utilizar `computeP`.

```
blurV2 :: Channel Float -> IO (Channel Float)
blurV2 img = R.computeP
    $ R.smap (/273)
    $ forStencil2 BoundClamp img
[stencil2| 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 |]

{-# NOINLINE blurV2 #-}
```

Extracto de código 3.8: Algoritmo filtro Gaussiano empleando 1 kernel (Repa)

Algoritmo en Accelerate

El algoritmo en Accelerate con código 3.8 es muy similar al de Repa, descargando del uso de usar funciones como `computeP` para que se ejecuten los algoritmos de forma paralela. Esto es debido a que Accelerate se encarga cuando ejecutamos la aplicación en CPU o GPU.

```

gaussianSmoothing :: Acc (Matrix Float) -> Acc (Matrix Float)
gaussianSmoothing img =
  A.map (/273) $ stencil gaussian clamp img
  where gaussian :: Stencil5x5 Float -> Exp Float
    gaussian ((a1,a2,a3,a4,a5)
              ,(b1,b2,b3,b4,b5)
              ,(c1,c2,c3,c4,c5)
              ,(d1,d2,d3,d4,d5),
              (e1,e2,e3,e4,e5)) =
      a1+(4*a2)+(7*a3)+(4*a4)+a5
      + (4*b1)+(16*b2)+(26*b3)+(16*b4)+(4*b5)
      + (7*c1)+(26*c2)+(41*c3)+(26*c4)+(7*c5)
      + (4*d1)+(16*d2)+(26*d3)+(16*d4)+(4*d5)
      + e1+(4*e2)+(7*e3)+(4*e4)+e5

```

Extracto de código 3.9: Algoritmo filtro Gaussiano empleando 1 kernel (Accelerate)

3.6. 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[40].

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.

	1	1	1
1			
9	1	1	1

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

Filtro media en Repa

Debido a que en la librería Repa no permite escribir stencils con números racionales, se tuvo que realizar en primer lugar la función `forStencil2` de Repa empleando como kernel la matriz de unos mostrada en la figura 3.2, y después, se empleó una función `smap` de Repa (un map para matrices producidas por las funciones de stencils) para poder dividir todos los elementos resultantes de la matriz por 9. Y para que la función `forStencil2` y `smap` puedan realizarse en paralelo, no debemos olvidarnos añadir la función `computeP`.

```
meanF :: Channel Float -> IO (Channel Float)
meanF img = R.computeP
    $ R.smap (/9)
    $ forStencil2 BoundClamp img
        [stencil2| 1 1 1
                    1 1 1
                    1 1 1 |]
```

Extracto de código 3.10: Algoritmo Filtro Media (Repa)

Filtro media en Accelerate

A diferencia de Repa, los stencils en Accelerate son mucho más adaptables, por lo que mediante la realización de la función `stencil` de Accelerate, podemos resolver todo el algoritmo.

```
meanChannel :: Acc (Matrix Float) -> Acc (Matrix Float)
meanChannel img = stencil meanK clamp img
  where meanK :: Stencil3x3 Float -> Exp Float
        meanK ((a,b,c)
                ,(d,e,f)
                ,(g,h,i)) = a/9+b/9+c/9+d/9+e/9+f/9+g/9+h/9+i/9
```

Extracto de código 3.11: Algoritmo Filtro Media (Accelerate)

3.7. 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[41].

<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>-1</td><td>0</td><td>+1</td></tr> <tr><td>-2</td><td>0</td><td>+2</td></tr> <tr><td>-1</td><td>0</td><td>+1</td></tr> </table>	-1	0	+1	-2	0	+2	-1	0	+1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>+1</td><td>+2</td><td>+1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>-1</td><td>-2</td><td>-1</td></tr> </table>	+1	+2	+1	0	0	0	-1	-2	-1
-1	0	+1																	
-2	0	+2																	
-1	0	+1																	
+1	+2	+1																	
0	0	0																	
-1	-2	-1																	
Gx	Gy																		

Figura 3.3: *Kernels* Gradiente eje X (vertical) y Gradiente eje Y (horizontal)

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:

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

El desarrollo del algoritmo Sobel en ambos módulos se llevó a cabo teniendo en cuenta que la imagen de entrada va a ser la luminancia de una imagen RGB. Se ha decidido de esta manera porque el canal de luminancia tiene información de los bordes mucho más importante que las bandas R, G o B.

Algoritmo Sobel en Repa

El algoritmo Sobel en Repa con código 3.12 se ideó de la siguiente forma:

En primer lugar, se realizó una función auxiliar que nos permita calcular el gradiente en el eje vertical de la imagen de entrada. Esto se hizo realizando la función `forStencil2` aplicando el primer kernel de la figura 3.3. Al hacer uso de la función `computeP`, hará que las convoluciones se hagan de forma paralela sobre CPU.

Posteriormente se llevaría a cabo otra función auxiliar parecida a la anterior, pero calculando el gradiente en el eje horizontal, para ello, la función se desarrolló de la misma manera cambiando el kernel por el segundo de la figura 3.3.

Finalmente, el algoritmo se ejecutó en primer lugar la creación del gradiente en el eje horizontal y vertical de la imagen gracias a las funciones auxiliares anteriormente mencionadas. A partir de esas dos imágenes, se realizó el cálculo de la magnitud gracias a la función `zipwith` de Repa, que calcula los elementos a partir de una función y los elementos de entrada que aparecen en la misma posición en ambos arrays. El cálculo de la magnitud se pudo realizar de forma paralela gracias a `computeP`.

```

gradX :: Channel Float -> IO (Channel Float)
gradX img = Repa.computeP
    $ forStencil2 BoundClamp img
    [stencil2| -1 0 1
              -2 0 2
              -1 0 1 |]
{-# NOINLINE gradX #-}

gradY :: Channel Float -> IO (Channel Float)
gradY img = Repa.computeP
    $ forStencil2 BoundClamp img
    [stencil2| -1 -2 -1
              0 0 0
              1 2 1 |]
{-# NOINLINE gradY #-}

sobel :: Channel Float -> IO (Channel Float)
sobel img = do
    gx <- gradX img -- Gradiente de X
    gy <- gradY img -- Gradiente de Y
    R.computeP $ R.zipWith (\x y -> sqrt (x*x + y*y)) gx gy

```

Extracto de código 3.12: Algoritmo Sobel (Repa)

Algoritmo Sobel en Accelerate

El algoritmo en Accelerate con código 3.13 se realizó de una manera muy similar al algoritmo realizado en Repa, con la diferencia de que en este módulo, se tiene una mayor libertad a la hora de escribir los *stencils*, permitiéndonos omitir valores nulos del kernel.

Además de no tener que añadir una función como `computeP` en cada función para que nos paralelice cada algoritmo como en Repa, debido a que en Accelerate si lo ejecutamos con Accelerate Native o PTX, se realizarán las funciones paralelizables de la misma forma que en Repa.

```

gradX :: Acc (Matrix Float) -> Acc (Matrix Float)
gradX img = stencil gradient clamp img
  where gradient :: Stencil3x3 Float -> Exp Float
        gradient ((t1,_,t2)
                  ,(l,_,r)
                  ,(b1,_,b2)) = -t1-(2*l)-b1+t2+(2*r)+b2

```

```

gradY :: Acc (Matrix Float) -> Acc (Matrix Float)
gradY img = stencil gradient clamp img
  where gradient :: Stencil3x3 Float -> Exp Float
    gradient ((t1,t2,t3)
              ,(_,_,_)
              ,(b1,b2,b3)) = -t1-(2*t2)-t3+b1+(2*b2)+b3

sobel :: Acc (Matrix Float) -> Acc (Matrix Float)
sobel img = A.zipWith (\x y -> A.sqrt (x*x+y*y))
                     (gradX img)
                     (gradY img)

```

Extracto de código 3.13: Algoritmo Sobel (Accelerate)

3.8. 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[42].

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

El Laplaciano $L(x,y)$ de una imagen con valores de intensidad $I(x,y)$ viene dado por [43]:

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

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.

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

4-neighbourhoods
8-neighbourhoods

Figura 3.4: Dos aproximaciones al uso del filtro laplaciano

La implementación de ambos módulos se realizó utilizando el primer kernel de la figura 3.4 como stencil de nuestro algoritmo y realizando la convolución de forma paralela como hemos visto durante el desarrollo de estos algoritmos.

