Escuela Técnica Superior de Ingeniería del Diseño

motores térmicos

TRABAJO DE FIN DE GRADO

Grado en Ingeniería Aeroespacial

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

USO DE HERRAMIENTAS DE PARALELIZADO PARA LA OPTIMIZACIÓN DE ALGORITMOS DE PROCESADO DE IMÁGENES EN CHORRO DIÉSEL

Memoria del Trabajo de Fin de Grado presentada, en la Escuela Técnica Superior de Ingeniería del Diseño de la Universidad Politécnica de Valencia, por:

Omar Diab Pascual

Dirigido por:

Jose Vicente Pastor Soriano

Valencia, 7 de Septiembre de 2017

GLOSARIO DE SÍMBOLOS Y ACRÓNIMOS

ALU: *Unidad de Control Aritmética*

API*: Application Programming Interface*

CFD: *Dinámica de Fluidos Computacional*

CMOS*: Semiconductor Complementario de Óxido Metálico*

CPU: *Unidad de Procesamiento Central*

CUDA: *Arquitectura Unificada de Dispositivos de Cómputo*

DMMT*: Departamento de Máquinas y Motores Térmicos*

EGR*: Recirculación de Gases de Escape*

GPU: U*nidad de Procesamiento Gráfico*

GPUPU*: General-Purpose Computing on Graphics Processing Units*

IR*: Radiación Infrarroja*

LTC*: Combustión a Baja Temperatura*

LEI*: Light Extinction Imaging*

LII*: Incandescencia Inducida por Láser*

MCIA: *Motor de Combustión Interna Alternativo*

MEC*: Motor de Encendido por Compresión*

MEP*: Motor de Encendido Provocado*

NAN*: Not A Number*

NL*: Luminosidad Natural*

PC*: Personal Computer*

PIB*: Producto Interior Bruto*

PMS*: Punto Muerto Superior*

RAM*: Random Access Memory*

SPMD*: Single Program Multiple Data*

RESUMEN

A lo largo de los últimos años las técnicas ópticas de medida y de visualización han venido siendo empleadas en multitud de estudios relacionados con los MCIA gracias a que son técnicas no intrusivas[1] que permiten adquirir un profundo conocimiento acerca de los más básicos procesos físicos y químicos que tienen lugar en una combustión, además de otra serie de potenciales bondades **[1]**.

La caracterización experimental de la inyección Diesel – ámbito en el que se centra especialmente la mayor parte del trabajo de Investigación que realiza el CMT – Motores Térmicos – requiere de estudios experimentales complejos que acogen a un amplio elenco de factores potencialmente influyentes en el comportamiento del chorro inyectado. Más concretamente, al emplear técnicas de visualización, se hace necesario plantear el correspondiente análisis promoviendo un elevado número de imágenes [2] por ensayo. Además, el estudio riguroso de un sistema de inyección, considerando la precisión deseada frente a la elevada variabilidad del evento en estudio, requiere de un cuantioso número de repeticiones para que pueda proporcionar resultados estadísticamente significativos. Todo esto implica la obtención de varios miles de imágenes por ensayo y estudio, introduciendo así el marco en el que se halla inmerso el presente trabajo: el análisis de la información mediante sistemas automáticos (algoritmos) de procesado de imágenes [3] .

Dicho procesado ha requerido de un gran tiempo invertido por parte de nuestro grupo de investigación, centrado en la elaboración de una serie de códigos en MatLab que permitan abordar y analizar con buena confianza los resultados obtenidos en los experimentos a partir de la sucesión de las imágenes captadas.

En este contexto, el coste y tiempo de cálculo requeridos son significativamente elevados – desde varias horas a varios días, según el número de casos a procesar – lo que lleva a la búsqueda de herramientas de programación en paralelo con objeto de agilizar los cálculos y, por ende, tratar de alcanzar la mejor y más completa optimización de los citados algoritmos. Esto se ha abordado introduciendo una nueva forma de programar, haciendo uso de la denominada arquitectura CUDA, aplicada a los scripts de MatLab ya implementados.

---

[1] *Esto hace alusión a que no interfieren de forma directa en los fenómenos físicos que se pretenden analizar, esto es, la radiación electromagnética incidente y su interacción con las moléculas/átomos no perturban ni la química ni el comportamiento del flujo. No obstante, sí presentan la necesidad de practicar accesos ópticos adecuados en el motor.*

[2] *A modo de ejemplo, en un ensayo típico se suelen tomar en torno a unas 10 imágenes por instante, con un paso temporal de unos 50 μs durante 2 o 3 ms. Con todo, se requieren del orden de unas 500 imágenes por ensayo.*

[3] *En este caso, haciendo uso de scripts y funciones programadas en MatLab.*

AGRADECIMIENTOS

Dar las gracias de corazón al CMT – Motores Térmicos por la formación que me ha brindado durante una amplia parte de la carrera y, con mayor intensidad, a lo largo de este último curso que ha permitido mi especialización en la rama de aeromotores.

Agradecer al grupo de Técnicas Ópticas mi incorporación y, en especial, a mis tutores Jose Vicente y Mattia, los cuales me acogieron con los brazos abiertos desde el primer día y, sin los cuales, este proyecto no habría podido salir adelante.

A mi familia, amigos y compañeros por haber creído en mi, apoyarme y hacerme este largo y duro camino más ameno respectivamente.

En especial, hacer patente mi inmenso y eterno agradecimiento a Marina, ella sabe bien por qué…

ÍNDICE

1. INTRODUCCIÓN

1.1. Motivación

Elaborar un Trabajo de Fin de Grado persigue recopilar, concentrar y focalizar una parte del conocimiento adquirido durante la formación larga, con el fin de poder abordar con garantías el planteamiento y posterior resolución de un problema o aplicación concreta, surgida de una necesidad real. Bajo estas premisas parte el presente trabajo, que ha sido llevado a cabo en el CMT – Motores Térmicos de la Universitat Politècnica de València. Concretamente, el principal objetivo radica en tratar de optimizar los algoritmos empleados por nuestro grupo de investigación, los cuales permiten procesar las imágenes adquiridas experimentalmente – *mediante técnicas ópticas* – en ensayos en maquetas ópticas, extrayendo así de las mismas los resultados de los ensayos realizados.

Todo ello haciendo especial hincapié en el carácter eminentemente académico que envuelve al citado proyecto, siendo este documento requisito para la obtención del Título de Grado en Ingeniería Aeroespacial en la Escuela Técnica Superior de Ingeniería del Diseño de la Universitat Politècnica de València. La tutorización del mismo ha sido realizada por el profesor Jose Vicente Pastor y como cotutor por parte del doctorando Mattia Pinotti.

Hoy día la industria de la automoción se ha destapado como uno de los pilares económicos con mayor preponderancia en países desarrollados como España e influye de manera determinante en indicadores como el PIB, así como en el desarrollo y crecimiento de poblaciones o en las políticas medioambientales **[2]**, entre otros. Precisamente éstas últimas promueven que, cada vez más, la industria del motor invierta una creciente cantidad de recursos – económicos, tecnológicos y humanos – al (I+D). El objetivo es tratar de hallar un compromiso en el "trade-off" que liga conceptos como los de mejora del rendimiento motor y la disminución de emisiones contaminantes [4] .

La importancia de los motores Diesel ha ido "in crescendo" en la última década gracias especialmente a que en 1997 **[3]** fue introducido el concepto de inyección directa (ID) y que ha permitido dar un salto cualitativo en la reducción tanto del consumo de combustible como de las emisiones generadas [5] . En este sentido, el cometido de la Ingeniería no es otro que el de tratar de plantear soluciones con un punto de encuentro entre el rendimiento global de la combustión y las progresivas reducciones de agentes contaminantes.

---

[4] *En la mayoría de casos estos dos objetivos resultan imposibles de cumplir de manera simultánea, pues la mejora del uno deriva en el deterioro del otro y viceversa.*

[5] *Notar que la introducción de la ID supuso un desplazamiento de la curva de "trade-off" hacia arriba, en tanto en cuanto a que permitió la mejora simultánea del rendimiento motor y de las emisiones*

En este sentido, la tendencia actual va en la dirección del diseño, construcción y testado de motores que, sin perder un nivel significativo de prestaciones, cumplan estricta y rigurosamente con las cada vez más duras legislaciones vigentes. En la Unión Europea se encuentra en vigor ahora mismo la llamada normativa "*Euro VI (2014)*" en lo que concierne a las emisiones contaminantes. La Figura 1.1 recoge una gráfica en la que se aprecia como el transporte por carretera prevalece, con mucho, sobre el resto, y otra con los estándares Euro desde que fuera introducida la Directiva 88/77 [6]. Cabe notar la evolución desde normativas muy laxas – especialmente en lo que concierne a partículas sólidas (con una nula restricción)– a cifras de emisión muy respetables en la actualidad.

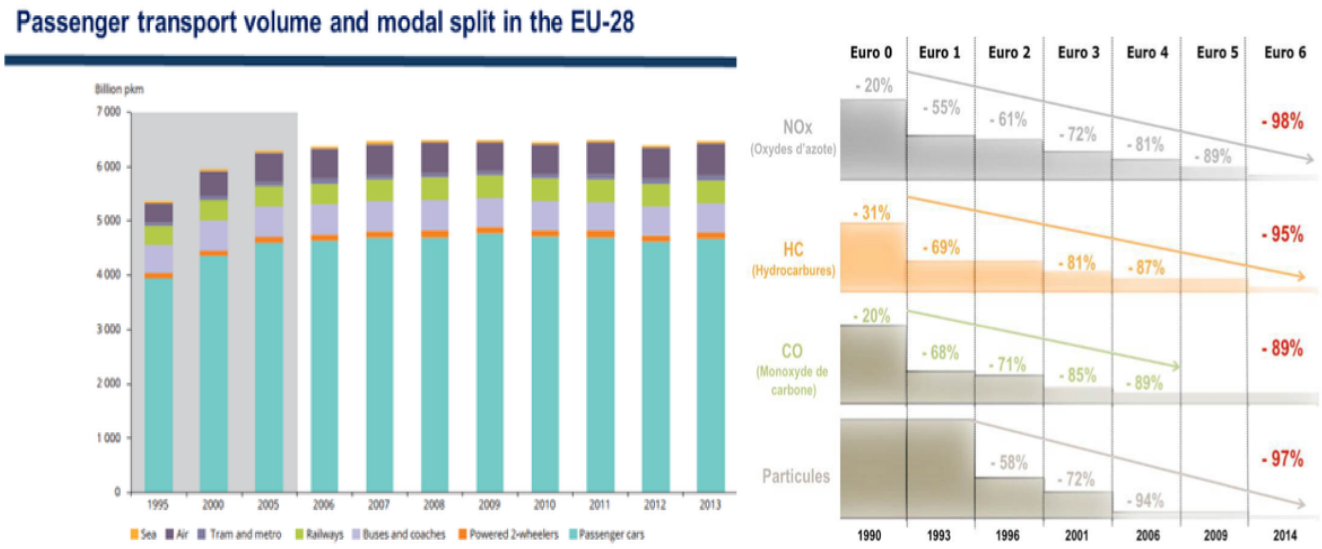


Figura 1.1. Evolución histórica de: a) Tasas de uso de diferentes medios de transporte en pasajeros en la Unión Europea. b) Normativas de restricción de emisiones contaminantes .

De la anterior figura se desprenden los principales contaminantes: hidrocarburos sin quemar (HC) y monóxido de carbono (CO) para motores MEP o gasolina y, por parte de los motores Diesel o MEC, óxidos de nitrógeno (NO_x) y partículas sólidas – principalmente hollín – que dan lugar a la formación de humos potencialmente tóxicos. Precisamente éstas últimas colocan a los motores Diesel en el punto de mira, ya que se asocian con graves patologías respiratorias y que, cada vez más, afecta de manera preocupante a las grandes ciudades.

Y es en este amplio contexto en el que se enmarca buena parte del trabajo de Investigación que desarrolla el *CMT-Motores Térmicos* de la Universitat Politècnica de València. Se centra especialmente en estudiar los complejos procesos termofluidodinámicos que tienen lugar dentro de un MEC, abordando los prismas tanto teórico como experimental.

[6] *Entra en vigor en 1990, fecha a partir de la cual la UE empieza a obligar a los fabricantes a rebajar los límites de emisiones contaminantes de sus MCIA. No obstante, la primera normativa reguladora data de 1970.*

1.2. Antecedentes

Lo que se ha expuesto en el anterior apartado sirve como punto de partida y permite centrar el motivo que da soporte a las investigaciones actuales y futuras. Tal y como ya se ha apuntado, la tendencia en los diseños en MCIA viaja en la dirección de rebajar las emisiones contaminantes y tratar así de ajustarse todo lo posible a las normativas vigentes. Cabe resaltar que las normativas Euro o *Estándares* exhiben un carácter de "tecnología neutral", lo que hace alusión a que los fabricantes de motores disfrutan de total libertad a la hora de seleccionar las técnicas que crean oportunas, siempre y cuando los límites establecidos continúen siendo respetados con rigurosidad. Esto lleva, sin duda, a que sea de importancia capital elucidar, con el mayor detalle posible, la naturaleza de los fenómenos – físicos y químicos – que dominan la combustión en el cilindro, puesto que son responsables directos de la formación de los agentes contaminantes y, por ende, lo que va a permitir plantear efectivas y eficientes soluciones tecnológicas [7] . De entre las diferentes estrategias se erigen dos generales: el impedimento de la formación de contaminantes implementando sistemas de control (atacar el problema desde el origen) y el post-tratamiento de los residuos de la combustión.

En los últimos años las *Técnicas Ópticas* y de visualización han venido siendo muy empleadas en los diferentes estudios llevados a cabo en MCIA gracias a su nula intrusividad ya que, salvo por la necesidad de practicar los accesos ópticos adecuados, no ejercen influencia alguna en los fenómenos físicos que se pretenden analizar. El *CMT-Motores Térmicos* dispone de maquetas y motores experimentales, ópticamente accesibles, que permiten visualizar los procesos de inyección y de combustión. En ellas se pueden reproducir, en mayor o menor grado, las condiciones físicas que se dan en un MEC real, además de aislar fenómenos que permiten medir parámetros físicos de interés.

Caracterizar experimentalmente la inyección Diesel requiere de estudios muy amplios, por la gran cantidad de factores que llegan a influenciar el comportamiento del chorro. La utilización de técnicas de visualización constituye una potente herramienta, capaz de arrojar mucha luz acerca de la fenomenología de lo que está ocurriendo en el seno del motor [8] , pero comúnmente se hace necesario procesar una ingente cantidad de imágenes por experimento. Además, el estudio de un sistema de inyección requiere de bastantes ensayos para poder plantear adecuadamente una combinatoria y los estudios paramétricos pertinentes. Esto implica el manejo de miles de imágenes que hace necesario el empleo de sistemas automáticos de procesado o, lo que es lo mismo, la implementación de algoritmos – en *MatLab* habitualmente – que, al manipular las imágenes como matrices de datos, deriva en altos tiempos de cálculo y costes computacionales.

---

[7] *Son muchos los factores que afectan a las características de la combustión, como por ejemplo el diseño de la propia cámara de combustión, el del sistema de inyección, el régimen de giro del motor, el punto de inicio de la inyección, el dosado, la sobrealimentación, el tipo de combustible…etc.*

[8] *Permiten la medida de parámetros característicos como la penetración, el ángulo de apertura…etc.*

Los códigos base empleados habitualmente en nuestro grupo de investigación del DMMT para procesar las imágenes tomadas en los experimentos, por parte de las diversas cámaras rápidas especializadas disponibles, se han ido desarrollando, puliendo y mejorando durante los últimos años, hasta alcanzar la versión con la que se trabaja actualmente. El punto de partida se puede atribuir a los trabajos de investigación que tratan de desarrollar y poner a punto las herramientas de procesado de imágenes que permitan medir los parámetros característicos del chorro Diesel **[4], [5]**. De hecho, los propios resultados experimentales, obtenidos del análisis de las imágenes, se suelen combinar con los que arrojan otras técnicas experimentales y/o teóricas de tal modo que, todo el compendio de conocimiento que se va adquiriendo sobre los procesos que tienen lugar en el motor, presten una inestimable ayuda en lo concerniente a la búsqueda de soluciones óptimas en las que prestaciones y emisiones se mantengan en niveles adecuados. Todos estos trabajos se han venido realizando desde hace años en el *CMT – Motores Térmicos*, dando lugar a Tesis Doctorales y publicaciones que pueden ser considerados antecedentes del trabajo de investigación que aquí se presenta y que trata de aplicar técnicas de paralelizado – basadas en la arquitectura CUDA – (que más tarde será introducida y detallada) con el fin de obtener códigos optimizados que permitan reducir significativamente tanto el *tiempo de cálculo* como el *coste computacional* que llevan asociados.

1.3. Objetivo del Proyecto

En el Trabajo de Fin de Grado que se presenta en este documento, se emplean herramientas de programación en paralelo basadas en la arquitectura CUDA. **El objetivo principal de este trabajo es el de evaluar el potencial que exhibe la paralelización, de una parte de los cálculos, en algoritmos para el procesado de las imágenes adquiridas en el estudio de la combustión en chorros Diésel por parte del grupo de Técnicas Ópticas del *CMT-Motores* de la Universitat Politècnica de València, analizando las mejoras en cuanto a la reducción del tiempo de cálculo y coste computacional.** En particular, se va a proceder al optimizado de un código base, implementado en *MatLab,* para procesar imágenes que han sido captadas por cámaras rápidas haciendo uso de una Técnica Óptica denominada LEI ("*Light Extinction Imaging*") que se basa en la detección de la emisión lumínica de las partículas de hollín, generadas a lo largo del evento de la combustión, para poder inferir su concentración y distribución en una llama de difusión. El concepto general que se considera como discriminante para la optimización del código es, esencialmente, su tiempo de cálculo, no su coste computacional, ya que la cantidad de imágenes a procesar para cada caso en estudio hace que, una reducción – incluso pequeña – del tiempo necesario para el procesado de una sola repetición, conlleve una importante ventaja a la hora de obtener los resultados de un caso de estudio completo.

El objetivo fundamental de este trabajo, es decir, la optimización de los tiempos de cálculo del código MatLab de procesado de imágenes, puede dividirse en una serie de sub-objetivos que ayudarán a conseguir el principal. Estos sub-objetivos concretos son:

- Paralelización del código base de nivel 1: paralelizado en la Unidad de Procesamiento Central (CPU).

• Paralelización del código base de nivel 2: paralelizado en la Unidad de Procesamiento Gráfico (GPU) con arquitectura CUDA core.

• Comparación del código optimizado con el original y análisis de las efectivas ventajas y mejoras obtenidas.

• Elección de la paralelización óptima y obtención del código alfa a través del análisis de las ventajas/desventajas que se asocian a cada uno de los tipos de paralelización conseguidas en los precedentes sub-objetivos.

Para lograr estos objetivos, el trabajo se ha estructurado en las siguientes etapas:

1. – Prospección bibliográfica de trabajos de paralelizado haciendo uso de las herramientas que ofrece el *Parallel Pool de MatLab*, empleando la CPU.

2. – Prospección bibliográfica de trabajos de paralelizado haciendo uso de la arquitectura CUDA y las herramientas que ofrece el *Parallel Computing Toolbox* de MatLab, empleando la GPU.

3. – Evaluación de los tiempos de cálculo y de los costes computacionales obtenidos por los paralelizados de los niveles 1 y 2 con respeto a los del código base.

4. – Síntesis de los resultados y valoración de las mejoras que arrojan los algoritmos optimizados para el desarrollo del código alfa.

1.4. Estructura y Desarrollo

A la hora de estructurar y organizar todo lo que en esta memoria se recoge se ha optado por seguir las líneas generales protocolarias que todo trabajo experimental exhibe.

De este modo y, tras un breve primer apartado de Introducción (Capítulo 1) que sirve como marco general para establecer un clima de acomodación al lector, se pasará a la descripción de los procesos físicos asociados al tipo de combustión Diesel de inyección directa (Capítulo 2) que trata de proporcionar el sustento teórico[9] necesario para la correcta comprensión de las Técnicas Ópticas que el grupo de investigación emplea con asiduidad para plantear y resolver la problemática de sus experimentos. Asimismo, se introducirá una somera explicación teórico – práctica del fundamento y motivación del uso de las citadas Técnicas Ópticas, haciendo especial incidencia en la técnica particular que va asociada al código base con el que se va a trabajar a lo largo del presente proyecto (LEI). Para ello se buscará apoyo en las experiencias precedentes, llevadas a cabo por diferentes autores y que, sin duda, constituyen una valiosa parte de la literatura ligada a este ámbito.

[9] *N*o se pretende hacer una revisión exhaustiva, pues no es el fin de este trabajo.

Acto seguido se dará paso a las herramientas y a los métodos experimentales que se utilizan (Capítulo 3). Dado que se trata de un proyecto informático en esencia, la idea es la de

introducir como herramienta empleada un ordenador, junto con el concepto de programación en paralelo y ayuda de la llamada arquitectura CUDA, en la que se basan las mas modernas tarjetas gráficas producidas por NVIDIA y que ofrece la posibilidad de aprovechar la potencia de cálculo de la memoria GPU, permitiendo realizar cálculos con una considerable mayor rapidez. En cuanto a la metodología utilizada, se presentan las técnicas de paralelización en CPU y en GPU en el entorno de *MatLab* respectivamente, junto con algún pequeño ejemplo de paralelizado que permita al lector intuir y adentrarse en las ventajas que esta potencial herramienta puede ofrecer.

La metodología experimental (Capitulo 4) reportará un esquema detallado de la organización seguida para la obtención de los algoritmos derivados del código base – fruto de la aplicación de los diferentes niveles de paralelización – junto a los estudios que se han propuesto para analizar las bondades y deficiencias existentes frente a la opción de no paralelizar. La idea, a su vez, es que haga las veces de guía para el posterior análisis de los resultados obtenidos (Capítulo 5), los cuales serán debidamente comentados y discutidos en relación a los objetivos marcados.

Finalmente se procederá al establecimiento de las conclusiones (Capítulo 6) donde se tratará de concentrar, en la medida de lo posible, los resultados más destacados, focalizando especialmente sobre el grado alcanzado en la consecución de los objetivos propuestos. Además, se proponen posibles ampliaciones y trabajos futuros que viajen en la misma línea con el fin de abrir caminos interesantes a nivel de investigación.

El último de los documentos corresponde al Presupuesto empleado en la realización del proyecto. Se ha estructurado de forma que se desglosan los costes en presupuestos parciales para justificar con claridad el presupuesto final.

2. CONTEXTUALIZACIÓN DEL PROBLEMA

2.1. Combustión Diesel

Si se toma como referencia la *Real Academia Española,* de las tres definiciones de combustión que en ella aparecen, se observa que la más técnica de ellas adopta la siguiente expresión: "*Reacción química entre el oxígeno y un material oxidable, acompañada de desprendimiento de energía y que habitualmente se manifiesta por incandescencia o llama*". Es precisamente esa cantidad de energía liberada en el proceso altamente exotérmico la que ofrece a los MCIA la posibilidad de generar potencia. Dentro de los MCIA se tienen dos grandes clases de motores, los Motores de Encendido Provocado, o comúnmente conocidos como de gasolina (MEP), y los Motores de Encendido por Compresión o Diésel (MEC). El concepto del encendido es básicamente donde se fundamenta la diferencia y lo que condiciona todo el resto de aspectos termofluidodinámicos, hasta el punto de que las emisiones contaminantes asociadas a cada uno de ellos se asienta en agentes químicos de distinta naturaleza. Mientras que en un motor de gasolina el encendido de la mezcla de aire y combustible se induce mediante el aporte de energía externa – generalmente haciendo saltar una chispa entre dos electrodos – los Diésel lo producen a través del autoencendido de la mezcla cuando se consiguen las condiciones termodinámicas – Presión y Temperatura – necesarias en el interior de la cámara de combustión.

Es objeto de este primer apartado del Capítulo 2 introducir el marco teórico concreto donde se asientan los experimentos que dan paso a los algoritmos de procesado y que constituyen el núcleo central del presente trabajo. Se va a hacer énfasis en el estudio de la combustión en chorros Diesel con inyección directa, puesto que son especial objeto de estudio en el DMMT y, más concretamente, en el equipo de Técnicas Ópticas.

2.1.1. Descripción General

El proceso de combustión en un chorro Diesel es un conjunto de procesos físicos [10] que abarcan toda la evolución desde que el combustible es introducido en fase líquida por el orificio de inyección en la propia cámara de combustión – con unas determinadas condiciones de presión y temperatura – hasta que el oxígeno allí presente en el aire ambiente reacciona con él **[6]**. Describiendo brevemente lo que ocurre las etapas son, en orden cronológico:

• **Frenado, difusión y ensanchamiento del chorro:** - Se produce, debido a la viscosidad, un intercambio de cantidad de movimiento entre el combustible y el aire, el cual genera un frenado y un ensanchamiento del chorro con cierto ángulo al ir éste difundiendo en su seno.

• **Englobamiento de aire:** - La cantidad de movimiento que porta el flujo inyectado induce un gradiente que arrastra radialmente a las líneas de corriente (aire) presentes en la cámara de combustión, de manera que envuelve al chorro.

---

[10] *De forma genérica (tanto en MEC como en MEP) el combustible sufre los procesos de atomización, calentamiento, evaporación, mezcla y una serie de reacciones previas precursoras de la combustión.*

• **Mezcla de combustible – aire:** - Se produce al estar éstos íntimamente en contacto.

• **Zonas de dosado en límites de inflamabilidad:** - Dentro del volumen que contiene la mezcla, corresponden a los contornos en los cuales se alcanzan condiciones de autoencendido.

• **Establecimiento de la posición de llama:** - A cierta distancia del orificio la llama se estabiliza.

Tanto las fases descritas como los procesos físicos implícitos en las mismas pueden intuirse en la Figura 2.1.

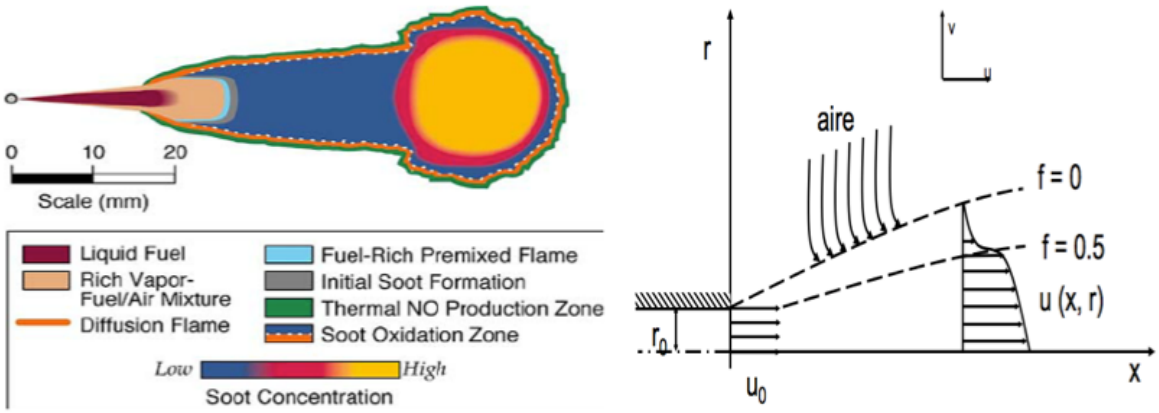


Figura 2.1. Esquema de la forma y diferentes zonas de un chorro Diésel ***[1]****. En la figura de la derecha f corresponde al dosado, r_0 al radio interior del inyector y u_0 es la velocidad inicial de inyección.*

Puede apreciarse el establecimiento de un perfil radial de velocidades u(x, r) que va desde r = 0, donde la velocidad del chorro inyectado es máxima (eje central del inyector), a un radio r tal que haga dosado nulo o, lo que es lo mismo, límite donde ya no hay chorro por frenado total del combustible (deja de difundir, encontrando únicamente aire a partir de dicho límite). La combustión tiene lugar en la llamada *Superficie Estequiométrica,* que corresponde a los límites de inflamabilidad, esto es, zonas locales donde el dosado relativo alcanza el valor unidad y la presión y la temperatura son adecuadas para producir las reacciones químicas que preludian el proceso de autoencendido.

A continuación se produce la primera etapa de encendido, que abarca desde las primeras reacciones químicas hasta el inicio de la fase rápida de liberación de calor. En esta fase – conocida como autoencendido a baja temperatura – tiene lugar la rotura de las cadenas largas de hidrocarburos en otras cadenas más cortas, así como reacciones de isomerización, dando lugar a la generación de radicales libres que emiten una radiación, de baja intensidad, pero ya medible mediante algunas técnicas ópticas. El tiempo que transcurre desde la inyección del chorro hasta que se verifica esta etapa se conoce como *tiempo de retraso del autoencendido o Cool Flame Ignition (en motores se emplea el Pressure Ignition Delay, y se detecta por presión).*

La segunda etapa de encendido es en realidad un proceso de *combustión premezclada*, que viene provocada por la acción de los radicales libres formados en la etapa anterior (siempre que se logre estabilizar el kernel de llama y no se produzca misfire), dando lugar a una reacción fuertemente exotérmica en la que se quema el oxígeno englobado y que eleva mucho la temperatura de la mezcla. En consecuencia, el quemado de combustible se dispara,

traduciéndose en un incremento brusco de la tasa de liberación de calor. En esta etapa se generan tanto los productos finales de la combustión como especies intermedias. Además, se consolida el frente de llama inicial y que caracterizará el resto de la combustión.

Finalizado el encendido, tiene lugar el comienzo de otra etapa conocida como *combustión por difusión,* y está gobernada por la cantidad de combustible que aun se está introduciendo a través del inyector. Por tanto, con una cantidad de movimiento, según la tasa de inyección que se haya elegido utilizar.

En esta etapa, el frente de llama ya consolidado, alcanza un estado cuasi-estacionario y la llama alcanza una forma característica que ya no abandonará hasta el final de la inyección. Está dividida en dos partes claramente diferenciadas, tal como se puede apreciar en la Figura 2.1: una primera – cercana a la tobera de inyección – compuesta por un tramo no reactivo y otro – aguas abajo – delimitado por el propio frente en el cual tiene lugar la combustión en condiciones estequiométricas. Por este motivo, se suele considerar que el chorro de combustible se quema en dos pasos distintos: uno de premezcla y el otro de difusión [11].