3.9. 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**[44], 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 3.14: 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 evalúa 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
```

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 GPU) 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*¹⁰ 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*,

¹⁰<https://raw.githubusercontent.com/AccelerateHS/accelerate-examples/a973ee423b5eadda6ef2e2504d2383f625e49821/lib/Data/Array/Accelerate/Examples/Internal/Backend.hs>

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 ¹¹.

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*.

¹¹<https://hackage.haskell.org/package/repa-3.4.1.5/docs/src/Data.Array.Repa.Base.html#deepSeqArray>

4. Análisis de rendimiento

A continuación se propone realizar un estudio sobre el rendimiento de los algoritmos desarrollados en el capítulo anterior. En la ejecución de estos algoritmos, se pretende analizar el rendimiento tanto cuando se ejecuta en CPU (Repa o Accelerate Native) como en GPU (Accelerate PTX).

Antes que nada, se explica las condiciones bajo las que se realiza las pruebas: características del sistema donde se ejecuta, que medida de tiempo se ha utilizado, entre otras cosas. Posteriormente, se adjunta una colección de capturas tras la aplicación de los algoritmos a las imágenes o vídeos, a fin de verificar como se visualizan los algoritmos realizados, así como asegurar que se comportan de manera adecuada.

Finalmente, se analiza la diferencia de rendimiento entre la ejecución de los distintos algoritmos realizados en Repa y Accelerate.

4.1. Detalles de experimentación

El entorno sobre el que se han obtenido los resultados tras haber realizado las pruebas de rendimiento es el siguiente:

- CPU : Intel Core i7-7700HQ, 3.80 GHz, 6MB cache, 4 núcleos
- GPU : NVIDIA GeForce GTX 1050, 2GB memoria, 640 núcleos CUDA
- RAM: 8 GB
- SO: POP!_OS 21.10

Se intentó recurrir al uso de los miniclústeres de GPU proporcionados por el grupo de investigación en Computación Natural de la Universidad de Sevilla para conseguir estudiar la escalabilidad de las soluciones en GPUs de mayor rendimiento. Sin embargo, pero debido a los diferentes conflictos que se desarrollaron al montar el proyecto en esos clusters, los cuales están basados en CentOS 7, 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.

4.2. Validación de los resultados

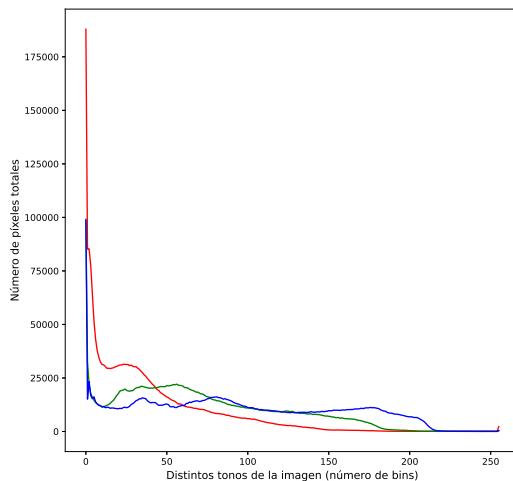
Todas las imágenes y vídeos de entrada van a ser en escala de colores RGB, aunque, para probar la eficiencia de algunos filtros, se le ha metido ruido a la imagen de entrada mediante el uso de `ImageJ`, un programa de procesamiento de imagen digital de dominio público escrito en Java[45].

4.2.1. Generación de Histogramas

Como podemos observar en la figura 4.1, el histograma nos revela que la imagen que hemos utilizado como ejemplo, tiene muchas zonas oscuras, debido a que la mayor concentración de píxeles de las 3 bandas, se encuentra en la zona de la izquierda, donde los tonos son muy sombríos.



(a) Imagen de entrada



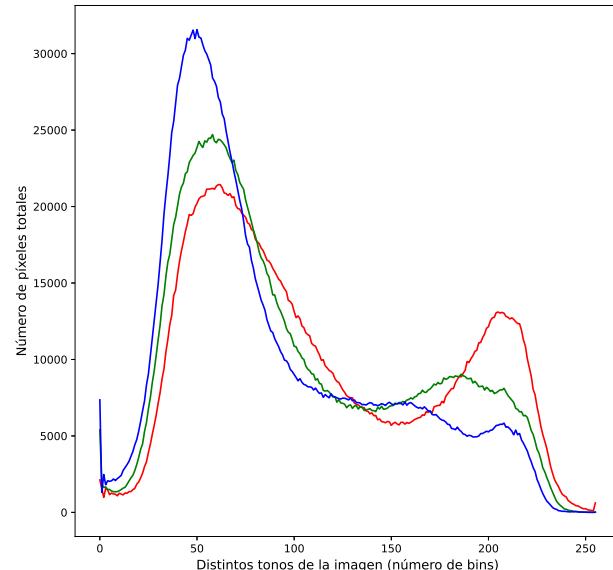
(b) Histograma RGB

Figura 4.1: Imagen de entrada y su histograma RGB (Accelerate). Este histograma es idéntico que el conseguido con una función `histograma` escrita en Haskell

Observando la figura 4.2, el histograma nos desvela que estamos ante una imagen bien expuesta, debido a que el histograma describe que la imagen tiene información tanto en las sombras como en altas luces.



(a) Imagen de entrada



(b) Histograma RGB

Figura 4.2: Imagen de entrada y su histograma RGB (Repa V3).Este histograma es idéntico que el conseguido con una función histograma escrita en Haskell

En conclusión, podemos sacar que el histograma se trata de una herramienta sencilla y útil para poder conseguir información que no es perceptible desde la imagen.

4.2.2. Escala de grises



Figura 4.3: Imagen de entrada y resultado transformación RGB a escala de grises (Accelerate). Esta transformación es idéntica que el conseguido con una función `grayScale` escrita en Haskell

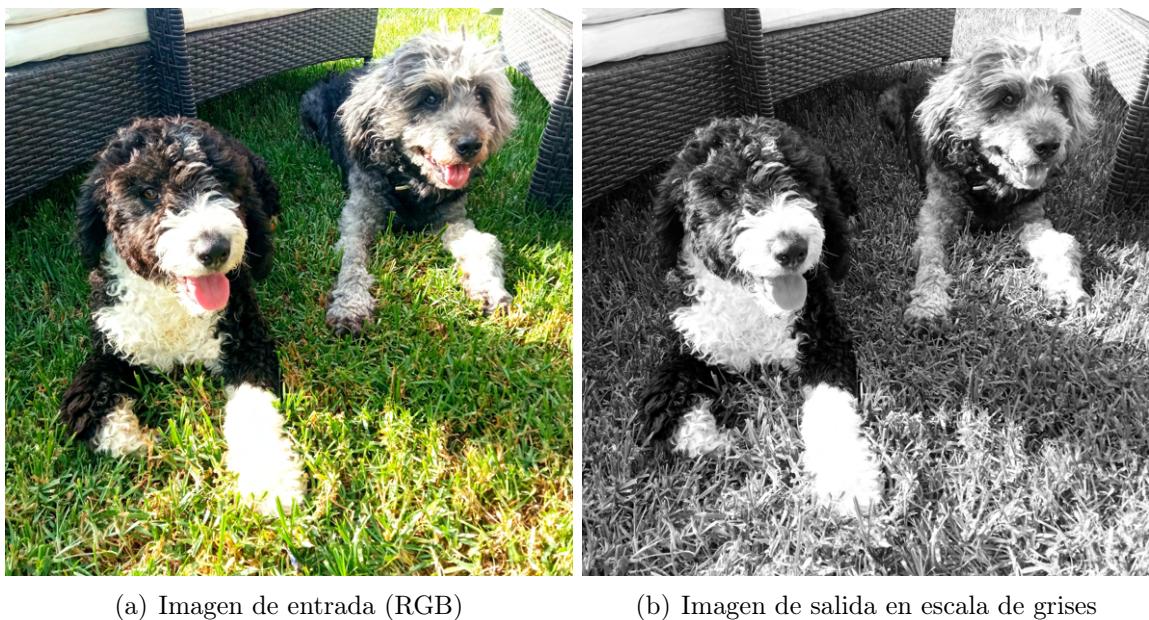


Figura 4.4: Imagen de entrada y resultado transformación RGB a escala de grises (Ripa V2). Esta transformación es idéntica que el conseguido con una función `grayScale` escrita en Haskell

La razón principal del desarrollo de este algoritmo, es que en vez de trabajar siempre con imágenes en color, podamos utilizar imágenes en escala de grises para simplificar ciertos algoritmos y así reducir los recursos computacionales. Esto es debido a que algunos algoritmos pueden no requerir tanta información, produciendo que tengamos información innecesaria, por lo que aumentaría la cantidad de trabajo que debemos realizar [46].

4.2.3. Filtro Gaussiano