Al cesar la inyección del chorro, la estructura cuasi-estacionaria de llama se pierde, momento a partir del cual la combustión pasa a estar exclusivamente controlada por la turbulencia [12] que exhibe el flujo en la cámara de combustión y que podrá permitir o no que se consuma todo el combustible atrapado en ella. Se puede apreciar también como hay una zona interna ocupada por combustible sin quemar y productos parciales de la combustión. Esta zona constituye un depósito de hollín, que se oxida al ir alcanzando éste el frente de llama. Un resumen visual de lo que se ha ido exponiendo puede apreciarse en la Figura 2.2.

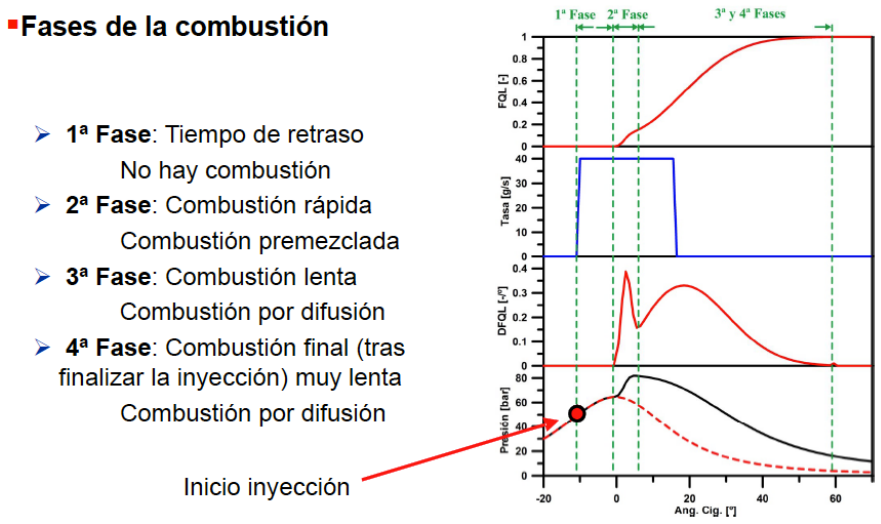


Figura 2.2. Calor total liberado, Tasa de inyección Tasa de calor liberado y Presión en el interior del cilindro en función del ángulo de giro del cigüeñal[13] para las etapas de una combustión Diésel.

[11] *Generalmente difusión turbulenta.*

[12] *Al dejar de inyectar combustible por la tobera su cantidad de movimiento axial irá disminuyendo por la acción de frenado del aire hasta que, a una cierta distancia del orificio, sea lo suficientemente pequeña en relación al perfil de velocidad radial turbulenta. El flujo turbulento se caracteriza por ser caótico, fluctuante en sus propiedades termofluidodinámicas y muy difusivo en términos de transporte.*

2.1.2. Motivos de estudio

De lo expuesto anteriormente se desprende que, comprender los fenómenos físicos asociados a cada una de las etapas que envuelven la evolución del chorro de combustible, desde que éste es inyectado, hasta que tiene finalmente lugar su combustión por autoencendido, ofrece la posibilidad de buscar y barajar entre distintas tecnologías de control que brinden una mejora global, tanto en términos de rendimiento como de emisiones. Esto justifica la constante búsqueda de información, proyectando diferentes estudios que hoy día realiza el DMMT, muchos de los cuales pueden plantearse a partir de experimentos en los que se lleven a cabo combustiones con chorro Diésel – en las condiciones de interés – y se recojan imágenes de todo el proceso haciendo uso de cámaras rápidas. En este contexto, las Técnicas Ópticas tienen su filón y un muy importante potencial.

En los últimos años, el problema de los contaminantes que emiten los motores ha enfocado las investigaciones en su reducción. Para ello, como ya se ha mencionado con anterioridad, existen dos tipos de soluciones: las que tratan los contaminantes a la salida del motor y las que se centran en el propio diseño del motor.

En este sentido, los MEC presentan grandes ventajas en cuanto a su rendimiento. Sin embargo, las emisiones contaminantes se revelan como un gran problema, en especial los NOx y las partículas de hollín. Al tratarse de motores que no suelen operar en regímenes estacionarios abarcan una gran zona de trabajo, por lo que la reducción de los contaminantes no puede focalizarse únicamente sobre un punto de operación. En la Figura 2.3 se puede observar la generación de hollín y NOx en función del dosado relativo y la temperatura. Solo así se comprenden las tendencias actuales en cuanto a diseños para reducir emisiones como son las de trabajar con dosados pobres y temperaturas bajas.

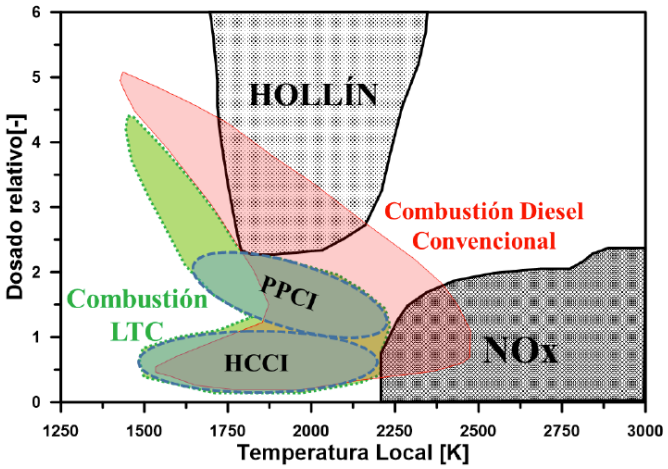


Figura 2.3. Mapa de la generación de hollín y NOx en función del Fr y la temperatura **[10]**, **[11]**.

[13] *Representa el grado de avance de cada ciclo de la combustión, en tanto en cuanto a que liga unívocamente la posición en cada instante del pistón en el interior del cilindro con los grados rotados por el cigüeñal.*

Pese a que, como se aprecia, son claramente dos agentes contaminantes los que acucian las estrategias de reducción en los MEC actuales, este proyecto se asienta en algoritmos que procesan imágenes de la combustión haciendo uso de la técnica óptica de extinción de hollín (LEI o Light Extinction Imaging). Por este motivo, se opta por abordar con especial atención tanto la naturaleza de este tipo de compuesto, como las técnicas activas que habitualmente se emplean para reducirlos en la medida de lo posible.

Se llama hollín a las partículas sólidas de tamaño muy pequeño (de 25 a 75 nm) en su mayoría compuestas de carbono impuro, pulverizado, y generalmente de colores oscuros más bien negruzcos resultantes de la combustión incompleta. Moléculas precursoras como el acetileno pueden constituir una de las posibles vías de formación, tal como se aprecia en la Figura 2.4. Consiste en nanopartículas aglomeradas con diámetros entre 6 y 30 nm. Es carbono amorfo en forma parecida al polvo. El hollín en fase gaseosa contiene hidrocarburos aromáticos policíclicos. Se asocia muy frecuentemente con cáncer y patologías pulmonares, lo que pone de manifiesto el riesgo que supone para la salud humana y la necesidad tanto de detectarlo, como de aplicar medidas en el motor que impidan su formación o bien que promuevan su eliminación antes de ser evacuados en forma de humos negros por el escape.

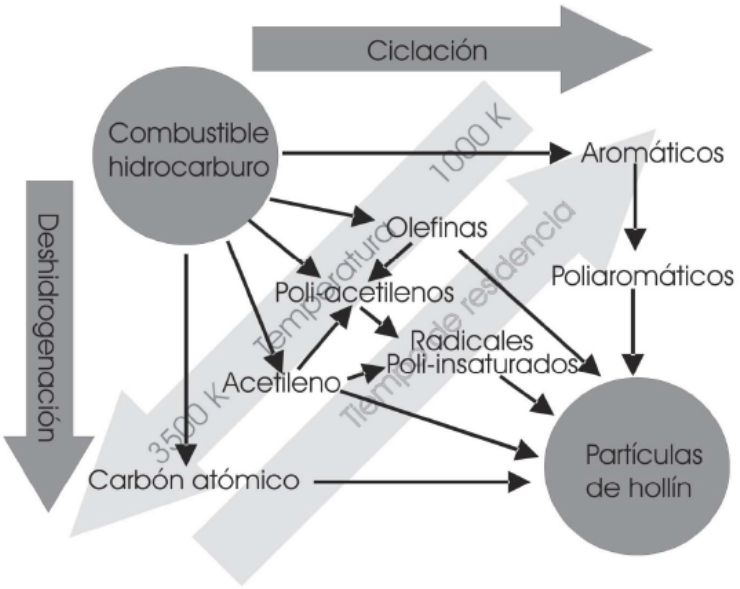


Figura 2.4. Esquema orientativo que muestra los tipos de reacciones químicas, las condiciones y las especies intermedias a partir de las cuales un hidrocarburo puede desembocar en hollín.

Tal y como ya se ha comentado, se distinguen entre soluciones activas y pasivas a la hora de atacar a las emisiones siendo las primeras, con mucho, las más interesantes, ya que se consigue atacar el problema directamente de raíz, evitando la formación de las especies contaminantes. Lo hacen mediante la búsqueda de la mejor optimización de los procesos que forman el ciclo de funcionamiento del motor, basándose en el diseño de aspectos tan relevantes como los de tamaño, cámara de combustión, sistema de formación de la mezcla (sobretodo en el caso del MEP), la distribución o el EGR, entre otros.

Existen varios tipos de soluciones activas actualmente que se emplean para reducir la cantidad de hollín que se forma en la cámara de combustión. Una de ellas es diseñando cuidadosamente el sistema de inyección y aplicando el concepto de la "*post-inyección*" **[7]**. Ésta se basa en hacer variar el último tramo de la 3ª y 4ª fase de la combustión Diésel – de acuerdo con lo que se vio en la Figura 2.2 – fomentando un repunte de la tasa de liberación de calor al inyectar una pequeña fracción de combustible tras la inyección principal (es como una inyección piloto pero en la parte final). La concentración de hollín se ve reducida porque mejora la oxidación del mismo al dar más tiempo para que el chorro englobe aire, pues se le aporta una cantidad de movimiento adicional con dicha inyección.

Otro método es el de *sobrealimentar* el motor. Tal y como se puede ver en la Figura 2.5, la sobrealimentación[14] hace aumentar la densidad de aire en el cilindro. Esto promueve un mayor englobamiento de aire y mezcla efectiva, aumentando la Temperatura Adiabática de Llama, lo que deriva en última instancia en una disminución del tiempo de retraso. Además, se consigue mejorar el consumo específico al acelerar la combustión.

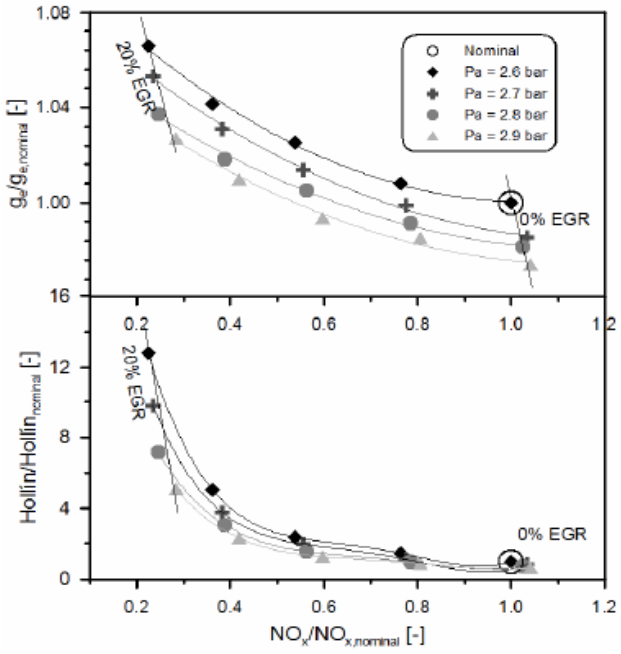


Figura 2.5. Gráficas mostrando los efectos beneficiosos de la sobrealimentación sobre el consumo específico del MEC y la reducción de los niveles de hollín.

[14] *Para un EGR dado, sobrealimentar rebaja los niveles de hollín. Sin embargo, aumentar la EGR penaliza al consumo específico y aumenta la cantidad de hollín, ya que es una solución activa antagónica, de modo que ralentiza la combustión. En este caso se reduce la Temperatura Adiabática de Llama, propiciando una disminución de los NO_x. Por tanto, se desprende el hecho de que no es posible la reducción de ambos tipos de contaminante simultáneamente, estudio que ya se publicó en* ***[8]****.*

También es posible disminuir el hollín con un adecuado *diseño de la cámara de combustión*. Esto es posible debido a que, si bien en la inyección Diésel el papel fundamental del proceso de mezcla lo ejerce el chorro, en las etapas finales, así como en inyecciones cortas (o incluso a bajas presiones de inyección) – una vez cesa la inyección, como se ha visto – el movimiento del aire es crucial para proseguir la combustión y producir la oxidación del hollín formado. En este caso la cámara debe de presentar una geometría tal que se garantice la inducción de turbulencia, pues es la que controla el proceso de mezcla en estas circunstancias. La Figura 2.6 muestra dos tipos de diseño de cámaras de combustión, una con geometría abierta, ancha y poco profunda, cuyo uso va destinado más a MEC para transporte pesado y otra con geometría reentrante, que es más profunda y se emplea en MEC en automoción.

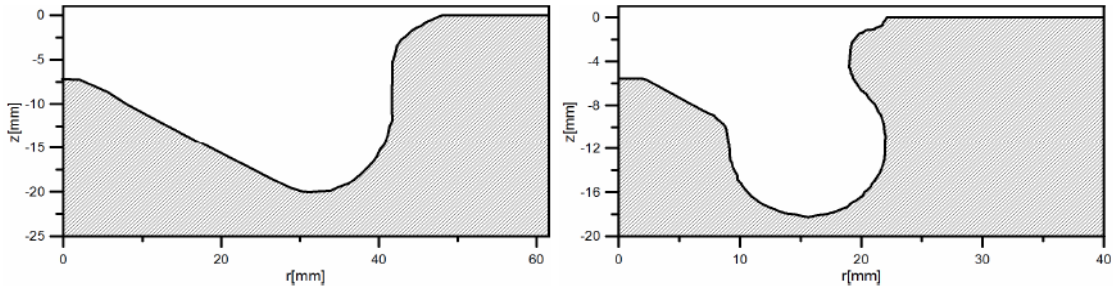


Figura 2.6. Cámaras de combustión para a) Transporte pesado (camiones) y b) Automoción (Turismos).

Finalmente, cabe destacar los procesos de combustión alternativos, cuyo objetivo es reducir de manera lo más simultánea posible la formación de humos y NO_x. Esto se consigue con la denominada *Combustión a Baja Temperatura* o LTC. Este tipo de combustiones alternativas se escinden en procesos de combustión a baja temperatura homogéneos (HCCI) y los de combustión parcialmente premezclada (PCCI). En ambos casos se reducen los NO_x por la disminución de la temperatura adiabática de llama, por la reducción de la formación vía mecanismos térmicos y por la alta energía de activación de las reacciones de formación de NO **[6].** Además, debido al incremento en los tiempos de retraso existe un mayor tiempo para la mezcla, con lo que las regiones de dosados ricos se reducen a costa de alargar la longitud del "Lift-off" [15], inhibiendo también la formación de hollín [16].

Las HCCI tienen en inconveniente de que alargar el LOL es equivalente a aumentar la penetración del chorro, lo que conlleva un alto riesgo por choque contra las paredes de la cámara de combustión, con la consecuente bajada de rendimiento del ciclo.

---

[15] *La LOL o Longitud líquida es donde se producen los fenómenos de intercambio de cantidad de movimiento entre el chorro combustible y el aire, el frenado y el ensanchamiento, así como el englobamiento de aire y la mezcla combustible – aire. Depende de muchos parámetros como la velocidad de inyección, la temperatura y la densidad. A mayor LOL más aire englobado se tiene y, en consecuencia, menor cantidad de hollín (pues no habrá tanto defecto de O_2 en la cámara).*

[16] *Existen muchos otros modos de combustión alternativos desarrollados en los últimos años, como son PCCI, RCCI, PPC, PPCI, MK, ATAC, TS, ARC…etc* ***[9].***

En las PCCI encontramos variantes, como la *inyección muy adelantada*, en la cual se inyecta el combustible en un punto avanzado de la carrera de compresión para aumentar el tiempo de mezcla (la combustión tiene entonces lugar muy cerca del PMS). También existe la posibilidad de producir una *inyección muy retrasada*, en la que la inyección se produce en el PMS o incluso en la propia carrera de expansión, lo cual da lugar a elevadas temperaturas y densidades del aire de admisión. Ello da lugar a mezclas muy buenas y se evita el choque del chorro contra la pared de la cámara de combustión, al reducir ostensiblemente el tiempo de retraso y, por ende, la LOL. El secreto de la estrategia de las combustiones premezcladas reside en que , tal y como se puede ver en la Figura 2.7, las etapas de mezcla y vaporización, de autoencendido y de combustión transcurren navegando por zonas de bajo dosado relativo y baja temperatura adiabática de llama.

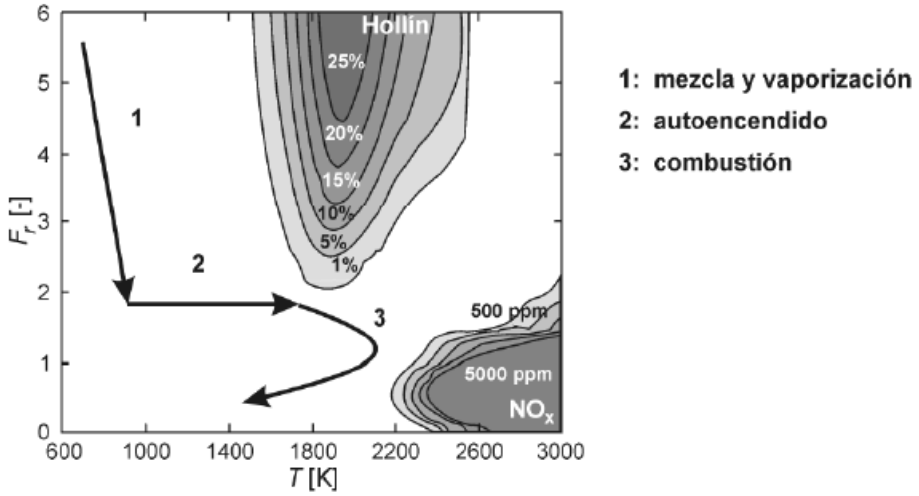


Figura 2.7. Camino que trazan las PCCI a través del mapa de generación de hollín y NO_x.

Como inconvenientes en PCCI se tiene que la variante retrasada empeora los rendimientos con respecto a la adelantada y que, en ésta además, es difícil conseguir que el tiempo de retraso sea mayor que el de inyección (pues se está inyectando muy tarde y la presión y la temperatura en cámara ya son muy elevadas en las proximidades de PMS). Esto provoca que exista una baja homogeneidad de los productos formados por zonas. En la Figura 2.3 se pueden apreciar las zonas del mapa en las que trabajan tanto la LTC homogénea como la premezclada.

2.2. Técnicas Ópticas

En el ámbito investigativo del fenómeno de la combustión, las técnicas ópticas se han destapado como una de las más relevantes herramientas experimentales. Brindan la oportunidad de adquirir un conocimiento íntimo acerca de los complejos fenómenos físico – químicos que tienen lugar en ella, bien desde una perspectiva cuantitativa o bien desde una semi-cuantitativa[17]. Entre sus fortalezas cabe destacar algunas como las elevadas resoluciones espaciales, temporales o los rangos dinámicos de sensibilidad (según la técnica concreta), además de permitir llevar a cabo la validación de modelos predictivos, tanto fenomenológicos como a través de códigos implementados en Dinámica de Fluidos Computacional (CFD).

2.2.1. Generalidades

Existe una amplia variedad de *técnicas ópticas* susceptibles de ser aplicadas al estudio de la combustión en general, tendiendo a agruparse principalmente en dos categorías: *técnicas láser avanzadas* y *técnicas de visualización*. Las primeras exigen de una notable sofisticación en aparatología y se aplican especialmente en experimentos simplificados, siendo su principal aplicación la de llevar a cabo el estudio focalizado en particulares fenómenos físico – químicos, lo que le confiere un carácter como precursor de modelos matemáticos. En lo que concierne a las segundas exigen, en contraposición, un equipamiento menos sofisticado y permiten adquirir una idea más global de los procesos analizados.

En función del tipo de técnica en concreto, es posible desde medir concentraciones de especies presentes en muestra[18] hasta magnitudes termodinámicas del campo fluido como la Presión, la Temperatura o la Velocidad. También es posible medir tamaños, concentraciones y fracciones de volumen en partículas o gotas, así como extraer información sobre dosados.

La característica común a todas las técnicas ópticas experimentales es la no intrusividad y la necesidad de adaptar accesos ópticos convenientes en el objeto o sistema de estudio, tratando de modificarlo lo mínimo posible. En la Figura 2.8 se puede ver un ejemplo generalista de montaje – dotado de equipos de medida asociados a distintas técnicas – que permite analizar el proceso de la combustión. Se hallan insertadas técnicas de los dos grandes grupos. Algunas requieren de una previa excitación por parte de la muestra, haciendo uso de un láser, mientras que otras directamente captan el tipo de radiación electromagnética concreto – a la longitud de onda que corresponda – que es capaz de emitir ésta directamente. Así, por ejemplo el detector que registra a 310 nm capta la aparición de radicales OH, el de 400 nm ofrece la posibilidad de recoger un mapa de colores con los niveles de hollín presentes en muestra y, a 514, 5 nm otro emplea la línea de visión para cuantificar la concentración del hollín a través de la caída de intensidad de la luz que atraviesa la llama.

---

[17] *Cabe decir que la cuantificación no siempre es posible, siendo además, en general difícil.*

[18] *El chorro inerte o las zonas en combustión o de productos/especies intermedias formadas.*

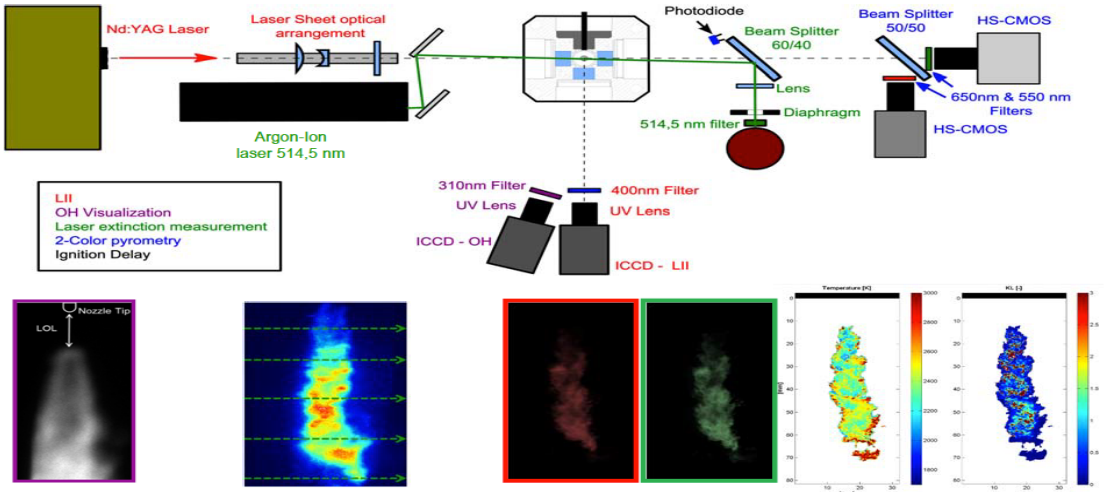


Figura 2.8. Montaje general con varios tipos de equipos instalados que ofrecen el uso de un compendio de técnicas ópticas.

En lo que concierne más específicamente a las técnicas de visualización, éstas suelen aportar información tanto del proceso de inyección como de combustión, según sea la base teórica sobre la que se apoya la técnica empleada. Como se ha visto y discutido en la Figura 2.8, se pueden registrar imágenes directamente aprovechando la emisión de radiación procedente de la muestra en estudio o bien mediante fenómenos de interacción radiación – materia a través del uso de una fuente de luz externa, que pueden ser láseres o lámparas.

Por otro lado, la emisión de radiación natural por parte de la muestra puede ser *incandescente* o *luminiscente*. En la primera – que es también conocida como radiación térmica – la luz emitida corresponde a la zona infrarroja cercana, y a parte de la visible, del espectro electromagnético, asociándose con una gran liberación de energía que tiene lugar cuando el cuerpo caliente supera cierta temperatura[19]. En la segunda tipología ocurre a la inversa, esto es, la luz es emitida con la muestra a baja temperatura. En este caso el fenómeno se debe a la emisión espontánea de un fotón, fruto de la desexcitación[20] de la molécula emisora.

En otro orden de cosas, tal como ya se ha apuntado en líneas anteriores, es objeto de este trabajo el manipular algoritmos de procesado de imágenes procedentes de ensayos en los que se estudia – en distintas condiciones y variando diversos parámetros (casos) – la combustión de un chorro Diésel, empleando la técnica LEI. Por esta razón, se optará por ir presentando primero (sucintamente) las técnicas de visualización más empleadas – omitiendo las técnicas de láseres avanzados – con el fin de adquirir una visión global en perspectiva y, ya en el último sub-apartado, la técnica LEI en particular.

---

[19] *La explicación teórica se fundamenta en la conocida Ley de Planck.*

[20] *Si la excitación proviene de la reacción química de oxidación u otro tipo, se tiene Quimioluminiscencia. Si es una fuente de naturaleza electromagnética la inductora se tendrá Fotoluminiscencia.*

Así, en los sub-apartados que le preceden, se va a proporcionar una breve introducción general acerca de los fundamentos físicos que dan soporte a algunas de las técnicas de visualización – habitualmente empleadas por el CMT – junto a una breve descripción de los tipos de medidas que pueden efectuar y las informaciones que aportan.

2.2.2. Luminosidad Natural

La técnica de *Luminosidad Natural*, o *Radiación Térmica del Hollín* corresponde a una de las más sencillas, puesto que no es otra cosa que la simple observación de la radiación que, de forma natural, emite la propia muestra[21]. Ópticamente, una llama que se puede ver con el ojo humano está dentro de la región visible. Es susceptible de ser considerada como un agregado de partículas a alta temperatura que, por la Ley de Planck, emite radiación térmica a una cierta intensidad **[12]**, siendo ésta función de la concentración y de la temperatura a la que se encuentren las partículas de Hollín, tal como refleja la expresión (2.1):

$$I_{Soot}(\lambda, KL_{2C}, T) = \left(1 - e^{-\frac{KL_{2C}}{\lambda^{\alpha}}}\right)\frac{C_1}{\lambda^5}\frac{1}{e^{\frac{C_2}{\lambda T}}-1} \quad (2.1)$$

El multiplicando en azul corresponde a la emisividad del hollín, mientras que el que está coloreado de granate corresponde a la radiación del cuerpo negro[22] **[13]**. Éste presenta idealmente un espectro de emisión (en términos de intensidad) tal como muestra la Figura 2.9. Se puede apreciar como, cuanto más caliente se encuentre el cuerpo negro, el pico de intensidad se desplazará hacia longitudes de onda cada vez más pequeñas o, lo que es lo mismo, energías más altas. Ningún cuerpo es capaz de emitir con mayor intensidad que un cuerpo negro. C_1 y C_2 son las llamadas constantes de Planck y su valor es conocido. $\alpha \approx 0{,}95 - 1{,}39$ **[14]**.

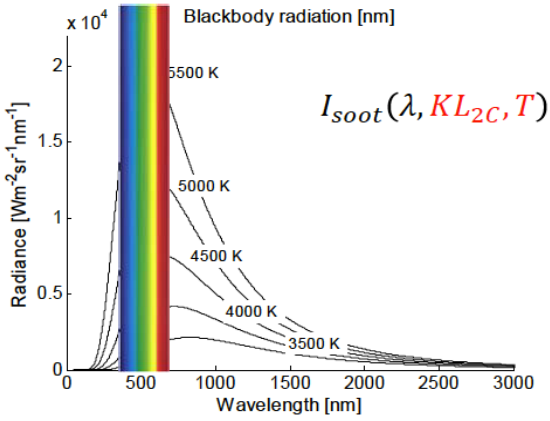


Figura 2.9. Espectro asociado a la radiación del cuerpo negro.

[21] *Llama o zona del chorro caliente con presencia de hollín.*

[22] *El cuerpo negro es considerado como aquél cuerpo que, a una temperatura dada, mayor intensidad radiativa es capaz de emitir.*

La conclusión es clara: cuanto menos temperatura tenga el cuerpo, mucho más débil será la intensidad de radiación y el pico se desplazará tanto más al IR cercano[23] (deja de ser visible). Con todo, se puede apreciar que la Intensidad lumínica con la que emite radiación el hollín es función también la longitud de onda. Por tanto, en el color rojo del espectro visible será posible visualizarlo cuando la temperatura sea tal que la intensidad alcanzada rebase el umbral perceptible por el ojo humano, lo cual ocurre durante la combustión.

El hollín empieza a formarse de forma significativa – como se vio en la Figura 2.7 – a partir de los 1400 K, siempre que el dosado local oscile por encima de 2. Se puede observar trascurrido un cierto tiempo desde el inicio de las primeras reacciones del proceso de encendido. De este modo, el fenómeno de radiación térmica puede aprovecharse de diferentes formas. En especial es posible cuantificar la producción de hollín durante la combustión en estudio, la temperatura de la llama e incluso obtener una medida de la penetración del chorro de combustible en las fases avanzadas de la combustión cuando el análisis que reportan otras imágenes asociadas a otras técnicas no son capaces de arrojar informaciones consistentes.

Para registrar la señal de incandescencia se pueden emplear sencillos sistemas de visualización con cámaras digitales relativamente simples, requiriendo accesos ópticos a la cámara de combustión o, si éstos no son fáciles de habilitar, introduciendo endoscopios en la citada cámara.

2.2.3. Método de los 2 Colores

Desarrollado en los años 40, permite estimar simultáneamente la concentración de hollín presente en la llama Diésel y la temperatura de ésta. Su fundamento está estrechamente relacionado con la técnica de la NL y la expresión (2.1) anteriormente introducida. De hecho y, como su propio nombre indica, es más propiamente un método que una técnica óptica en si. El punto de partida radica en la suposición de que la llama de hollín, percibida por un observador puntual, depende de la longitud de onda, de la temperatura y de la cantidad de hollín presente.