Debido a que uno de los propósitos de los filtros es poder eliminar el ruido de las imágenes conservando el detalle de las mismas, se ha decidido añadirle un ruido gaussiano de $\sigma = 25$ mediante la librería ImageJ para comprobar que tal reduce este tipo de ruido, uno de los filtros más adecuados en estas circunstancias.

Además, se han utilizado 2 aplicaciones del filtro gaussiano en cada algoritmo.

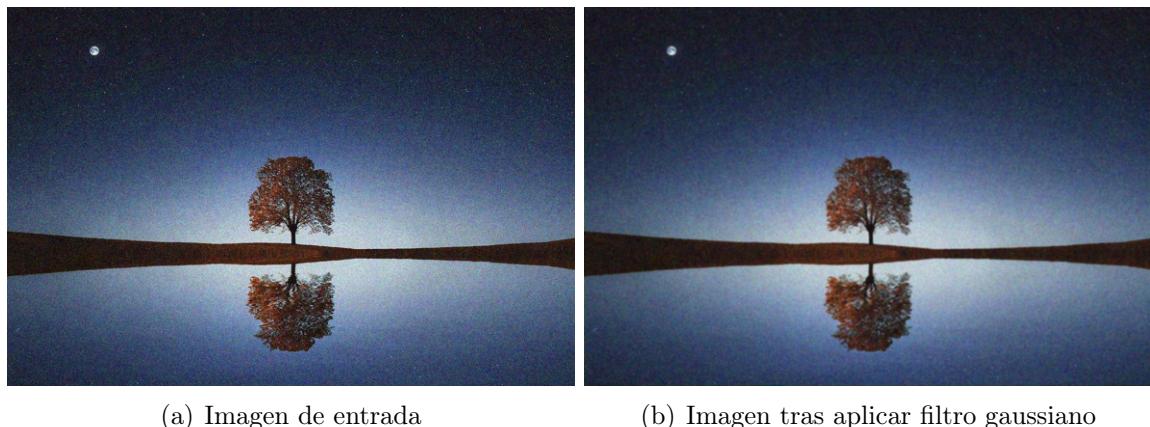


Figura 4.5: Imagen de entrada con ruido gaussiano $\sigma = 2$ y resultado tras aplicarle filtro gaussiano (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro gaussiano escrita en Haskell



Figura 4.6: Imagen de entrada con ruido gaussiano $\sigma = 2$ y resultado tras aplicarle filtro gaussiano (Repa). Esta transformación es idéntica que el conseguido con una función de filtro gaussiano escrita en Haskell

Podemos observar como en ambas figuras se podido reducir el ruido gaussiano gracias a aplicar este filtro. Cuantas más iteraciones realicemos, podremos eliminar más ruido con la pega de que también difuminamos la imagen.

4.2.4. Filtro media

El filtro media es uno de los más adecuados para la eliminación del ruido de disparo o *poisson noise*^[47], por lo que para el muestreo de estos resultados, se le ha añadido este tipo de ruido a las imágenes de prueba se ha recurrido al uso de un plugin de la librería ImageJ, debido a que por defecto no viene integrada esta adición.



(a) Imagen de entrada con ruido de disparo

(b) Imagen tras aplicar el filtro media

Figura 4.7: Imagen de entrada con ruido de disparo y resultado tras aplicarle el filtro media (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro media escrita en Haskell



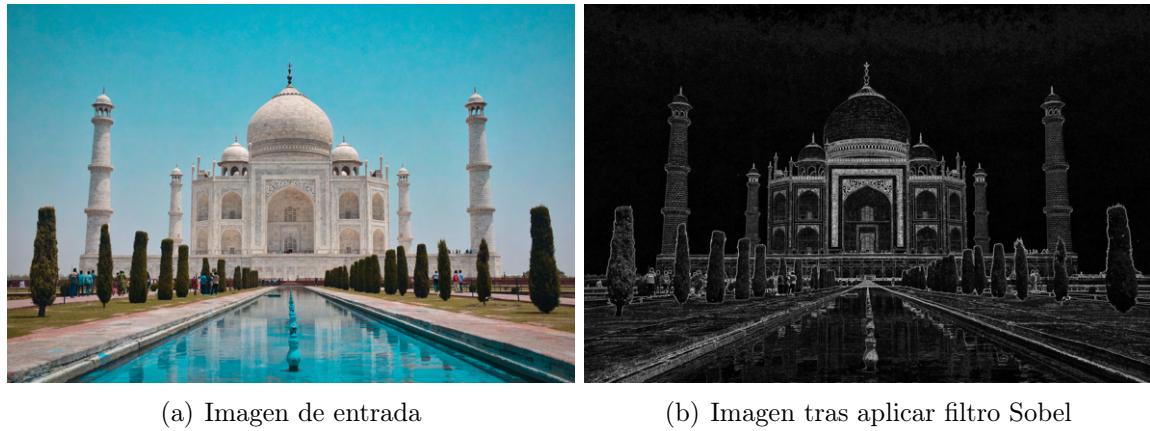
(a) Imagen de entrada con ruido de disparo

(b) Imagen tras aplicar el filtro media

Figura 4.8: Imagen de entrada con ruido de disparo y resultado tras aplicarle el filtro media (Repa). Esta transformación es idéntica que el conseguido con una función de filtro media escrita en Haskell

Examinando ambas imágenes, se puede observar como el ruido de disparo desaparece de la imagen.

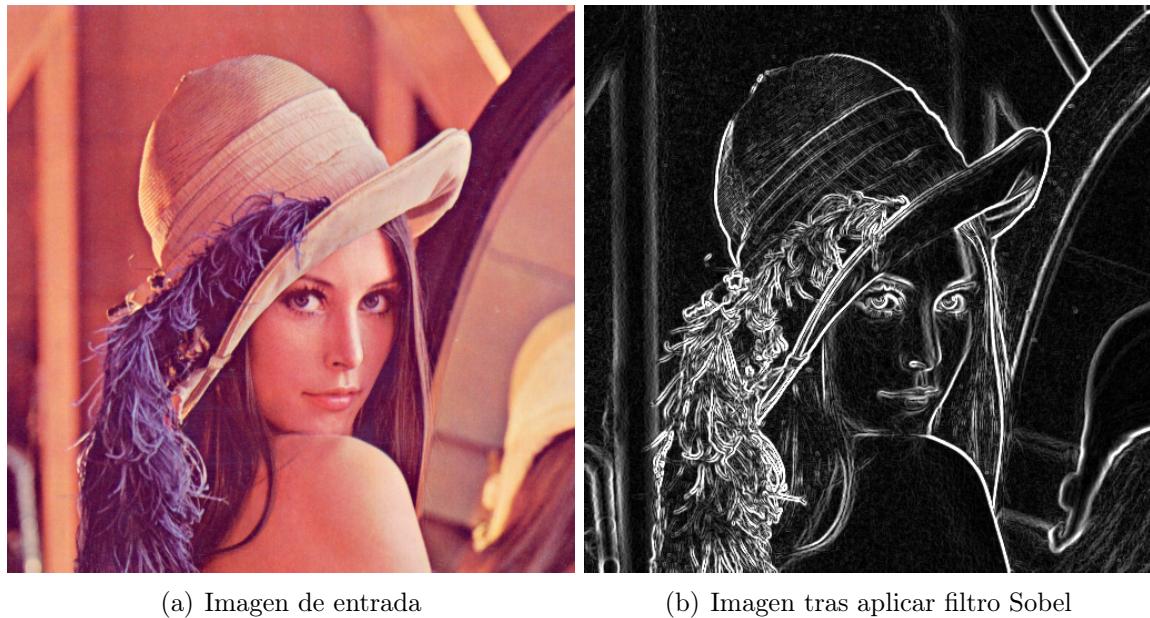
4.2.5. Sobel



(a) Imagen de entrada

(b) Imagen tras aplicar filtro Sobel

Figura 4.9: Imagen de entrada y resultado tras aplicarle el filtro Sobel (Accelerate). Esta transformación es idéntica que el conseguido con una función Sobel escrita en Haskell



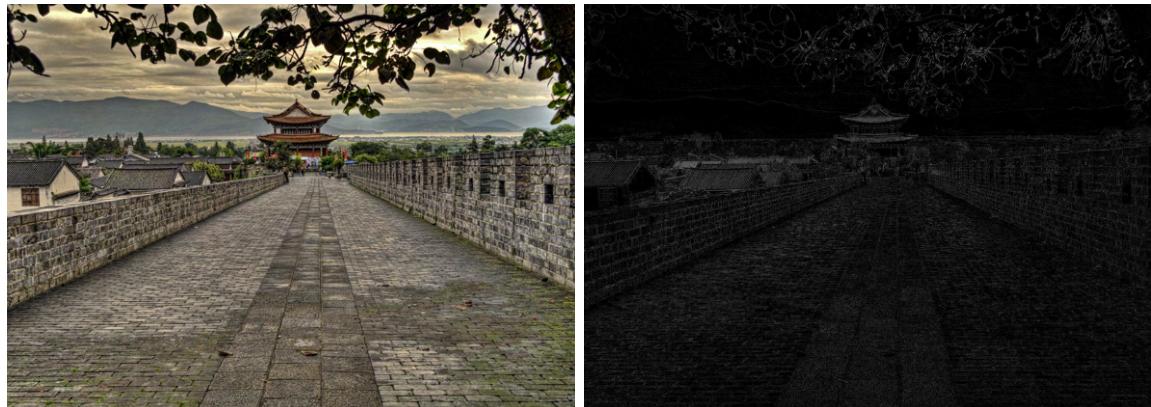
(a) Imagen de entrada

(b) Imagen tras aplicar filtro Sobel

Figura 4.10: Imagen de entrada y resultado tras aplicarle el filtro Sobel (Accelerate). Esta transformación es idéntica que el conseguido con una función Sobel escrita en Haskell

Observando las figuras anteriores, podemos concluir que gracias a este filtro, podemos generar un algoritmo de detección de bordes simple, que posteriormente nos puede ayudar a generar algoritmos de encontrar bordes de manera mucho más precisa, como puede ser Canny.

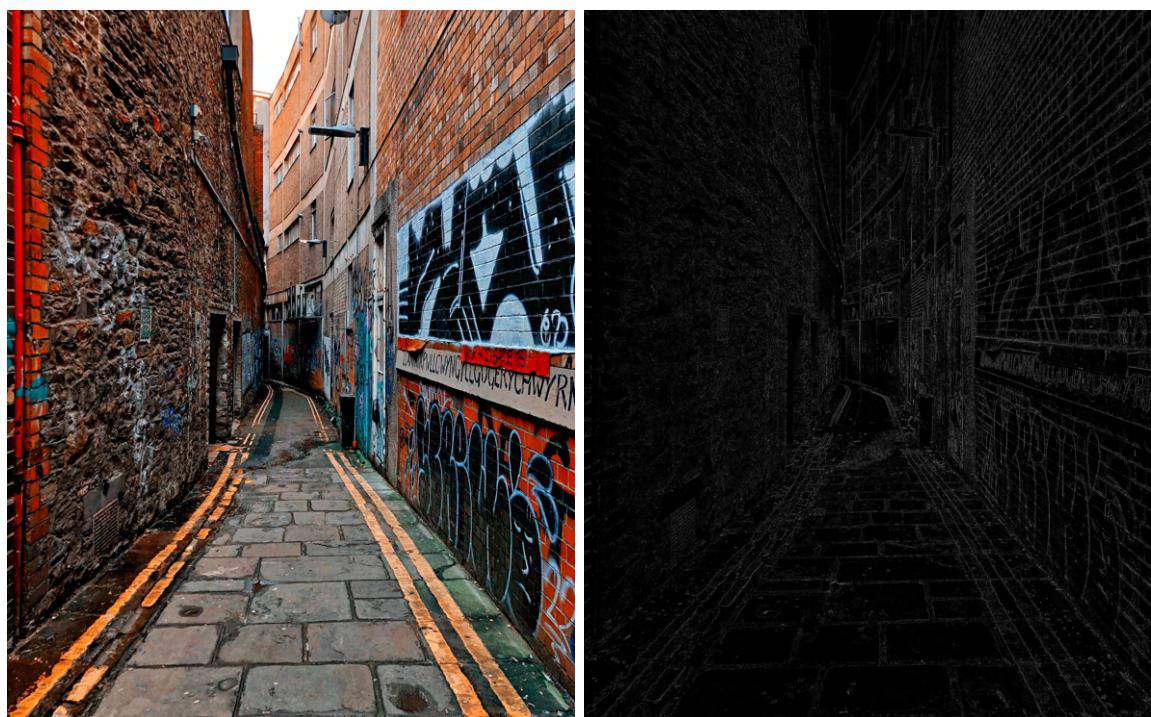
4.2.6. Filtro Laplaciano



(a) Imagen de entrada

(b) Realce de la imagen

Figura 4.11: Imagen de entrada y resultado tras aplicarle el filtro Laplace (Accelerate). Esta transformación es idéntica que el conseguido con una función de filtro Laplaciano escrita en Haskell



(a) Imagen de entrada

(b) Realce de la imagen

Figura 4.12: Imagen de entrada y resultado tras aplicarle el filtro Laplace (Repa). Esta transformación es idéntica que el conseguido con una función de filtro Laplaciano escrita en Haskell

El filtro laplaciano detecta bordes en la imagen, produciendo bordes finos de un píxel de ancho. Así que este filtro nos puede permitir realzar nuestras imágenes combinando (es decir, restando) la imagen filtrada y la imagen original.

4.2.7. Vídeos de prueba

Para finalizar la sección, se han probado algunos algoritmos anteriormente mencionados, pero ahora en vídeos, para probar la eficacia que tienen nuestros cálculos cuando se están trabajando con un gran número de imágenes.

Debido a los problemas que dan las imágenes que no son RGB8, además de los problemas que hubo con el hardware que se estaba usando, se decidió la lectura y escritura de vídeos de tamaño 640x360 con el algoritmo gaussiano en Accelerate y el filtro Media en Repa:

Repa

1. [Ejemplo 1](#)

Accelerate:

1. [Ejemplo 1](#)
2. [Ejemplo 2](#)
3. [Ejemplo 3](#)

4.3. Pruebas de rendimiento

En este apartado se estudia el rendimiento que tienen los diferentes algoritmos realizados en el capítulo anterior [3](#). Esto se hará mediante el uso de gráficas donde cada línea reflejará el algoritmo realizado respecto al tiempo medio de ejecución como al tamaño de las diferentes imágenes o tablas de tiempo para poder observar el tiempo entre los diferentes algoritmos.

Cabe señalar que el algoritmo escrito en Accelerate puede ejecutarse por su intérprete, paralelizando mediante CPU o sobre GPU. De manera que un mismo código puede elaborar 3 versiones distintas, de los cuales pondremos a prueba su rendimiento.