Se resuelve un sistema de dos ecuaciones con dos incógnitas que se obtiene gracias a la medida de dos valores de radiación I_{soot1} e I_{soot2}. Cada radiación de hollín se mide con una longitud de onda del espectro visible diferente λ_1 y λ_2. De la ecuación (2.1), por tanto, se obtiene directamente T y KL_{2C}.

---

[23] *Por esta razón decimos habitualmente que un material se encuentra al rojo vivo, puesto que al calentarse adquiere una curva de intensidad cuyos niveles – incluso lejos de su pico – empiezan a ser ya perceptibles en el visible, especialmente en el color rojo. En torno a unos 3000 °C tiene lugar la transición del IR cercano al rojo visible pero, si se continúa calentando el material, teóricamente debería de pasar por el resto de la escala cromática (lo que ocurre es que ningún material conocido aguanta tanta temperatura y se funde antes, por eso no lo vemos). En cambio, las estrellas con una elevada temperatura superficial – como el sol, que ronda los 6000 K – se ven amarillentas y las que poseen una mucho mayor aun (10000K como Sirio) se aprecian azules.*

Para obtener medidas más fiables con el método de los 2 colores es mejor elegir longitudes de onda que pertenezcan al rango del visible (de 400 a 750 nm). Esto se hace por varias razones: 1) Para evitar que, en la medida de lo posible, no interfieran con las radiaciones del H_2O, CO_2 y radicales OH. 2) La temperatura tiene una medida más fiable si se mide en el rango del visible y no es sensible al valor de α en este rango. 3) La influencia de los reflejos de la pared opuesta no es importante en el rango visible.

Como ventajas cabe citar la sencillez del concepto (no de la implementación de la técnica en sí, que no es fácil) y la facilidad del montaje (que además es barato, pues en esencia es el mismo que para NL). En cuanto a sus inconvenientes, se sabe que en motores Diésel existe una notable dispersión cíclica (variación en las condiciones termodinámicas de ciclo a ciclo) que puede afectar de manera significativa a la concentración de hollín cuantificada. Además, en este método se parte de la hipótesis de que la distribución del hollín (o fracción volumétrica) y su temperatura son uniformes a lo largo de todo el recorrido óptico[24], lo cual no es cierto rigurosamente, aunque constituye una aceptable aproximación.

2.2.4. Incandescencia Inducida por Láser

La técnica de incandescencia inducida por láser o LII es una útil herramienta de diagnóstico para medidas de alta resolución, tanto espacial como temporal, de hollín. Nos reporta información sobre su masa, su fracción volumétrica y su tamaño. El método se basa en el calentamiento del hollín contenido en un cierto volumen, hasta aproximadamente la temperatura de vaporización, por medio de un pulso láser de alta energía. Ésta es absorbida por la nube de hollín, que emite una radiación cercana a la de un cuerpo negro. Dicha radiación se registra con un sistema apropiado. Resolviendo las ecuaciones de balance de energía y de masa y, teniendo en cuenta los fenómenos relevantes en la absorción de la energía y las pérdidas de calor, es posible determinar la temperatura y el diámetro del hollín en cualquier instante de tiempo.

De este modo se pueden estudiar las relaciones entre la señal LII, la concentración volumétrica de hollín y su tamaño considerando la Ley de Planck para un cuerpo negro – con algunas simplificaciones (por ejemplo que el hollín es esférico) – y también considerando que el hollín absorbe la suficiente energía como para elevar al máximo su temperatura. La relación fundamental de la señal LII para la medida de concentración de hollín viene dada por la expresión (2.2), en la que f_V es la fracción volumétrica de hollín, C_n es su densidad y d_h es el diámetro medio:

$$S_{LII} \propto f_V \approx C_n \cdot d_h^3 \quad (2.2)$$

[24] *El factor KL que aparece en la ecuación (2.1) se suele emplear como variable para definir la cantidad de hollín en la muestra, ya que es un factor proporcional a la fracción volumétrica de hollín y al camino óptico recorrido por la radiación, L.*

La distribución de tamaños del hollín se infiere a partir del comportamiento temporal del enfriamiento. Requiere una temperatura de referencia que en principio es desconocida, pero es estimable, por ejemplo, haciendo uso del Método de 2 Colores visto anteriormente. La determinación de la concentración de hollín sale directamente de aplicar la ecuación (2.2) una vez conocido el diámetro medio de las partículas de hollín. La Figura 2.10 muestra cómo realizar una interpolación para adquirir el valor del d_h necesario **[15]** a partir del ensayo de enfriamiento. Se asume que el hollín se compone de partículas primarias.

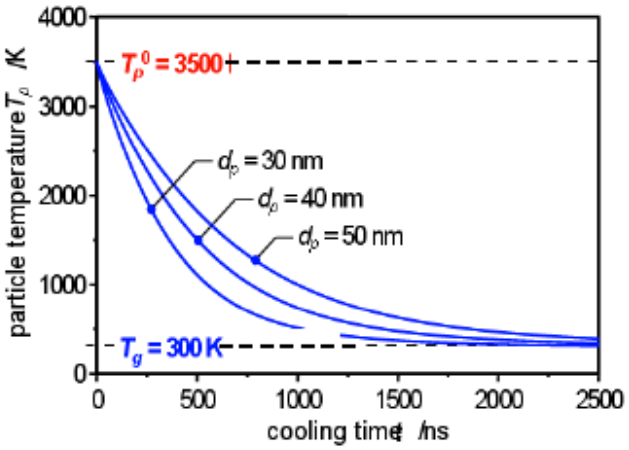


Figura 2.10. Tiempo de enfriamiento para diferentes partículas .

2.2.5. Medida del Radical OH*

El radical OH* es una especie química intermedia que se caracteriza por aparecer en los albores de la combustión cuando la llama aún no es visible, siendo por tanto un precursor de la combustión y poniendo de manifiesto la importancia de poder disponer de este tipo de técnicas que permiten seguir el transcurso de la combustión cuando todavía ésta no arroja signos de vida a nivel sensorial humano. El OH* denota la verdadera posición de una *cool flame*. Su detección se verifica mediante el uso de cámaras intensificadas ICCD (en el UV).

Cuando se toma una imagen con NL, se recoge la emisión de llama en el espectro al completo[25], sin necesidad de seleccionar ninguna longitud de onda en particular. Sin embargo, es sabido que la molécula del OH* solo emite en una franja espectral, centrada en *310nm*[25]. Se requiere del uso de un filtro en las cámaras para que éstas puedan captar únicamente la franja espectral de interés, evitando interferencias y visualizando solo así el OH* deseado.

$$1.\ CH + O_2 \rightarrow CO + OH^*$$

$$2.\ OH^* \rightarrow OH + h\nu$$

[25] *Esto ocurre porque, al tratarse de radiación térmica, por la Ley de Planck (Figura 2.9) la intensidad radiativa es, aun cuando pueda ser muy pequeña e imperceptible, existente para todas las longitudes de onda del espectro (como se puede intuir al examinar la rama asintótica derecha). En cambio, cuando se excita al OH*, éste emitirá a una única λ por asociarse a la energía del fotón que emite al desexcitarse y que, de acuerdo con la Mecánica Cuántica, es una concreta y no otra.*

En la Figura 2.11 puede apreciarse el aspecto de los fotogramas captados durante la combustión de un chorro Diésel en los que se evidencia la presencia de partículas de hollín, por las técnicas NL y LII, así como otra captura mostrando la presencia de radicales OH*. Puede verse como, en éste último caso, efectivamente se aprecia el inicio de su aparición cuando en el instante de tiempo 1140 microsegundos todavía no se percibe una combustión a simple vista.

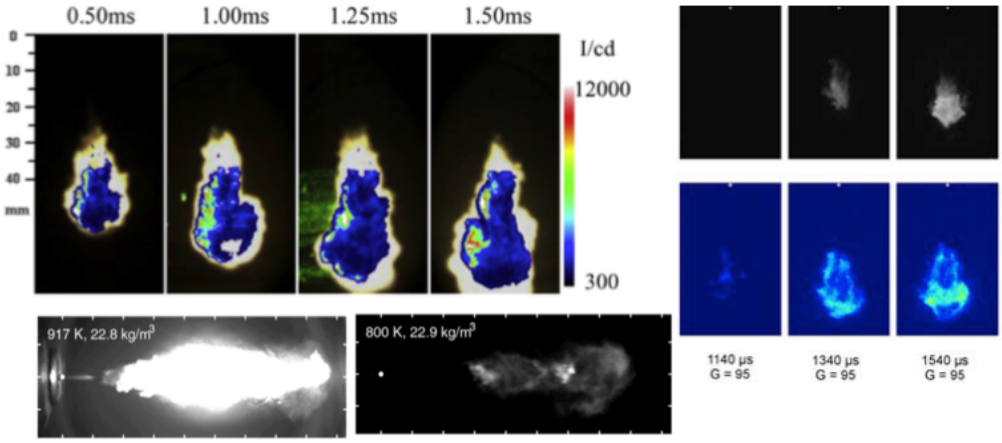


Figura 2.11. Capturas de imagen de la combustión en un chorro Diésel. A) Parte superior izquierda: LII. B) Parte superior derecha: OH. C) Parte inferior izquierda (dos imágenes): NL.*

2.2.6. Light Extinction Imaging (LEI)

2.2.6.1. Fundamento Teórico

Contrariamente a lo que ocurría en LII, en la técnica de *Extinción de Luz* no se produce un intercambio energético entre el haz de luz procedente de la fuente láser externa (u otro tipo de fuente) y las partículas de hollín. En su lugar, el método se basa en que, si un rayo de luz atraviesa la nube de hollín, su intensidad (o su potencia) se verá atenuada, por lo que comúnmente suele decirse que la presencia de hollín extingue la luz incidente. Si se profundiza en el fenómeno físico que da explicación a esto se tiene que, tanto la atenuación como una hipotética extinción, se deben a una combinación de los fenómenos de dispersión y absorción de luz por parte de las moléculas de hollín, tal como se representa en la Figura 2.12.

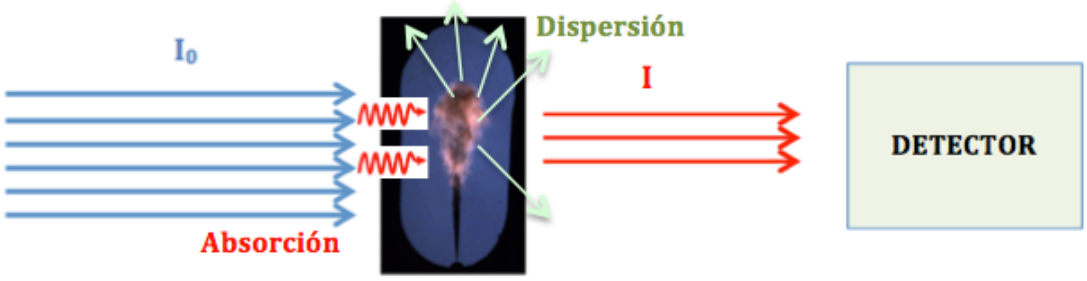


Figura 2.12. Esquema ilustrativo de los procesos físicos que tienen lugar al incidir el haz de luz láser de la fuente externa en las partículas de hollín.

Matemáticamente es muy sencillo cuantificar la cantidad de hollín haciendo uso de la conocida *Ley de Lambert – Beer*, que viene definida por la expresión (2.3):

$$\frac{I}{I_0} = e^{-KL} \quad (2.3)$$

Donde I_0 (W/m^2) es la intensidad de la luz que incide sobre la muestra, I (W/m^2) la intensidad de luz que la abandona una vez la ha atravesado, K es el *coeficiente de extinción* para una nube de hollín (m^{-1}) y L el *recorrido óptico* que efectúa la radiación (m)[26].

Es importante tener en cuenta que una limitación del método se encuentra en el parámetro L ya que, si éste es demasiado elevado (lo que equivale a decir que el espesor de la nube de hollín es demasiado grande) la extinción de luz es muy notoria y no es posible efectuar medidas adecuadamente. Siebers y Pickett **[16]** consideraban que si el factor KL supera un valor de 4 no se pueden obtener medidas fiables, pero con LEI ya se ha hecho.

La fracción volumétrica de hollín f_v puede determinarse en esta técnica a partir del cálculo del coeficiente de extinción con ayuda de la ecuación (2.3), pues I_0 es conocida, I es medida y L es el camino óptico que sigue el haz, que también puede ser determinado. La expresión para hallar dicha fracción viene dada por (2.4) y ha sido obtenida de relaciones derivadas de la *teoría de pequeñas partículas Mie* [27]:

$$f_V = \frac{K \cdot \lambda}{k_e} \quad (2.4)$$

λ es la longitud de onda del LED que se emplea (conocida, suele ser a 410nm) y k_e corresponde al *coeficiente óptico de extinción*, que puede determinarse mediante la ecuación (2.5), siendo ésta una expresión general en la que el término en azul tiene en cuenta el efecto de la dispersión en la extinción – al considerar que el hollín crece y forma aglomerados **[17]** – a través del factor de corrección α, mientras que el término en negro se asocia al proceso de absorción. Por su parte m corresponde al *índice de refracción del hollín*, que se puede obtener de manera aproximada a partir de la longitud de onda del láser utilizado, aunque es conveniente ser cuidadosos debido a que puede introducir un error bastante considerable.

$$k_e = (1 + \alpha) \cdot 6\pi m I \frac{(m^2 - 1)}{(m^2 + 2)} \quad (2.5)$$

Si se toma la simplificación de asumir como primarias esféricas las partículas de hollín, es posible anular el factor α anterior y considerar que el proceso de extinción es debido exclusivamente a los procesos de absorción.

---

[26] *Notar que existe una consistencia dimensional en las unidades.*

[27] *Se trata de una teoría, cuyo nombre fue introducido en honor al físico Gustav Mie, y que constituye una solución analítica a las ecuaciones de Maxwell para la dispersión de radiación electromagnética por parte de partículas de geometría esférica.*

2.2.6.2. Aplicaciones prácticas

Cuando se trabaja como una llama hay que tener en cuenta el aporte de intensidad que ésta genera al captar las imágenes. Por tanto se debe conocer esta intensidad de la llama para poder desestimarla. Una buena forma de cuantificarla es realizar una medida sin ningún aporte de luz externo, pues de esta forma toda la intensidad capturada es la propia de la llama. Por otra parte la técnica requiere de medidas con una fuente de luz para medir la extinción del hollín, es decir requerimos de datos adquiridos con y sin luz, algo que es imposible conseguir simultáneamente. La solución adoptada es pulsar la fuente de luz a la mitad de la frecuencia de adquisición de la cámara, para lo que se requiere una fuente luz LED.

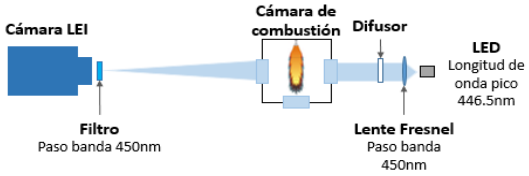


Figura 2.13. Montaje experimental de extinción de luz. La señal se registra con fotodiodo.

Para realizar las medidas de extinción de hollín se utiliza por tanto una cámara rápida, que permita ver la evolución de la llama en diversos instantes y también un LED como fuente de luz, que permita un encendido y apagado rápido para seguir de este modo el ritmo marcado por la cámara. Este montaje se prepara como se detalla en la Figura 4.2, teniendo la cámara y el LED en ventanas opuestas.