En las gráficas el tiempo de ejecución medio se ha decidido utilizar como medida estándar **ms** con escala logarítmica.

4.3.1. Generación de Histogramas

En la figura 4.13 se observa que las versiones realizadas en Repa obtienen peores resultados que las ejecuciones de CPU y GPU en Accelerate. Esto es debido a que cuando programamos las distintas versiones de Repa, tuvimos que hacer uso de otras estructuras de datos como acumulador de los histogramas, puesto que hacerlo todo en Repa provocaba ejecuciones interminables.

Entre las diferentes ejecuciones en Accelerate, la GPU es la que está a la cabeza de todas, seguida de la CPU y en la cola podemos ver el intérprete, que durante todos los análisis de rendimiento, ya adelantamos que es de los más lentos, esto es debido a que la función de este *backend* no es el de obtener buenos rendimientos, si no el de definir la semántica de forma adecuada.

Dentro de las versiones producidas en Repa, la más veloz es la que paralelizamos mediante CPU la generación del histograma por cada banda, y dentro de cada banda paralelizamos por cada fila la creación de un histograma para finalmente fusionar todos los histogramas mediante una operación suma (Repa V3).

Aunque lo más sorprendente es que la versión secuencial de la generación de histogramas (Repa V1) sea solamente un poco más lenta que la versión donde se realiza la realización del histograma de forma secuencial, pero paralelizando las 3 bandas (Repa V2).

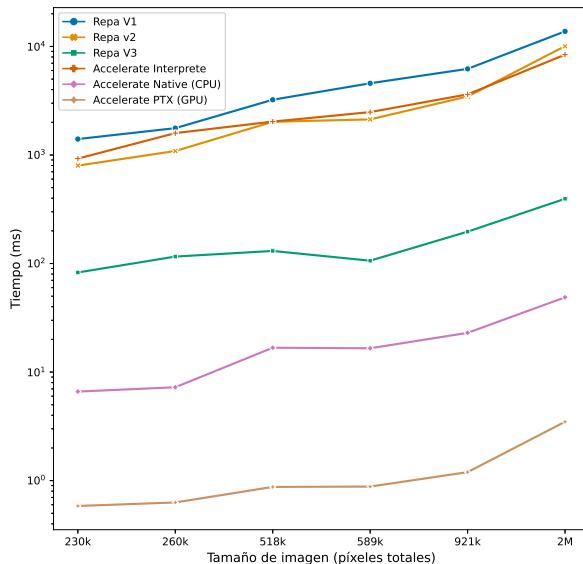


Figura 4.13: Tiempos de ejecución para las diferentes versiones de generar histogramas. Tenga en cuenta la escala logarítmica en el eje de coordenadas.

4.3.2. Escala de grises

Viendo la figura 4.14, se puede observar que la versión más rápida de todas sigue siendo Accelerate con GPU, seguida de Accelerate paralelizando CPU.

De las 2 versiones que se realizaron en la librería Repa, la segunda versión es mucho más rápida debido a que al fusionar bandas, reducimos el número de operaciones a realizar, y el coste de esta fusión de bandas en una terna por cada píxel es de $\mathcal{O}(1)$, a diferencia de la primera versión, en el que tenemos que realizar la suma de dos bandas y el resultado de esa suma, sumar la tercera banda aplicando los pesos para que quede la imagen en escala de grises.

Lo más destacable de esta gráfica es como Accelerate Native obtiene un mayor rendimiento que Repa, una librería que está enormemente centrada en la paralelización sobre CPU, mientras que Accelerate está enormemente diseñado para la paralelización sobre GPU.

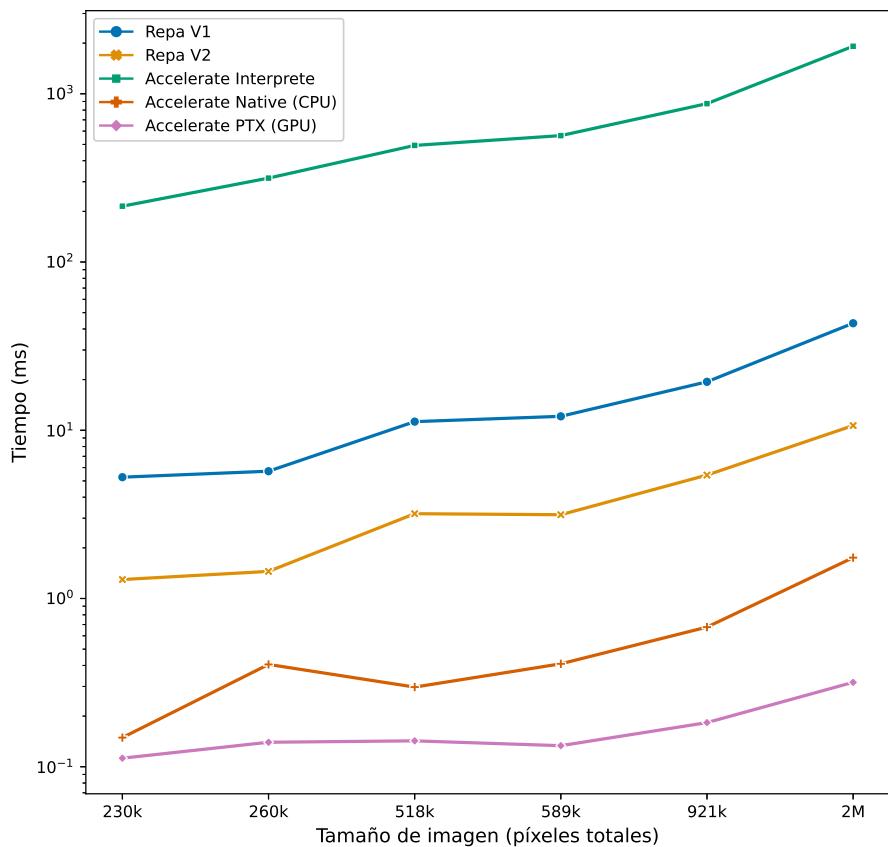


Figura 4.14: Tiempos de ejecución para las diferentes versiones de convertir una imagen RGB a escala de grises. Tenga en cuenta la escala logarítmica en el eje de coordenadas.

4.3.3. Filtro Gaussiano

Para el análisis de rendimiento del filtro gaussiano, se ha decidido evaluar el rendimiento de los algoritmos utilizando un kernel gaussiano de dimensión 5x5, posteriormente, lo compararemos con la propiedad de separar ese kernel en 2 (1x5 y 5x1).

En la primera tabla 4.1 podemos observar que Accelerate por GPU sigue siendo muy veloz a comparación del resto de versiones. Mientras tanto, el intérprete de Accelerate sigue teniendo unos resultados muy malos.

Siendo Repa V2 y Accelerate mediante CPU algoritmos que paralelizan CPU, hay que destacar que aunque en imágenes con pocos píxeles tienen cierta similitud de rendimiento teniendo ventaja Accelerate, sin embargo, cuando se va aumentando el número de píxeles la diferencia de tiempo entre ambos se va duplicando.

	230k	260k	518k	589k	921k	2M
Repa V2	12.19	11.97	24.65	25.28	37.93	77.98
Accelerate Interprete	9242	10210	20690	25140	39870	81740
Accelerate Native (CPU)	9.016	9.74	16.54	19.05	22.5	43.17
Accelerate PTX (GPU)	1.332	1.306	1.61	1.67	2.3	4.784

Cuadro 4.1: Tabla de rendimiento en ms - Filtro Gauss (1 kernel 5x5)

Observando la tabla 4.2 podemos observar que se repite el patrón del rendimiento de las distintas versiones, con la peculiaridad de que ahora los tiempos son mucho más rápidos con respecto a al tema de usar 1 solo kernel.

Esto es debido a que al separar ese kernel en 2, producimos que no se tengan que hacer operaciones tan grandes, provocando que tengamos muchos menos operaciones que realizar.

	230k	260k	518k	589k	921k	2M
Repa V1	8.703	9.908	16.34	17.46	26.84	52.68
Accelerate Interprete	4213	4613	9738	10870	17060	38440
Accelerate Native (CPU)	6.237	6.159	8.534	9.605	13.7	26.57
Accelerate PTX (GPU)	1.214	1.457	1.298	1.622	1.622	2.617

Cuadro 4.2: Tabla de rendimiento en ms - Filtro Gauss (2 Kernels 1x5 y 5x1)

4.3.4. Filtro media

Observando la figura 4.15, Accelerate PTX tiene el mayor rendimiento en aplicar un filtro media a una imagen.

Sin embargo, cuando hablamos de paralelización mediante CPU, Repa como Accelerate Native tienen un rendimiento muy similar en el cálculo de este algoritmo.