El montaje puede ser divido en dos partes, una de iluminación y transmisión de luz y otra de recolección y adquisición de luz. En cuanto a la parte de iluminación, se utiliza como fuente de luz un LED de alta intensidad con una longitud de onda pico de 446.5nm, que es colimada mediante una lente de 7mm de distancia focal. Seguidamente el difusor genera un campo circular de intensidad Lambertina de 100mm. Una luz difusa permite evitar efectos de *beam steering* (desviación de haz) que generaría una imagen con pequeños efectos de *Shadowgrapgy*, en la que las zonas negras podrían deberse a tal efecto. Respecto a la parte de adquisición de datos, la luz que ha atravesado la cámara de combustión es recogida, correctamente focalizada, en la cámara LEI, habiendo pasado antes por un filtro con un paso de banda de 450nm

Aplicaciones haciendo uso de esta técnica para llevar a cabo estudios sobre la presencia de hollín en chorros Diésel, así como cuantificaciones, se pueden encontrar muchas en referencias acudiendo a la literatura. Así, por ejemplo, J. V. Pastor Et al. **[18]**, propone un *paper* en el que se realiza un estudio comparativo entre el Método de 2 Colores y la Extinción de Luz. Se basa en dilucidar cuál de las dos técnicas es más sensible a la medida de Hollín, resultando ser ésta última finalmente.

La razón por la que se introduce esta reseña aquí es con el fin de ilustrar cómo es el aspecto de las imágenes que los algoritmos de procesado en *MatLab* arrojan – de los que se va

a empezar a hablar a partir del siguiente capítulo – pues cabe recordar una vez más y, llegados a este punto, que trabajan con imágenes que las cámaras rápidas capturan durante el experimento en montajes empleando la técnica LEI.

En el caso que nos ocupa, la Figura 2.14 muestra el *Setup* experimental, que se divide en dos partes. Por un lado se tiene la fuente externa, que en este caso concreto es un LED azul. El haz se hace diverger para, a continuación, hacerlo pasar a través de la lente que amplificará su intensidad. En lo que respecta a la parte colectora, la luz transmitida atraviesa una lente esférica (cuyo propósito es el de asegurar que el tamaño de las imágenes enfocadas posteriormente en las lentes de la cámara es menor que el de dichas lentes). Finalmente, el haz es recogido por la cámara rápida. En la misma figura puede verse el aspecto que presenta una de estas imágenes 2D, tomadas en un instante dado de la combustión de un chorro Diésel, junto a la concentración de hollín que se determina a lo largo de la dirección axial de éste. La línea verde (KL medido) se encuentra en todo momento por debajo de la roja (línea de saturación) y puede constatarse que, en ese instante de tiempo de avance de la combustión, los mayores niveles de hollín se han generado en el tramo líquido del chorro y en la llama de difusión respectivamente.

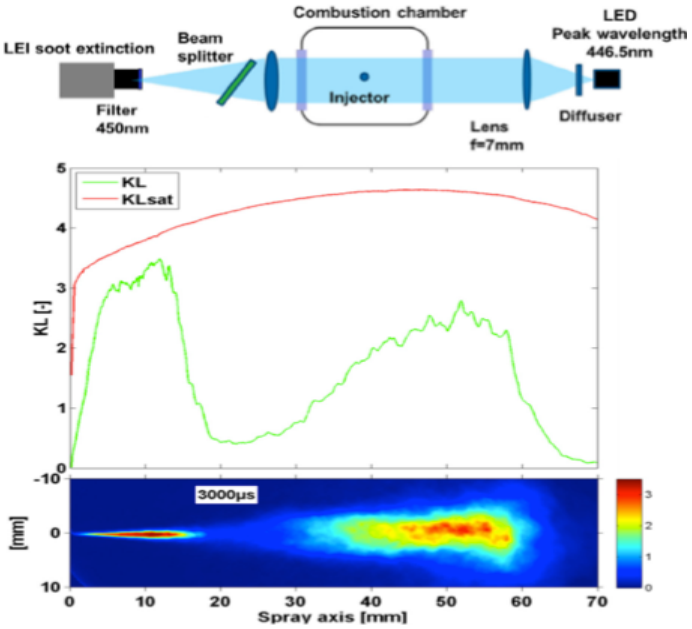


*Figura 2.14.a) Montaje experimental **[18]** para la medida y cuantificación del Hollín empleando la técnica óptica LEI. b) Extracto de imagen del chorro Diésel tras procesado con algoritmo.*

3. MATERIALES Y MÉTODOS

3.1. Introducción

Este capítulo está destinado a introducir datos, descripciones, características y modos de empleo de las distintas herramientas, tanto prácticas como teóricas, que se han empleado a lo largo del desarrollo del proyecto. Dado que éste es de carácter particularmente informático, se procederá a describir tanto el hardware como el software empleados para trabajar con los algoritmos de procesado de imágenes que habitualmente utiliza el grupo de Técnicas Ópticas del CMT–*Motores Térmicos* para sus ensayos de combustión en chorros Diésel.

Cabe destacar que, si bien ha sido necesario establecer en capítulos anteriores un marco teórico y un contexto a través de la introducción del fundamento de este tipo de combustiones, así como de las técnicas ópticas empleadas a fin de llevar a cabo los ensayos, es la búsqueda hacia la optimización de los códigos base que se emplean ya habitualmente lo que motiva y centra este trabajo. Si la primera parte de un ensayo corresponde al montaje, ajuste, calibración de equipos y toma de medidas – haciendo uso de las técnicas ópticas – la segunda y, no menos importante, es la tarea de procesar las imágenes ya registradas, con el fin de llevar a cabo acciones como las de segmentar la imagen para evaluar zonas de interés, medir diferentes parámetros relevantes en cada estudio[28] o la de programar líneas de código adicionales que eviten errores presentes en las imágenes o, incluso, el procesar más de una vez un mismo caso. Todo ello acaba afectando en última instancia al tiempo de cálculo, que es el parámetro en el que este proyecto se ha enfocado para tratar de reducirlo al mínimo posible. A este fin se aplican herramientas de paralelizado, mediante el uso de la tarjeta gráfica instalada en el ordenador de trabajo.

En primer lugar se describirá el PC en el que se ha trabajado, haciendo especial hincapié en las especificaciones técnicas, junto a las de su gráfica. En este punto, será interesante plantear una base teórica acerca de lo que trata la denominada Arquitectura CUDA – exclusiva de las tarjetas *NVidia GeForce* – y que permite exprimir todo el potencial de las mismas para agilizar cálculos a la hora de programar.

Finalmente se dará paso a la descripción de los tres niveles de paralelizado[29] en los que se suele trabajar – que serán introducidos en orden de complejidad y profundidad de programación – finalizando con la ilustración de algunos ejemplos. Se dejará la parte metodológica de ensayo enteramente para el Capítulo 4.

---

[28] *Como la concentración de hollín en llama, la penetración del chorro, la medida de LOL…etc.*

[29] *En este trabajo se han llevado a cabo paralelizados empleando los niveles 1 y 2. El tercer nivel requiere de programación a más bajo nivel empleando código C para poder acceder a los hilos de la GPU individualmente. Se considera un nivel de paralelizado experto y enfocado ya más al ámbito informático, de Sistemas y Automática. Sin perjuicio de ello, se incluirá en el Capítulo 6 un sub-apartado en el que se proporcionará una introducción, a modo guía, acerca de las pautas que se habrían de seguir si se optase por sumergir en este tipo de paralelización más profunda ya en trabajos futuros.*

3.2. Arquitectura GPU

Tal y como se ha apuntado, todo el material empleado para la realización del proyecto ha consistido en un ordenador personal equipado con una tarjeta *NVIDIA GeForce GTX 960*, dotada de arquitectura CUDA. En cuanto a los métodos, únicamente se ha requerido el uso de *MatLab 2017*, junto a sus librerías específicas para programación en paralelo con la GPU.

El grupo de Técnicas Ópticas del CMT – *Motores Térmicos* ha ido desarrollando códigos a lo largo de los últimos años – implementados en MatLab – que cargan ficheros con imágenes, previamente guardadas en discos duros procedentes de diversas cámaras rápidas, y las procesan para obtener los parámetros relevantes de la combustión de un chorro Diésel que más interesan, según el estudio que se desea realizar. En particular y, como se ha resaltado en repetidas ocasiones a lo largo de este documento, las imágenes que procesan los algoritmos empleados en el proyecto han sido adquiridas haciendo uso de la técnica óptica LEI, y registradas por una cámara CMOS Photron SA-5.

La concreción del material – aportando especificaciones técnicas – así como de los métodos utilizados, se deja para sub-apartados siguientes, si bien antes merece la pena introducir al lector en lo que se hace llamar desde hace pocos años como arquitectura CUDA, ya que su conocimiento es de vital importancia para entender adecuadamente los procesos de paralelizado que se han llevado a cabo a lo largo del proyecto a que hace alusión la presente memoria.

3.2.1. Arquitectura CUDA

CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) y se corresponde con una tecnología asociada a tarjetas gráficas de la casa NVidia, que fue lanzada a mediados del 2007 y que funciona únicamente para los modelos G8X en adelante. Intenta explotar las ventajas de las GPU frente a las CPU de uso general. En la Figura 3.1 se incluye una tabla de hitos alcanzados y perfil del usuario diana desde su lanzamiento hasta la actualidad. Cabe decir que la 6.0 es una versión que explota las posibilidades de programar con lenguajes de alto nivel como en *MatLab*.

Versión de CUDA [año]	Usuarios y rasgos más sobresalientes
1.0 [2007]	Muchos investigadores y algunos usuarios madrugadores.
2.0 [2008]	Científicos y aplicaciones para computación de altas prestaciones.
3.0 [2009]	Líderes en la innovación de aplicaciones.
4.0 [2011]	Adopción mucho más extensa de desarrolladores.
5.0 [2012]	Paralelismo dinámico, enlazado de objetos, Remote DMA.
6.0 [2014]	Memoria unificada CPU-GPU
Próximamente	Operandos de punto flotante de 16 bits (half)

Figura 3.1. Tabla que muestra como CUDA ha ido extendiéndose desde usuarios más especializados hasta ofrecer características atractivas para los más generalistas.

Antes de pasar a profundizar en detalle y responder a cuestiones del tipo *qué es*, *cómo funciona* y *cómo se puede aprovechar*, es necesario empezar por describir el funcionamiento de una tarjeta gráfica instalada en un ordenador ya que, en las aplicaciones gráficas para las que trabaja realizando cálculos, ésta ya realizaba intrínsecamente una ingente cantidad de procesos paralelos para conseguir mover objetos en pantalla con la fluidez con la que acostumbran antes de que se desarrollara la tecnología CUDA. En este sentido, no es exclusivo de NVidia el ofrecer al mercado GPUs que paralelizan procesos de cálculo, pero sí que puedan programarse de tal forma que lleven a cabo exactamente aquellos paralelizados que el usuario, con ciertas nociones de programación, desee, y poder así emplearlas en su beneficio.

En la Figura 3.2 se puede apreciar la estructura que propuso John Von Neumann en 1945 – en los albores de la informática doméstica – para instaurar la arquitectura más básica de un ordenador. Su propuesta constaba de cuatro pilares básicos: la memoria, la ALU, la unidad de control y los dispositivos de entrada/salida, que se reparten las tareas de almacenamiento, procesado e intercambio de información, siendo la unidad de control la que gestiona todas las actuaciones. En la misma figura se presenta, en la parte derecha, la arquitectura de un ordenador convencional en la actualidad. Existen principalmente dos partes: una conocida como *Host* y otra llamada *Device*, las cuales se asocian al concepto de maestro y esclavo. En el *host* se encuentra la CPU junto con la memoria principal, así como los dispositivos de E/S que permiten la comunicación con el *Device*. En el *Device* se encuentra la tarjeta gráfica junto con una memoria de video asociada.

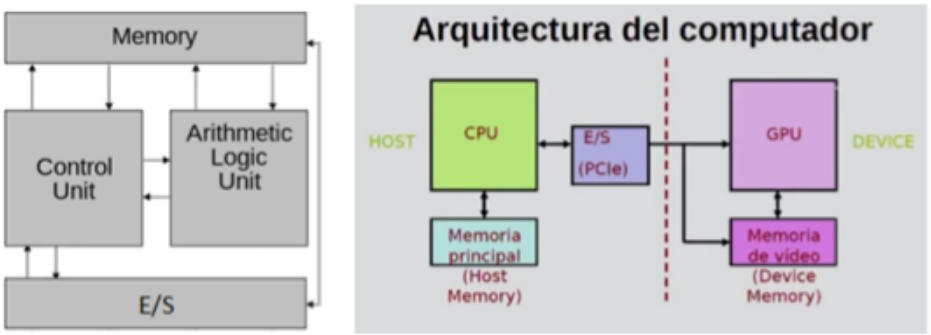


Figura 3.2. Estructuras de un ordenador con: a) Arquitectura Von Neumann. b) Arquitectura convencional actual.

En los primeros ordenadores la CPU se encargaba de procesar todo tipo de información. Sin embargo, conforme aumentaba su popularidad y surgían nuevas aplicaciones que exigían más recursos – como programas de edición gráfica o videojuegos – se hizo necesario el desarrollo de nuevos dispositivos que cumpliesen las nuevas exigencias de los usuarios. En ese contexto surge el nacimiento de la GPU tal y como la conocemos actualmente. Deriva de un componente llamado coprocesador matemático, dispositivo que se utilizaba para acelerar ciertas operaciones y procesamiento de datos, esto es, como segundo procesador. No fue hasta 1999 cuando también NVidia acuñó el término GPU, desde ese instante la mejora de rendimiento se ha ido incrementando gracias, en gran medida, al éxito en el campo recreativo de los videojuegos. En este paradigma, la GPU – dividida en unidades funcionales – estaba destinada a realizar las tareas de procesar píxeles y vértices. Por consiguiente, resolvía exclusivamente problemas gráficos.

Con todo, de manera tradicional los algoritmos se venían ejecutando sobre el Host, utilizando la CPU y siguiendo el modelo de ejecución secuencial. Las consecuencias de emplear este sistema cuando se manejaban grandes cantidades de información eran la ralentización y el aumento del tiempo necesario para obtener resultados. Hasta que en el 2007, la tecnología CUDA de NVidia mejora dichas funcionalidades, incorporando a cada tarjeta gráfica una ALU y una memoria que almacenara el *caché* [30] necesario para la ejecución de los programas, volviendo a reactivar el concepto de coprocesador matemático al fusionarlo con el de GPU. Así nacen las *GPUPU*, dispositivos capaces de ejecutar código que normalmente está destinado a ser ejecutado en la CPU. De esta forma, en la CPU (maestro) se ejecutará código que no puede o no necesita ser paralelizado, mientras que en la GPU (esclavo) se ejecutará código en múltiples unidades de procesamiento, permitiendo así alcanzar velocidades de cómputo muy superiores.