Finalmente, el intérprete de Accelerate sigue teniendo los peores rendimientos en estas pruebas.

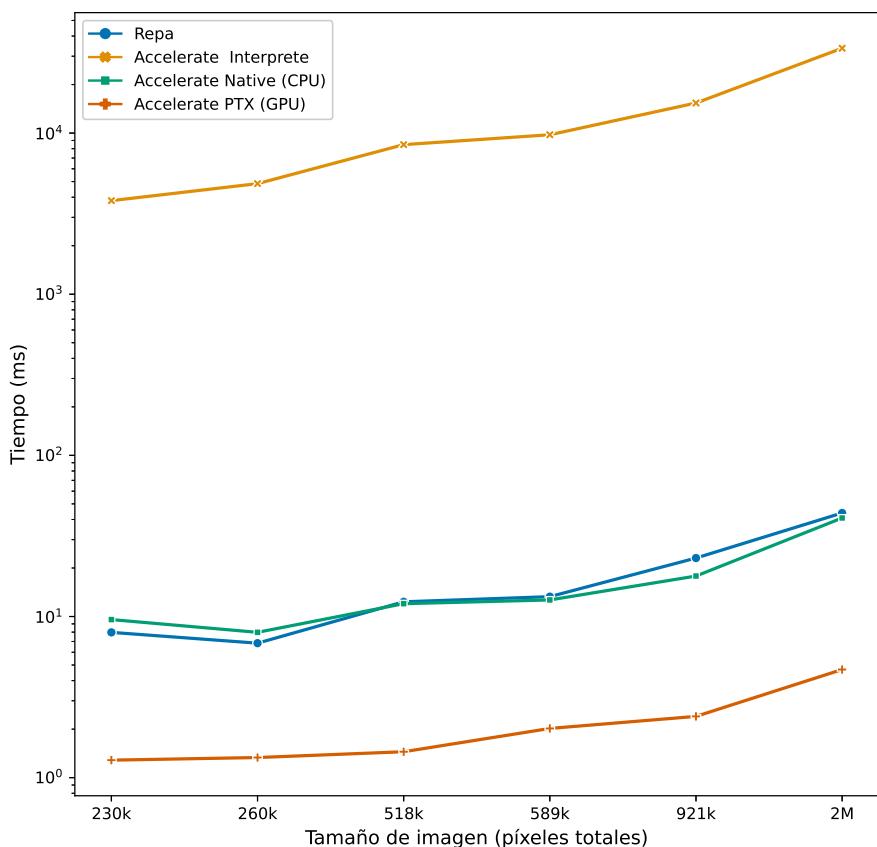


Figura 4.15: Tiempos de ejecución para las diferentes versiones de realizar el filtro media. Tenga en cuenta la escala logarítmica en el eje de coordenadas.

4.3.5. Sobel

Examinando la figura 4.16, Accelerate PTX sigue siendo la versión más rápida en ejecutar el algoritmo, mientras que Accelerate usando su intérprete solamente consigue resultados muy lentos a comparación del resto.

Accelerate Native sigue teniendo un mejor rendimiento que Repa, sin embargo, la diferencia entre ambas versiones no es tan grande.

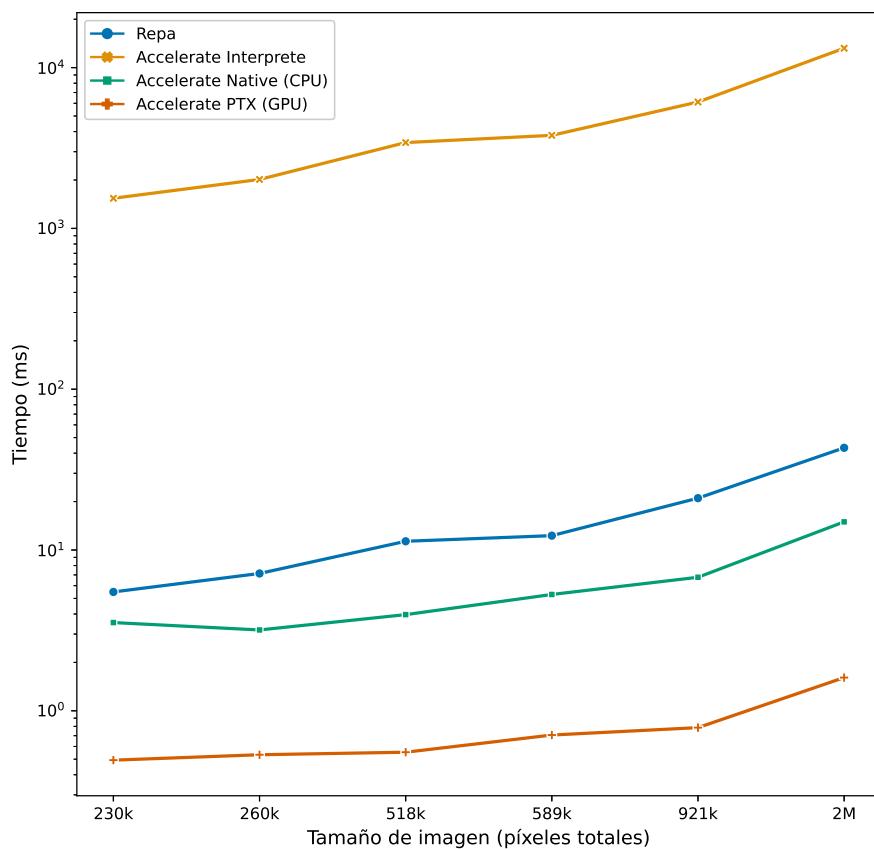


Figura 4.16: Tiempos de ejecución para las diferentes versiones de realizar el algoritmo Sobel. Tenga en cuenta la escala logarítmica.

4.3.6. Filtro Laplaciano

Viendo la 4.17, podemos finalizar esta parte de análisis de rendimiento que el ganador en estas pruebas de rendimiento que hemos realizado es Accelerate con su paralelismo GPU.

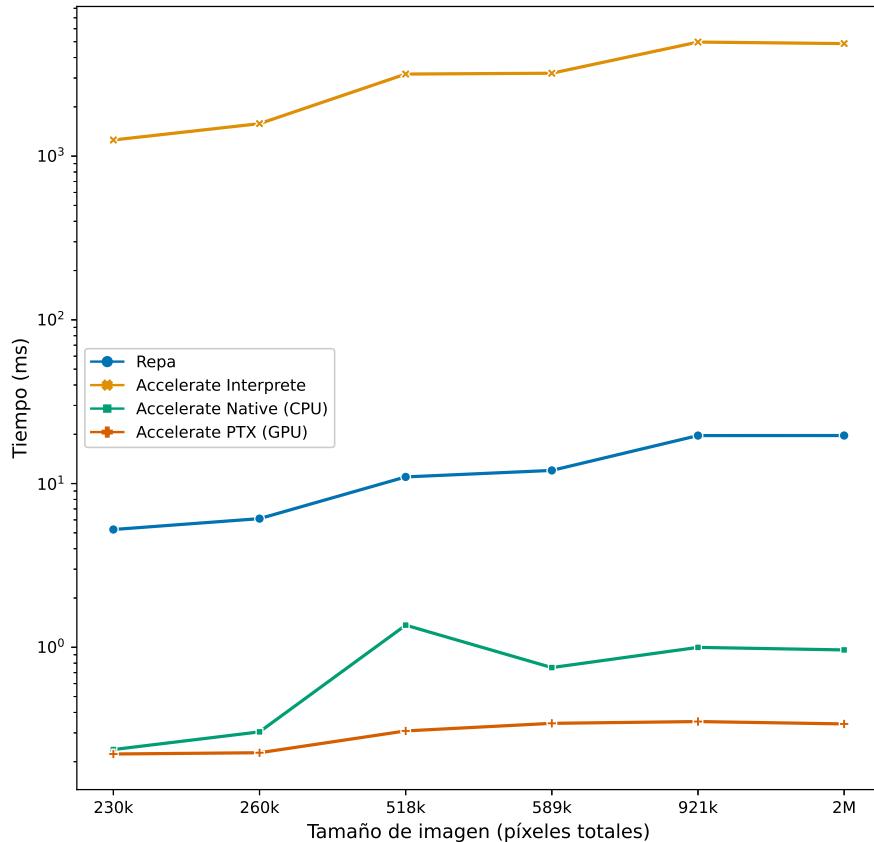


Figura 4.17: Tiempos de ejecución para las diferentes versiones de realizar el algoritmo Laplace. Tenga en cuenta la escala logarítmica en el eje de coordenadas.

4.3.7. Tabla resumen

Esta sección expone una tabla resumen con las aceleraciones máximas conseguidas en cada algoritmo en los diferentes módulos.

Para la creación de las tablas se ha tenido en cuenta la versión del algoritmo en Repa que sea más eficiente y en Accelerate solamente se va a mostrar los algoritmos acelerados sobre CPU (Accelerate Native) o GPU (Accelerate PTX). Y como datos de entrada para probar los algoritmos solamente se han tenido en cuenta la imagen más pequeña y la más grande que hemos testeado en las pruebas anteriores.

Repa vs Accelerate Native

	Repa Tamaño de entrada		Accelerate Native Tamaño de entrada	
Algoritmos	230k	2M	230k	2M
Histograma	76.65	390.1	1.486	44.93
Grayscale	1.253	10.4	0.1394	1.71
Gaussiano (2 kernels)	8.513	50.89	6.109	24.89
Gaussiano (1 kernel)	11.93	75.59	8.901	40.9
Media	7.602	41.46	8.88	34.97
Sobel	5.288	42.43	3.484	13.46
Laplace	5.079	19.32	0.2276	0.8826

Cuadro 4.3: Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Repa y Accelerate Native

Repa vs Accelerate Native

	Repa Tamaño de entrada		Accelerate PTX Tamaño de entrada	
Algoritmos	230k	2M	230k	2M
Histograma	76.65	390.1	0.5705	2.366
Grayscale	1.253	10.4	0.1106	0.3146
Gaussiano (2 kernels)	8.513	50.89	1.2	2.576
Gaussiano (1 kernel)	11.93	75.59	1.304	4.749
Media	7.602	41.46	1.266	4.657
Sobel	5.288	42.43	0.4831	1.6
Laplace	5.079	19.32	0.2194	0.3326

Cuadro 4.4: Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Repa y Accelerate PTX

Accelerate Native vs PTX

	Accelerate Native Tamaño de entrada	Accelerate PTX Tamaño de entrada	
Algoritmos	230k	2M	230k
Histograma	1.486	44.93	0.5705
Grayscale	0.1394	1.71	0.1106
Gaussiano (2 kernels)	6.109	24.89	1.2
Gaussiano (1 kernel)	8.901	40.9	1.304
Media	8.88	34.97	1.266
Sobel	3.484	13.46	0.4831
Laplace	0.2276	0.8826	0.2194
			0.3326

Cuadro 4.5: Tabla de aceleración máxima en los diferentes algoritmos realizados (ms) en Accelerate Native y PTX

4.3.8. Rendimiento en videos

Para finalizar el capítulo, se van a mostrar el rendimiento que tienen los vídeos mostrados en la sección anterior.

Para la medición del tiempo de los vídeos, se ha utilizado la función :set +s en las opciones del ghci.

Repa

1. **Ejemplo 1:** 3266.85 segundos.

Accelerate:

1. **Ejemplo 1:** 241.23 segundos.
2. **Ejemplo 2:** 436.51 segundos.
3. **Ejemplo 3:** 638.97 segundos.

Se intentaron probar con más ejemplos en Repa, pero tardaban mucho en ejecutar los algoritmos, por lo que por falta de tiempo, se descartó la idea de seguir intentando ejecutar más algoritmos. Aún así, la conclusión que podemos dar es que Accelerate tiene un mejor rendimiento que Repa.

5. Conclusiones y trabajo futuro

A continuación, se concluye el reporte con algunas conclusiones y debates de posibles líneas de trabajo futuro.

5.1. Conclusiones

En el presente trabajo se ha explicado el concepto del paralelismo y computación paralela, donde se ha tratado sus definiciones, el uso que tienen las GPU y CPU, programación GPU mediante CUDA y la relación que tiene la programación paralela con el paradigma funcional. Se ha expuesto la base teórica del lenguaje de programación funcional que hemos utilizado para desarrollar todos los algoritmos como también la explicación básica de como funcionan las dos librerías que se han utilizado para realizar el análisis de rendimiento. Este análisis se ha basado en el desarrollo de un conjunto de algoritmos de procesamiento de imágenes.