Es importante apuntar que, el hecho de que la arquitectura CUDA permita ejecutar código en GPU que en principio ejecutaría la CPU, puede inducir a pensar que ambas trabajan simultáneamente. Esto, en según qué aplicaciones o entornos de programación puede ser posible e incluso potencialmente útil, pero si se piensa en un script de *MatLab* por ejemplo, el código se debe de ejecutar siempre secuencialmente para poder funcionar bien. En tal caso, aparentemente puede parecer baladí el transferir – alcanzada cierta línea de código – parte de la ejecución del script que normalmente realiza la CPU, a la GPU. Pero si se piensa que en un PC convencional de 2, 4, 6 o incluso 8 núcleos, los cálculos se van a repartir entre éstos, se entenderá fácilmente por qué el uso de una potente tarjeta gráfica, como por ejemplo la NVidia GeForce GTX Titán Z – que contiene 5760 núcleos capaces de trabajar de manera simultánea – hace tan atractiva la migración de código de la CPU a la GPU. A este concepto de paralelizado[31] es al que se alude habitualmente cuando se piensa en el uso de la GPU en entornos de programación como MatLab, que constituye el marco en el que se ha desarrollado este trabajo. La Figura 3.3 esquematiza conceptualmente lo que se acaba de discutir[32], mientras que la Figura 3.4 resume el flujo de procesamiento que tiene lugar entre la CPU y la GPU.

---

[30] *En informática, la memoria caché es la memoria de acceso rápido de un microprocesador que guarda temporalmente los datos recientes de los procesados (información).*

[31] *A modo ilustrativo se puede pensar en un bucle "for" que no utiliza información de forma recursiva, esto es, que ha sido implementado de modo que, con cada avance del contador, se realiza un cálculo para el cual no es necesario conocer su valor en la vuelta anterior. En ese caso, cuando la CPU procesa el bucle, automáticamente reparte la carga de cálculo entre sus núcleos. Si en lugar de repartir entre 2, 4 u 8 se hace entre 5760, el tiempo de cálculo se reduce de manera ostensible.*

[32] *Como se verá más adelante, solicitar a la GPU que ejecute una parte del cálculo – lo que se conoce formalmente como migración – requiere insertar comandos que modifican ligeramente el código original o incluso sub-rutinas, pero en ocasiones con un 5 % de reescritura se consiguen reducciones del 50%.*

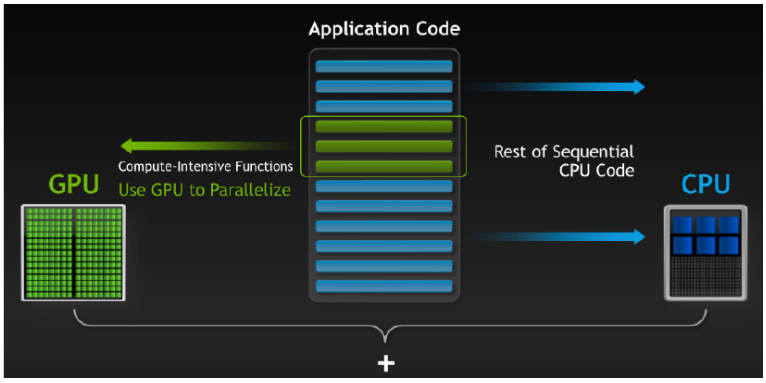


Figura 3.3. Esquema de programación heterogénea que muestra el tramo de código que migra a la GPU.

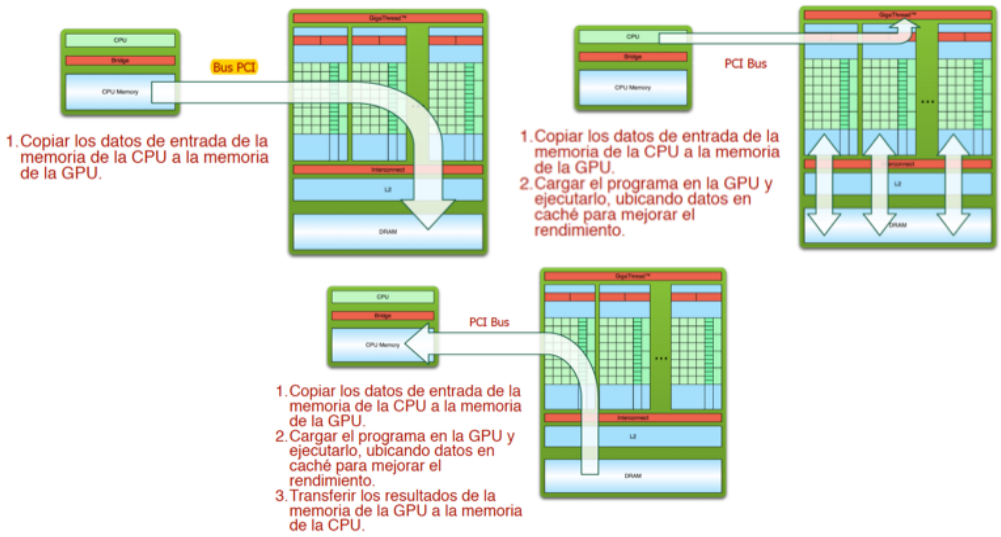


Figura 3.4. Esquema del flujo de procesamiento.

3.2.2. Jerarquía entre malla, bloques e hilos

Una vez introducido el fundamento de la GPU y lo que permite la arquitectura CUDA, es momento de entrar a describir cómo se organizan las tarjetas gráficas a nivel de hardware **[19].** Antes conviene familiarizarse con algunos elementos que irán apareciendo:

• **Dispositivo GPU:** - Conjunto con N multiprocesadores.

• **Multiprocesador:** - Conjunto de procesadores y memoria compartida. Cada uno contiene M núcleos.

• **Kernel:** - Programa listo para ser ejecutado en la GPU.

• **Malla (Grid):** - Conjunto de bloques cuyos hilos ejecutan un Kernel.

• **Bloque:** Grupo de hilos (threads) capaces de ejecutar un Kernel de manera independiente y que pueden comunicarse entre ellos a través de la memoria compartida del multiprocesador.

Dentro de la GPU podemos encontrar múltiples núcleos de procesamiento capaces de ejecutar programas de manera paralela. Dichos elementos se organizan siguiendo una jerarquización que facilita la programación y el correcto reparto de recursos. En la Figura 3.5, a simple vista se observan los dos elementos principales, el host y el device (representados ambos por dos rectángulos azules). El Device está dividido en mallas (verdes), como elemento que contiene a todos los demás. Cada malla dispone de distintos bloques y cada bloque está formado por un número de hilos determinado que ejecutan una función denominada *Kernel*.

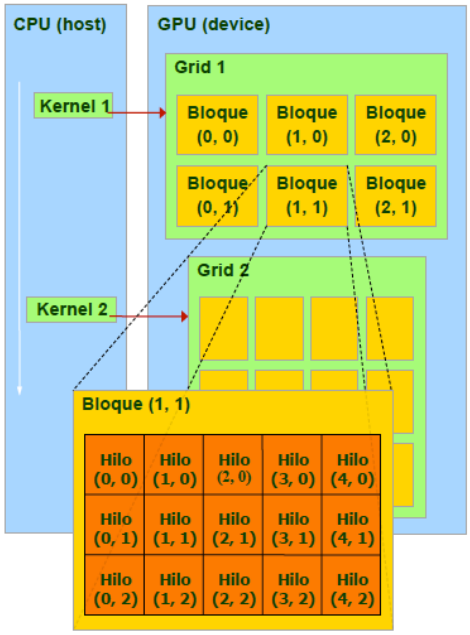


Figura 3.5. Ejecución de un Kernel en el modelo de programación CUDA .

El host es el encargado de configurar y ordenar la ejecución de los kernels, los cuales se ejecutarán en mallas diferentes. En el ejemplo que ilustra la figura, el Kernel 1 se ejecuta en la Malla 1. Ésta se encuentra formada por una matriz de bloques constituida por dos filas y tres columnas. Atendiendo al bloque correspondiente al elemento (1,1) se aprecia una estructura de hilos, de tres filas y cinco columnas. En este caso, cada hilo será el encargado de ejecutar el Kernel 1 de manera independiente. Ello permite que éste pueda ser ejecutado por un total de 90 hilos simultáneamente, lo que da una idea de la ventaja que puede suponer cuando el código asociado al Kernel, por sus características no recursivas, puede ser repartido entre los hilos en lugar de dejar toda la carga computacional en manos de la CPU [33].

Antes de entrar en materia acerca de cómo se organizan los hilos merece la pena recordar una vez más que, para paralelizar un tramo de código, es preciso que los cálculos no sean recursivos sino independientes, pues si bien cada hilo podrá acceder a la memoria de la CPU y rescatar un dato, el cálculo que vaya a realizar no debe depender en ningún caso del que está llevando a cabo su homólogo en el mismo instante.

---

[33] *El ejemplo ha sido expuesto a modo ilustrativo. Realmente es el programador quién debe declarar el número de hilos a los que encomendar la ejecución del kernel. El límite lo pone la memoria disponible en la GPU.*

La función, algoritmo o fracción de código que se desea ejecutar de forma masiva en el device se denomina, como se ha indicado, kernel. Éste se invoca desde la CPU, la cual delega el trabajo en la GPU, creando para ello un determinado número de hilos encargados de ejecutarlo de manera individual e independiente. Los datos sobre los cuales trabaja cada hilo pueden ser los mismos o distintos.

Por otro lado, cada hilo lleva asociado un número que le permite identificarse de manera unívoca, a la vez que también permite tomar decisiones de control o direccionar la memoria. El usuario es libre de declarar tantos hilos como desee (siempre y cuando no rebase la memoria disponible en GPU) y de organizarlos de tres formas distintas: como vector, como matriz bidimensional o como matriz tridimensional, según la aplicación a la que vayan destinados. Se puede ver en la Figura 3.6.

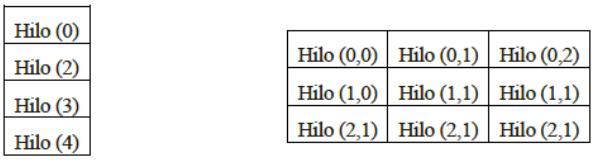


Figura 3.6. Organización de los hilos declarados. A la izquierda, 4 hilos declarados como vector. A la derecha, 9 hilos declarados como matriz bidimensional.

Así, por ejemplo, si el kernel contiene la instrucción de realizar una simple operación sobre cada elemento de un vector, tendrá sentido inicializar los hilos en forma de vector, ya que cada hilo podrá realizar individualmente esa operación en cada uno de los elementos que componen el vector. En cambio, dado que las imágenes son elementos 2D, lo razonable será inicializarlos como matriz (como es el caso de los algoritmos de procesado que se han empleado para este proyecto).

Una vez introducido el concepto de hilo cabe repasar algunas de las características que exhiben los bloques:

• Pueden contener un conjunto de hilos almacenados como vector, matriz 2D o matriz 3D.

• Una vez que se ha ejecutado un kernel ya no es posible modificar el tamaño del bloque.

• Todos los hilos de un bloque pueden cooperar entre sí, pero no con los de otro bloque.

• Los bloques se agrupan en mallas y también poseen un indicador bidimensional.

• El identificador de hilo es único en cada bloque. Esto quiere decir que hilos de dos bloques diferentes pueden tener el mismo identificador, pero al diferir en el identificador de bloque, para localizarlos se conserva la univocidad.

Para asignar el número identificador a cada hilo la expresión (3.1) muestra como procede la GPU:

$$int\ idx = (blockIdx.x \cdot blockDim.x) + threadIdx.x \quad (3.1)$$

Con objeto de ilustrar las características que se acaban de ir mencionando, además de ver cómo se emplea la expresión (3.1), se alude a la Figura 3.7. Se declara un bloque compuesto por 12 hilos en forma de vector, como se puede apreciar. Supóngase que, por el tipo de algoritmo, interesa realizar un paralelizado de grano fino[34] y se decide particionar los 12 hilos en tres bloques que contengan cuatro hilos cada uno. Puede verse como cada bloque y cada hilo tiene su propio número identificador, y como la expresión (3.1) permite identificar a cada hilo de forma unívoca con un número entero, evitando así indeterminaciones.

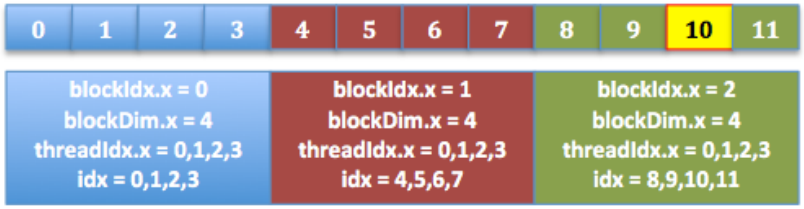


Figura 3.7. Declaración de 12 hilos en tres bloques y su etiquetado por parte de la GPU.

¿Cómo sabrá la GPU cuál es el décimo hilo cuando necesite hacer uso de él?

Cada bloque tiene su número identificador. Cada hilo dentro de cada bloque tiene el suyo también (aunque se repiten de bloque a bloque), y se conoce la dimensión de cada bloque (4 en este caso). La GPU no tiene más que aplicar la ecuación (3.1) para reconocer qué hilo es el que necesita coger cuando le llegue la instrucción:

$$10 = 2 \cdot 4 + threadIdx.x \ ;$$

$$threadIdx.x = \mathbf{2}$$

Por tanto, para seleccionar el hilo 10 lo que hará es acudir al bloque verde con identificador 2 y tomar el hilo cuyo identificador dentro de éste sea 2. El *blockIdx.x* la GPU lo obtiene de manera inequívoca porque se tiene que satisfacer la ecuación y *threadIdx.x* ≤ 3 (si el identificador de bloque fuera cero ó uno la suma no podría llegar al valor de diez).

Es interesante – para comprender las diferencias que existen a la hora de ejecutar un algoritmo empleando solo la CPU o migrando una parte del código a la GPU – analizar cómo realiza las tareas de paralelizado de grano fino la GPU, planteando un ejemplo sencillo como el siguiente: - Supóngase que en un determinado script[35] se crea un vector de N elementos con un comando que permite generarlos de manera aleatoria (en MatLab sería el comando ***rand***). En tal caso, se tendría un vector de longitud N con cada posición ocupada por un número al azar. Sea el objetivo la lectura del número que contiene cada elemento del vector, de modo que si éste es par se sustituya por un cero y, si es impar, se deje como fue generado.

A continuación se muestra un cuadro que recopila las opciones de programado.

---

[34] *Un poco más adelante se explicará cuál es la diferencia sustancial entre la paralelización de grano fino y la de grano grueso. Anticipar únicamente que es un concepto relacionado con el control de los hilos.*

[35] *Por el momento se supondrá un lenguaje de programación neutro. Esto es, sin concretar el entorno con el que se lleva a cabo, bien sea en C, C++, Fortran, MatLab…etc.*

Algoritmo 1 (CPU)	Algoritmo 2 (CPU/GPU)	Algoritmo 3 (Kernel)
1: V = [1 5 3 5 2 4 5 2 8 6 …N]	1: V = [1 5 3 5 2 4 5 2 8 6 …N]	1: Obtener int idx en cada hilo
2: Bucle desde i = 1 hasta i = N	2: Ejecutar Kernel en GPU	2: Bucle desde i = 1 hasta i = N
si V_i es par	3: Recuperar los datos para la CPU	si V_i es par
$V_i = 0$		$V_i = 0$
3: Fin		3: Fin
4: Fin		4: Fin

Si se ejecuta el Algoritmo 1 a través de la CPU, será necesaria la inclusión de un bucle que permita realizar las citadas tareas para cada elemento del vector. Esto es, se necesitará que el contador vaya desde 1 (la posición del primer elemento del vector) hasta N (la posición que ocupa el último). Como se puede observar, el algoritmo es secuencial, pues la tarea de inspeccionar la naturaleza del número albergado en cada posición se realizará únicamente obedeciendo al valor del contador a cada vuelta del bucle. Esto propicia que haya que dar N vueltas, con N cálculos, de manera que no se procederá a la inspección y eventual sustitución, por ejemplo, del número contenido en el elemento 17, si antes no se ha hecho en el del elemento 16. Esto constituye un *cuello de botella*, especialmente si N muy elevado.