Además de esto, se ha descrito como hemos implementado el concepto de poder leer distintos tipos de formato de imágenes en Repa y Accelerate, debido a que ambos paquetes solamente aceptan archivos en formato BMP. Además, también se realizaron funciones de conversión de nuestros elementos de las matrices para poder trabajar de manera eficiente en las librerías anteriormente mencionadas, como en JuicyPixels, el módulo que nos permite leer y escribir imágenes.

Posteriormente se ha descrito como hemos implementado los distintos algoritmos de procesamiento de imágenes, así como el marco teórico bajo el que operan. También se ha explicado cómo se han podido recolectar el rendimiento de ejecución de las distintas versiones realizadas para poder crear tablas y generar gráficas donde se pueda visualizar el rendimiento que tienen con respecto a la cantidad de píxeles en una imagen.

Finalmente gracias a estas pruebas de rendimiento, se ha demostrado la utilidad que tiene el uso de la aceleración mediante GPU de Accelerate para conseguir un mayor rendimiento en aplicaciones paralelizables, como es el caso de los algoritmos de procesamiento de imágenes expuestos. Otra cosa que se ha demostrado que, aunque Accelerate es una librería especializada en paralelismo GPU, los rendimientos que ha logrado sobre CPU a comparación de Repa, que es una librería especializada en paralelismo sobre CPU, son muy buenas, por lo que si se sigue trabajando en el desarrollo de esta librería puede que en unos años nos encontremos ante el nuevo estándar en Haskell.

Además de lo expuesto, Accelerate tiene la ventaja de que a diferencia de tecnologías como OpenCL, en este módulo existe una gran facilidad para ejecutar el mismo algoritmo tanto en CPU y GPU, debido a que no tenemos que cambiar nuestro algoritmo para paralelizar en estos *backends*. A diferencia de Repa, Accelerate cuenta un mayor nivel de abstracción en la hora de utilizar el lenguaje, ya que no tenemos que especificar en el array si estamos utilizando matrices retardadas o matrices de manifiesto.

No obstante, Accelerate también tiene sus desventajas como que necesitamos obligatoriamente una gráfica NVIDIA compatible con la tecnología CUDA para poder ejecutar esos algoritmos sobre GPU. O también que Accelerate al ser un lenguaje eDSL, requerimos el uso de tecnologías auxiliares (como LLVM) para poder compilar ese código, haciendo que esos algoritmos sean menos eficientes que trabajar CUDA C por ejemplo, o también provocar problemas a la hora de instalar esos paquetes en algunos sistemas operativos.

Aún Accelerate consiguiendo un buen rendimiento, Repa es un módulo para comenzar con el mundo del paralelismo en lenguajes funcionales, esto es debido a que es un módulo con una sintaxis muy entendible, con un gran número de soluciones ya desarrolladas por otras personas y ha sido el origen para que existan librerías de procesamiento de imagen que paralelicen CPU.

Repa contiene una gran capacidad expresiva en el tema de desarrollar convoluciones, por lo que no me costó mucho entender los conceptos, y al tener una buena sintaxis, el cambio de escribir código Accelerate vieniendo de Repa no se te hace tan complejo.

Una característica a tener en cuenta si queremos desarrollar algoritmos en ambas librerías, es que tanto Accelerate como Repa no permiten el *nested parallelism*, pudiendo producir que no se puedan realizar algunos algoritmos o tener que adaptarlos de maneras mucho más complejas.

En conclusión, aunque Accelerate es una excelente opción para conseguir el máximo rendimiento en aplicaciones paralelizables, si eres nuevo en el concepto de paralelización, es mejor que empieces aprendiendo Repa para posteriormente aprender Accelerate de primeras, esto es debido a que Repa tiene una documentación mucho más elaborada que Accelerate, ya que ésta última librería lo realizó en su mayor medida Trevor L. McDonell para su tesis doctoral, además de que Accelerate tiene conceptos más complejos como Exp o Acc. Aunque si quieras trabajar con vídeos muy grandes, mejor descartes la idea de usar Repa.

5.2. Trabajo futuro

Basado en los resultados de este trabajo, existe una amplia gama de trabajos futuros muy interesantes. Los capítulos anteriores y el análisis de los resultados de las pruebas de rendimiento ya han proporcionado sugerencias para el trabajo futuro, pero aquí discutiremos las más importantes.

5.2.1. Optimización de programas Repa

En este trabajo se ha demostrado que Repa proporciona un marco de programación conveniente para describir operaciones matriciales y tiene unos beneficios como que se eliminan las matrices intermedias cuando hacemos uso de la fusión de arrays y que podemos paralelizar una función automáticamente empleando los núcleos disponibles haciendo uso de operaciones como `computeP`.

Además de estos beneficios, existen una serie de buenas prácticas en Repa para escribir código mucho más eficiente ¹:

1. Añada la instrucción para el compilador `INLINE` ² en todas las funciones que calculen valores de la matriz. Por ejemplo :

```
{-# INLINE f #-}
f x = x + 1

arr' = R.force
      $ R.zipWith (*) (R.map f arr1) (R.map f arr2)
```

Las llamadas a funciones perezosas que no sean `INLINE` pueden costar más de 50 ciclos cada una, mientras que cada operador numérico solo cuesta un ciclo (o menos), por lo que las funciones que tienen este pragma nos aseguran que estén especializadas en los tipos numéricos apropiados.

2. Agregue *Bang patterns* ³ a todos los argumentos de una función y en todos los campos de tus tipos de datos. No desea confiar en el analizador de rigor (strictness analyser) código numérico debido a que si no devuelve un resultado perfecto, el rendimiento del programa será pésimo.
3. Compile los módulos que usan Repa con las siguientes banderas:

```
-Odph -rtsopts -threaded -fno-liberate-case
-funfolding-use-threshold1000
-funfolding-keeness-factor1000
-fllvm -optlo-O3
```

Si no tiene instalado LLVM en el ordenador, la opción `-fllvm` no funcionará, el programa puede funcionar sin él, pero el código irá más lento.

4. Repa escribe en el registro de eventos de GHC al principio y al final de cada cálculo paralelo, haga uso de `threadscope` para ver lo que está haciendo su programa.
5. Cuando esté seguro de que su programa funciona correctamente, cambie a las versiones inseguras de las funciones como por ejemplo `traverse`, este tipo de funciones inseguras no hacen comprobaciones de límites.

En nuestro proyecto se han aplicado algunas de estas buenas prácticas, pero como trabajo futuro se puede intentar seguir todos estos consejos para generar un código mucho más eficiente, y poder volver a llevar a cabo las pruebas de rendimiento para ver como ha mejorado Repa.

¹<https://hackage.haskell.org/package/repa-3.4.1.5/docs/Data-Array-Repa.html>

²https://wiki.haskell.org/Inlining_and_Specialisation

³https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/bang-patterns.html

5.2.2. Uso de otros lenguajes de programación

El desarrollo del proyecto se ha realizado con Repa y Accelerate, dos módulos que pertenecen al lenguaje de programación Haskell.

Por lo que como trabajo futuro sería interesante probar el rendimiento de los mismos algoritmos que hemos experimentado, pero en otros lenguajes funcionales que soporten paralelismo sobre GPU como puede ser Scala con la librería [Firepile](#)⁴, o SPOC⁵, una librería para programar sobre GPU en el lenguaje OCaml.

Entre los lenguajes funcionales que soportan aceleración mediante GPU, Futhark destaca por su simpleza y por lo fácil que es de aprender. Este pequeño lenguaje no pretende remplazar los lenguajes existentes, si no que el objetivo es emplearlo en partes de una aplicación relativamente pequeña de otro lenguaje de programación que necesite un uso intensivo de cómputo.

Por lo que otro trabajo a futuro sería realizar los algoritmos realizados en los módulos que se han hecho en este proyecto en el lenguaje Futhark, y compararlos con los resultados obtenidos en Accelerate y Repa.

No obstante, también podemos realizar los algoritmos en otros lenguajes fuera del paradigma funcional como pueden ser CUDA C, Python con la librería Numba, entre otros. Esto nos permitiría probar el rendimiento que existe entre lenguajes funcionales y lenguajes de otro paradigma que soporten paralelismo sobre GPU.

5.2.3. Pruebas de rendimiento en distinto hardware

Durante la realización de las pruebas de rendimiento, además de hacerse con el hardware del autor, se intentó recurrir al uso de miniclusters de GPU proporcionados por el RGNC de la Universidad de Sevilla, sin embargo, por problemas con la instalación de los paquetes necesarios y falta de tiempo, se tuvo que descartar la idea.

Por tanto, un trabajo a futuro puede ser el realizar las mismas pruebas de rendimiento de los distintos algoritmos que se han desarrollado en este trabajo en tarjetas gráficas y CPUs mucho más potentes, para verificar que se sigan los patrones de rendimiento que hemos mostrado en la capítulo de análisis de rendimiento [4](#).

5.3. Consejos y buenas prácticas

Tras haber pasado mucho tiempo investigando y programando sobre diferentes módulos de Haskell, en esta sección se va a desarrollar una recapitulación de consejos y buenas prácticas que el autor ha ido recolectando mientras hacía uso de esos módulos.

⁴<https://www.inf.usi.ch/faculty/nystrom/papers/firepile.pdf>

⁵<https://mathiasbourgoin.github.io/SPOC/>

5.3.1. Accelerate

Accelerate es el módulo que nos proporciona el realizar operaciones matriciales sobre CPU y GPU. Por lo que en este caso, se va a tratar una serie de ideas que pueden resultar útiles al lector cuando empiece a hacer uso de Accelerate:

- Cuando estés trabajando con k-tuplas donde cada elemento es una matriz que está integrada en el lenguaje Accelerate, si quieras que cada elemento se lleve a cabo en paralelo en diferentes kernels y no se hagan en uno solo, debes añadir la función `compute` a cada elemento de la k-tupla.
- Solo hacer uso de la función `run` del intérprete de Accelerate cuando queramos ver si la semántica del lenguaje es correcto, para realizar cálculos acelerados hacer uso de Accelerate Native o PTX.
- Una cosa a tener en cuenta es que Accelerate no permite el **parallelismo anidado**, esto quiere decir que no podemos desarrollar array de arrays, así que tenerlo en cuenta a la hora de querer desarrollar vuestros algoritmos.
- El uso de las funciones `lift` y `unlift` es uno de los tipos de errores más comunes cuando hacemos uso de Accelerate porque GHC no es bueno muchas veces determinando las expresiones [un]lift, por lo que hay que añadir los datos de forma explícita.

5.3.2. Repa

Repa es el módulo que nos proporciona el hacer operaciones sobre matrices paralelizando sobre CPU. Aunque anteriormente ya se ha visto algunas buenas prácticas sobre como escribir código con mayor rendimiento, en este caso, se va a hablar una serie de conceptos que pueden ser útiles para el lector cuando empiece a programar en Repa:

- Si estamos trabajando con funciones que van a devolver una función monádica, como puede ser el haber empleado `computeP`, lo mejor es emplear el operador `(>=>)`, esto va a permitir obtener un código mucho más legible con el mismo resultado, además de ahorrarnos líneas de código.
- Los stencils en Repa solamente aceptan kernels hasta 7×7 , aunque es un tamaño muy grande, existen kernels de un mayor tamaño que no se podrían realizar en esta librería, como puede ser el Laplaciano del Gaussiano (LoG)⁶, que tiene unas dimensiones de 9×9 .
- Cuando creamos stencils en Repa también tenemos que tener en cuenta que solamente acepta números enteros, por lo tanto, si queremos trabajar con números fraccionarios, lo mejor que podemos hacer es intentar aproximar esos números a enteros, como tuvimos que hacer en el filtro gaussiano con la propiedad separable.
- Hasta donde llegan mis conocimientos de la librería, Repa no tiene una referencia a un área de memoria que se pueda modificar sin copiar toda la matriz. Por lo que, incluso si queremos realizar una operación en la que se modifique un píxel, vamos a

⁶<https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm>

tener que provocar una copia completa incluso si estamos con matrices retrasadas. Por lo que si nos encontramos en la situación de querer realizar modificaciones a un array en Repa (como por ejemplo, crear un histograma), si es en una posición donde vamos a llevar a cabo muchos cálculos similares de este estilo, lo mejor es cambiar Repa por una librería de arrays mutables, o si es un caso puntual, pues nos podemos apañar haciendo uso de otras librerías como acumuladores (como Sequence o Vector).

5.3.3. JuicyPixels

Este módulo es el más simple para comenzar en el campo del procesamiento de imágenes digitales, además de soportar la mayoría de formatos como pueden ser PNG y JPEG, y cuenta con muchos *wrappers* hacia distintos módulos que se encargan de realizar cálculos sobre matrices o librerías especializadas en el procesamiento de imágenes.

Una de las características más importantes del uso de JuicyPixels, es que al estar escrito totalmente en Haskell, no se necesita instalar ninguna herramienta externa.

Si quiere realizar muchas ejecuciones de entrada y salida en Repa o Accelerate en formatos populares como PNG y JPEG, va a tener que necesitar la ayuda de JuicyPixels para llevar a cabo esta tarea, esto es debido a que ambas librerías solamente pueden leer y escribir en formato BMP. Pero, si lo que quiere hacer son funciones simples, o no quiere aprovechar el paralelismo que nos proporcionan estas librerías, puede que valga la pena mejor efectuar todo en JuicyPixels.

Aunque existen *wrappers* de Accelerate (accelerate-io-JuicyPixels) y Repa (JuicyPixels-repa), la estructura de datos que devuelven puede ser difícil de entender, por lo que si usted quiere trabajar con imágenes de JuicyPixels en ambas librerías, puede adaptar o copiar los módulo JuicyRepa⁷ o JuicyAccelerate⁸ del repositorio del proyecto.

5.3.4. FFmpeg-light

ffmpeg-light es un módulo desarrollado en Haskell que nos permite realizar unos enlaces mínimos a **ffmpeg**, una biblioteca donde podemos grabar y convertir vídeos a otros tipos de formatos. En este apartado se darán una serie de recomendaciones si queremos hacer uso de este módulo para trabajar con vídeos en Haskell:

- Cuando queramos instalar la herramienta **ffmpeg** en Windows, hay que instalar la versión correcta para que funcione el módulo, si no, nos encontraremos con muchos fallos. Por lo que revisando una issue de la librería⁹, podemos ver que si queremos instalar ffmpeg-light-0.14 en Windows, podemos instalar **ffmpeg 4.4**.

⁷<https://github.com/kennyfh/TFG-kenflohua/blob/main/src/JuicyRepa.hs>

⁸<https://github.com/kennyfh/TFG-kenflohua/blob/main/src/JuicyAccelerate.hs>

⁹<https://github.com/acowley/ffmpeg-light/issues/64>

- Como se pudo ver en la etapa de desarrollo, el módulo `ffmpeg-light` solamente acepta escribir imágenes de JuicyPixels que tenga como valores del tipo Pixel8 (Word8), PixelRGB8 o PixelRBGA8. Esto quiere decir que si queremos realizar algunos algoritmos como Sobel, donde el tipo de Pixeles que devuelven es un PixelF, tendremos que transformarlos a Pixel8, donde muchas veces lo que realizamos es truncar los valores, produciendo que no se vea la imagen.
- Cuando estamos realizando la generación de los videos en Accelerate, existe un problema de cuello de botella debido a tener que traernos de nuevo todas las matrices integradas en el lenguaje Accelerate de nuevo a Haskell para poder transformarlos a JuicyPixels, que es el formato con el que se generan los vídeos, por lo que si usted va a trabajar con grandes vídeos en Accelerate, lo mejor es que busque otra forma de generar vídeos que haciendo uso de `ffmpeg-light`.

5.3.5. Criterion

`Criterion` es el módulo que nos proporciona una forma potente pero fácil de escribir pruebas de rendimiento en Haskell ¹⁰. Si bien esta librería se esfuerza por automatizar la mayor parte posible del proceso de evaluación de nuestras funciones, hay una serie de conceptos que debemos prestar atención para realizar unas correctas pruebas de rendimiento:

- Debido a que las mediciones serán tan buenas como el entorno donde se compilan, así que intente asegurarse de no abrir más procesos cuando esté ejecutando pruebas de rendimiento
- A causa de elegir mal el tipo `Benchmarkable` en algunas funciones, los resultados no estaban saliendo bien debido a que no se estaban ejecutando en las mismas condiciones, por lo que tenga mucho cuidado cuando elijas `nf` y `whnf`.

Por lo que se dejan algunas pautas que puedes seguir para elegir entre los tipos `Benchmarkable`:

- Si el resultado es algo simple como un `Float`, lo más seguro es utilizar `whnf`, aunque si usas `nf` no habrá ningún costo adicional.
- Si el resultado es una estructura perezosa, o una mezcla de perezosa y estricta, debería emplear evaluar todo el resultado haciendo uso de `nf`.
- Cuando se ejecutan varias pruebas de rendimiento en una sola invocación del programa, a veces puede haber interferencias, por lo que para evitar eso, cuando necesite resultados muy confiables, ejecute cada medición como una invocación separada del programa.
- Este módulo para poder realizar pruebas de rendimiento, necesita que existan instancias de `NFData` de sus tipos de datos para que se asegure que toda la estructura de datos se evalúe por completo, por lo que módulos como Accelerate o los tipos predeterminados de Haskell ya tienen estas instancias, pero, existen

¹⁰<http://www.serpentine.com/criterion/tutorial.html>

otras librerías que no viene con estas instancias, como Repa, por lo que habrá que crear nuestras propias instancias para que funcionen las pruebas.

- Si queremos una forma más precisa de medir el tiempo en Criterion, podemos utilizar la estimación OLS (`time`) en vez de la estimación media (`mean`), debido a que este tipo de medición elimina la sobrecarga de medición y otros factores constantes.

A. Apéndices

A.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.

A.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. Aunque por experiencia personal, en sistemas RedHat, CentOS o derivados puede traer problemas el instalar 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 [48].

A.1.2. Dependencias

Además de tener instalado el lenguaje de programación Haskell, deberemos de instalar **Stack** [49], 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 A.1: Instalación de Haskell

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

Extracto de código A.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 A.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.[\[50\]](#)

FFmpeg

Para poder leer y escribir vídeos, es necesario instalar FFmpeg. Por lo que se requerirán las siguientes librerías externas:

```
ffmpeg libavutil-dev libavformat-dev  
libavcodec-dev libswscale-dev libavdevice-dev
```

Extracto de código A.4: Instalación de FFmpeg

A.1.3. Ejecución del programa

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

B. Bibliografía

- [1] Rick Merritt. What Is Accelerated Computing?, September 2021. URL <https://blogs.nvidia.com/blog/2021/09/01/what-is-accelerated-computing/>.
- [2] Wire HPC. Using cloud-based, gpu-accelerated ai for fraud detection, May 2022. URL https://www.hpcwire.com/solution_content/microsoft-nvidia/using-cloud-based-gpu-accelerated-ai-for-fraud-detection/. Sponsored content by Microsoft/NVIDIA.
- [3] NVIDIA. Tus aplicaciones creativas favoritas aceleradas con GPU NVIDIA, 2022. URL <https://www.nvidia.com/es-es/studio/software/>.
- [4] Alex Handy. Everyone's talking about Haskell, July 2009. URL <http://web.archive.org/web/20130203023724/http://www.sdtimes.com/blog/post/2009/07/27/Everyonee28099s-talking-about-Haskell.aspx>.
- [5] 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>.
- [6] RunAI. HPC GPU, 2022. URL <https://www.run.ai/guides/hpc-clusters/hpc-gpu>.
- [7] NVIDIA. ¿GPU vs. CPU? ¿Qué es la computación por GPU? | NVIDIA, 2022. URL <https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/>.
- [8] 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.
- [9] Jesús Adrian Toro Sanchez. Paralelismo, tipos, July 2016. URL <https://arqcompingener.wordpress.com/2016/07/21/paralelismo-tipos/>.
- [10] Camilo Mosquera and Santiago Peña. Programación paralela, 2020. URL http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#four.
- [11] Michael McCool, Arch D. Robison, and James Reinders. Chapter 3 - patterns. In Michael McCool, Arch D. Robison, and James Reinders, editors, *Structured Parallel Programming*, pages 79–119. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: <https://doi.org/10.1016/B978-0-12-415993-8.00003-7>. URL <https://www.sciencedirect.com/science/article/pii/B9780124159938000037>.
- [12] Michael McCool, Arch D. Robison, and James Reinders. Chapter 5 - collectives. In Michael McCool, Arch D. Robison, and James Reinders, editors, *Structured Parallel Programming*, pages 145–177. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: <https://doi.org/10.1016/B978-0-12-415993-8.00005-0>. URL <https://www.sciencedirect.com/science/article/pii/B9780124159938000050>.

- [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] 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/.
- [15] Kamran Karimi, Neil Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *Computing Research Repository - CORR*, arXiv:1005.2581, 05 2010.
- [16] NVIDIA Developer. Cuda zone - library of resources, July 2017. URL <https://developer.nvidia.com/cuda-zone>.
- [17] Miran Lipovača. Introduction - so what is haskell, 2011. URL <http://learnyouahaskell.com/introduction#so-whats-haskell>.
- [18] Techopedia. What is Parallel Functional Programming?, 2019. URL <http://www.techopedia.com/definition/33988/parallel-functional-programming>.
- [19] Games Solutions Architect Alfredo. A pragmatic look at functional programming: abstraction, parallelism and testing (Part 2-3), February 2020. URL <https://www.theworkshop.com/es/blog/a-pragmatic-look-at-functional-programming-abstraction-parallelism-and-testing-part-2-3>.
- [20] Universitet Datalogisk Institut på Københavns. Why Futhark?, 2020. URL <https://futhark-lang.org/>.
- [21] HaskellWiki. Language and library specification, 2020. URL https://wiki.haskell.org/Language_and_library_specification.
- [22] Fernando Sancho Caparrini. Haskell: el lenguaje funcional, 2016. URL <http://www.cs.us.es/~fsancho/?e=110>.
- [23] Daniel Eddelund and Oscar Söderlund. Repa 3 tutorial for curious haskells, 2013. URL <http://www.cse.chalmers.se/edu/year/2013/course/pfp/Papers/RepaTutorial13.pdf>.
- [24] Simon Marlow. Parallel and concurrent programming in haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, page 339–401, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642320958. doi: 10.1007/978-3-642-32096-5_7. URL https://doi.org/10.1007/978-3-642-32096-5_7.
- [25] The DPH Team. Data.Array.Repa, 2015. URL <https://hackage.haskell.org/package/repa-3.4.1.5/docs/Data-Array-Repa.html>.

- [26] Michael McCool, Arch D. Robison, and James Reinders. Chapter 7 - stencil and recurrence. In Michael McCool, Arch D. Robison, and James Reinders, editors, *Structured Parallel Programming*, pages 199–207. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: <https://doi.org/10.1016/B978-0-12-415993-8.00007-4>. URL <https://www.sciencedirect.com/science/article/pii/B9780124159938000074>.
- [27] Simon Peyton Jones, Ben Lippmeier, and Gabriele Keller. Efficient parallel stencil convolution in haskell. In *Submitted to ICFP 2011*, January 2011. URL <https://www.microsoft.com/en-us/research/publication/efficient-parallel-stencil-convolution-in-haskell/>.
- [28] 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.
- [29] Trevor McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, UNSW Sydney, 2015. URL <http://hdl.handle.net/1959.4/55818>.
- [30] The Accelerate Team. accelerate, 2020. URL <http://hackage.haskell.org/package/accelerate>.
- [31] Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Image analysis - intensity histogram, 2003. URL <https://homepages.inf.ed.ac.uk/rbf/HIPR2/histogram.htm>.
- [32] 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.
- [33] 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>.
- [34] Roman Leshchinskiy. Data.Vector, 2010. URL <https://hackage.haskell.org/package/vector-0.12.3.1/docs/Data-Vector.html>.
- [35] The University of Glasgow. Control.parallel.strategies, 2010. URL <https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html>.
- [36] TechTarget Contributor. What is grayscale?, 2010. URL <https://www.techtarget.com/whatis/definition/grayscale>.
- [37] Charles Poynton. Color FAQ - Frequently Asked Questions Color, 2006. URL http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToc11.
- [38] 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>.

- [39] Mark Seemann. Kleisli composition, April 2022. URL <https://blog.ploeh.dk/2022/04/04/kleisli-composition/>.
- [40] 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>.
- [41] 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>.
- [42] 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>.
- [43] 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>.
- [44] Bryan O'Sullivan. criterion: a haskell microbenchmarking library, 2014. URL <http://www.serpentine.com/criterion/>.
- [45] Wayne Rasband. ImageJ User Guide, 2011. URL <https://imagej.nih.gov/ij/docs/user-guide-USbooklet.pdf>.
- [46] Christopher Kanan and Garrison W. Cottrell. Color-to-grayscale: Does the method matter in image recognition? *PLOS ONE*, 7(1):1–7, 01 2012. doi: 10.1371/journal.pone.0029740. URL <https://doi.org/10.1371/journal.pone.0029740>.
- [47] Anisha Swain. Noise filtering in digital image processing, July 2020. URL <https://medium.com/image-vision/noise-filtering-in-digital-image-processing-d12b5266847c>.
- [48] NVIDIA Developer. CUDA GPUs - Compute Capability, June 2012. URL <https://developer.nvidia.com/cuda-gpus>.
- [49] Commercial Haskell Group. The haskell tool stack, 2015. URL <https://docs.haskellstack.org/en/stable/README/>.
- [50] NVIDIA Developer. CUDA Toolkit 11.7 Downloads, April 2021. URL <https://developer.nvidia.com/cuda-downloads>.