Si se ejecuta el Algoritmo 2, éste se inicia en la CPU, quién realizará el mismo cálculo en la primera línea que el asociado al Algoritmo 1. Pero en la segunda línea ahora llama al Algoritmo 3, esto es, al Kernel. Éste asigna un número entero como identificador para cada hilo que haya sido declarado por el programador (N hilos, N identificadores). El resto de líneas son exactamente las mismas que contenía el Algoritmo 1, solo que ahora ya no se van a ejecutar en la CPU, sino en la GPU. Cada hilo tendrá acceso al vector V, guardado en la memoria compartida en la GPU al realizar la migración. De este modo, la operación que tiene que realizar cada hilo es tan simple como analizar y ejecutar las operaciones sobre el elemento del vector V que corresponde a su identificador de hilo. Así, por ejemplo, el hilo ocho inspeccionará solo la octava posición del vector V, verá que el contenido es un dos y, dado que es par, lo sustituirá por un cero. El resultado será un nuevo vector V con ceros en las posiciones donde originalmente habían números pares y números aleatorios en las impares.

Este procedimiento de paralelización llevado a cabo en el Algoritmo 2, que para este ejemplo tonto puede parecer innecesario, se hace muy útil cuando N es un número especialmente elevado, pues se opera sobre los N elementos de manera simultánea, mientras que en el Algoritmo 1 la operación sobre cada elemento del vector requiere la espera de la terminación de la vuelta de bucle anterior.

3.3. Características técnicas del ordenador de procesado

A continuación se van a introducir las especificaciones técnicas del ordenador de sobremesa que se ha utilizado, así como de la tarjeta gráfica que constituye la piedra angular del proceso de paralelizado. Con esto se cierra la descripción de los materiales empleados y se deja para el último apartado de este capítulo la parte más asociada a los diferentes métodos de programación en paralelo que dan soporte al proyecto realizado.

En la Figura 3.8 se recogen las especificaciones técnicas del PC con el que se ha llevado a cabo todo el TFG. Se compone de un procesador con cuatro núcleos.

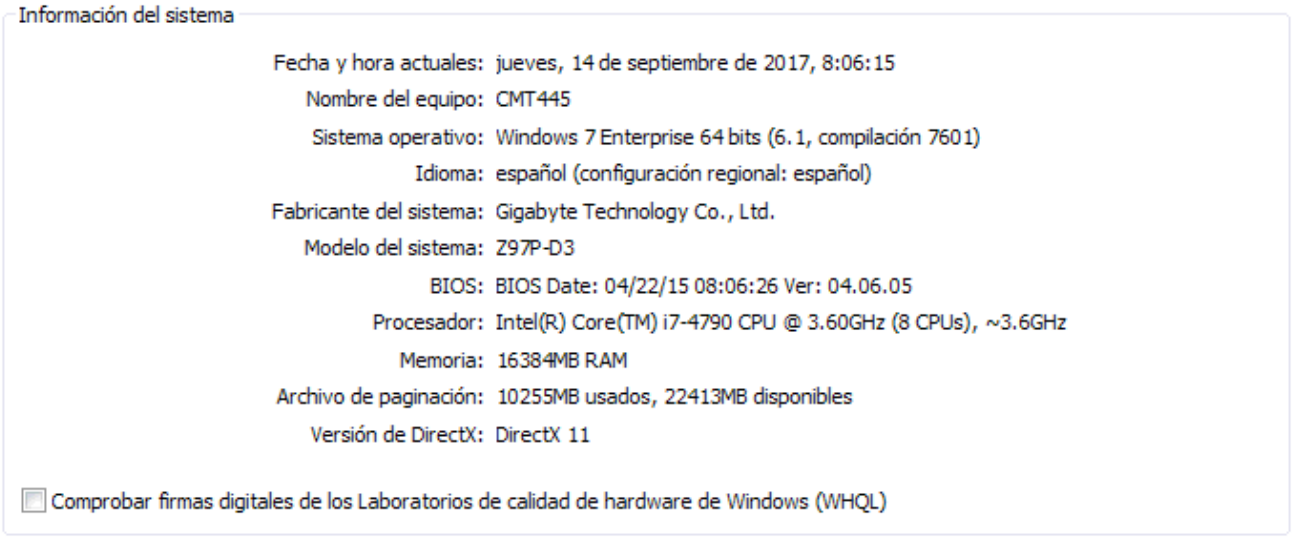


Figura 3.8. Características técnicas del ordenador de sobremesa utilizado para el TFG.

En la Tabla 3.1 se adjuntan las especificaciones de la ***GPU NVidia GeForce GTX 960***, que es una tarjeta gráfica de gama media enfocada al procesado de gráficos para ordenadores de sobremesa. Se basa en la llamada estructura *Maxwell*, tecnología usada en las gráficas más potentes del mercado actual.

El campo en el que más se usa este tipo de dispositivo, y para el cual está enfocado, es el de los videojuegos, debido al aumento de su popularidad y de la creciente complejidad que están adquiriendo. No obstante, gracias al desarrollo en este campo – y de acuerdo con lo que ya se ha visto – las tarjetas gráficas constituyen herramientas de paralelizado cada vez más potentes para la optimización de algoritmos, ofreciendo a usuarios cada vez menos especializados la oportunidad de hacer uso de la arquitectura CUDA en entornos de programación tan variopintos como pueden ser *C*, *C++*, *OpenCL*^{tm}, *Direct Compute*, *Fortran*, *MatLab* o *Java* y *Phyton*.

GeForce GTX 960			
Especificaciones de la GPU		**Especificaciones**	
Núcleos CUDA	1024	**Máxima resolución digital**	5120 x 3200
Frecuencia de reloj normal	1127 Mhz	**Máx. VGA resolución**	2048 x 1536
Frcuencia acelerada	1178 Mhz	**Conectividad multimedia**	Display Port. HDMI
Tasa de relleno de texturas	72 GifaTexels/s	**Multipantalla**	Sí
Especificaciones de la memoria		**HDCP**	Sí
Frecuencia de la memoria	7 Gbps	**Audio HDMI**	Internal
Cantidad de memoria	2 GB	**Potencia y Temperatura**	
Interfaz de memoria	128-bit GDDR5	**Temperatura máxima**	98 °C
Ancho de banda máximo	112 GB/s	**Consumo**	120 W
Características de la tarjeta		**Requisitos mín. de potencia**	400 W
Entorno de programación	CUDA	**Conexiones de alimentación**	6-pin
DirectX	12 API	**Dimensiones**	
OpenGL	4,4	**Altura**	11,16 cm
Soporte de bus	PCI-E 3.0	**Longitud**	24,13 cm

Tabla 3.1. Especificaciones técnicas de la GPU NVidia GeForce GTX 960.

Como recopilación breve de lo que una tarjeta gráfica con arquitectura CUDA ofrece actualmente, y de lo que aún no es capaz de ofrecer, se mencionan las ventajas y desventajas:

Ventajas

CUDA presenta ciertas ventajas sobre otros tipos de computación en GPU utilizando APIs gráficas:

- **Lecturas dispersas:** - Se puede consultar cualquier posición de memoria.
- **Memoria compartida: -** CUDA pone a disposición del programador un área de memoria de 16KB (o 48KB en la serie Fermi) que se compartirá entre hilos. Dado su tamaño y rapidez puede ser utilizada como caché.
- **Lecturas más rápidas de y hacia la GPU**.
- **Soporte para enteros y operadores a nivel de bit**.

Limitaciones

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable.
- No está soportado el renderizado de texturas.
- En precisión simple no soporta números desnormalizados o NaNs.
- Puede existir un cuello de botella entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.
- Los hilos de ejecución, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.

3.4. Niveles de programación en paralelo

En los tres apartados anteriores se ha hecho énfasis en describir el material empleado, así como en profundizar en los conceptos básicos que dan soporte a la tecnología de programación en paralelo haciendo uso de tarjetas gráficas. En este que ahora si inicia, la idea es la de empezar a ver la otra cara de la herramienta, esto es, describir cómo se programa un código o líneas de código – en el lenguaje de programación que corresponda – que se pretende sea objeto de paralelización. Si antes se ha hecho hincapié en el fundamento físico y arquitectura que hacía posible funcionar a la GPU, en esta parte interesa saber cómo implementar todo eso en código para sacar provecho de la agilidad de cálculo que proveen los hilos y permitan reducir todo lo posible el tiempo de cálculo para optimizar un algoritmo complejo.

La paralelización de cálculos se puede abordar en tres niveles de complejidad y profundidad de programado. Se empezará viendo una ligera forma de agilizar códigos empleando los núcleos de la CPU (nivel 1) para abordar después en mayor profundidad las posibilidades de la GPU programando particularmente en un entorno concreto que es MatLab (nivel 2), gracias a sus *toolbox* especializadas introducidas en las últimas versiones y mediante lenguaje en C (nivel 3), que hace gala de la máxima capacidad de exprimir la potencia de la GPU a través de lo que se conoce como paralelizado de grano fino.

3.4.1. Paralelización en CPU

Paralelizar con la CPU es muy simple, aunque como se verá a continuación, constituye un método muy limitado. Se basa en el reparto de la carga de cálculo entre los distintos núcleos que tiene insertados el procesador. Todo lo que se requiere es un ordenador, el entorno de programación *MatLab* y disponer de una versión que tenga equipado el *Parallel Computing Toolbox*[36], en aras de aprovechar los distintos núcleos que pueda tener un PC, dando lugar así la ejecución de scripts mucho más rápida y eficientemente.

Se recomienda el visionado de tutoriales y webinars para recibir lecciones acerca de este tipo de paralelización tan simple con MatLab, especialmente para explorar las posibilidades que ofrece, conocer sus limitaciones y adquirir algunas nociones que pueden ser de utilidad para sortearlas en algunos casos concretos. De entre las recomendaciones que se sugieren a nivel de hardware, para sacar todo el provecho posible se necesita como mínimo un PC con dos núcleos y 2 GB de RAM. Como ya se dejó patente en la Figura 3.8, el ordenador empleado para este proyecto cumple con creces los prerrequisitos mínimos.

---

[36] *Se verá en el próximo sub-apartado que, para paralelizar con la GPU, el Parallel Computing Toolbox sigue siendo la herramienta a emplear cuando se trabaja con MatLab.*

En páginas anteriores se ha visto como, de forma intuitiva, resultaba bastante lógico pensar que el concepto de paralelizado guardara una tan estrecha relación con el de bucle. Esto ocurre porque, en entornos como MatLab, la ejecución de un determinado programa que lleve a cabo las acciones que interesan al usuario para su aplicación particular requiere de una secuencia de pasos, no siendo posible dar saltos entre líneas para retomarlas después [37]. Dicho de otro modo (para que no queden lagunas en este sentido), lo que no es posible es solicitar a la CPU que vaya ejecutando un tramo del código mientras se prosigue con la ejecución de las líneas siguientes y recibir, transcurrido un tiempo, el resultado arrojado por dicho tramo. Ese es el tipo de paralelización que, al menos en MatLab, no es posible realizar.

En esta línea de pensamiento, como se ha dicho, el carácter secuencial con el que se trabaja en MatLab abre las puertas a que las sentencias ***for***, ***while***, ***do while***…etc, constituyentes de bucle, generen cuellos de botella con más facilidad que líneas de código convencionales en las que se realizan asignaciones, se generan matrices o vectores o en las que se llevan a cabo cálculos. Es más, combinar bucles con operaciones complejas que trabajan con matrices muy pesadas de datos es sinónimo de ralentización del código. El número de vueltas será entonces un factor que escalará ese tiempo de cálculo, multiplicándolo.

Con esta problemática, en MatLab la solución se ha hallado de la manera más sencilla que cabría esperar. Solo es necesario escribir el comando ***parfor*** en los bucles ***for*** en los que se desee paralelizar y, automáticamente, el entorno mandará una orden interna a la CPU para que reparta el cálculo que está realizando entre el total de sus núcleos. Cuando se invoca a un simple bucle *for*, no se puede asegurar que, en general, la CPU esté solicitando ayuda de más de un núcleo (de ahí la utilidad del *parfor*)

Lo que el programa hace cuando llega a la línea de código donde se encuentra la sentencia *parfor* es abrir una herramienta denominada *Parallel Pool*, que es la que de forma autónoma gestiona el reparto de la carga computacional entre los núcleos. Esto se puede apreciar en la Figura 3.9. Mientras que lo habitual es visualizar la barra de carga en color azul – símbolo de que es la CPU convencional la que está trabajando – cuando se emplea el comando parfor, automáticamente se vuelve verde durante el transcurso de la ejecución del bucle. Cuando se ha inicializado, en consola debe aparecer el número de núcleos que usará la CPU para realizar el cálculo en paralelo, que coincidirá obviamente con los que tiene el procesador.

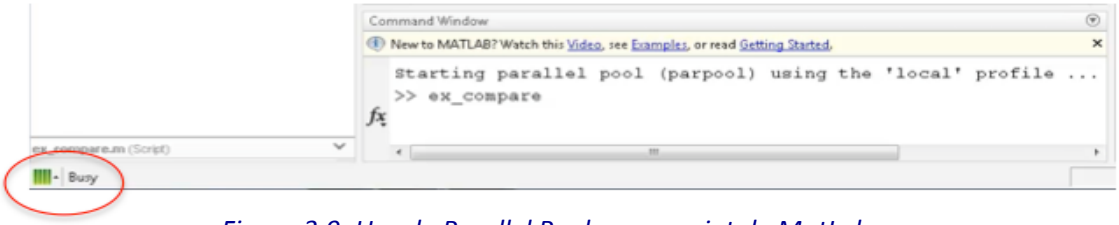


Figura 3.9. Uso de Parallel Pool en un script de MatLab.

[37] *No confundir con las sentencias condicionales **if**, **else**, **elseif** que denotan toma de decisiones y, con ellas, sí que es posible dar saltos de líneas. Si, además, se encuentran dentro de un bucle **for** o **while**, será posible retornar al punto del código donde se encuentran y que eventualmente es saltado.*

Otras maneras de inicializar el *Parallel Pool* manualmente es clickando directamente sobre la pestañita de la citada barra de carga (que habitualmente está en azul, aun cuando MatLab no está en ejecución). También es posible activarlo mediante el comando ***parpool***.

Como todo no puede ser tan bonito, emplear *parfor* (que, como se puede intuir, únicamente es válido para el tipo de bucle *for*) no siempre es posible, estando bastante limitado. De hecho, en ocasiones no funcionará debido a estas limitaciones. Por esta razón y, aunque no siempre serán subsanables, se ha creído interesante exponer este número de limitaciones persiguiendo dos objetivos. El primero, entender el mecanismo por el que funciona el comando *parfor*, ya que podría evitar pérdidas de tiempo innecesarias[38]. Y el segundo, analizar las opciones o "trucos" existentes para esquivar dichas limitaciones.

El problema de la recursividad es un mal endémico del propio concepto de paralelizado, tal y como ya se ha discutido en apartados anteriores de este mismo capítulo. Un ejemplo se puede ver en la Figura 3.10. El propio *MatLab* advierte de que el comando no podrá ser ejecutado, a través de un subrayado en rojo, junto con la variable responsable de ello.

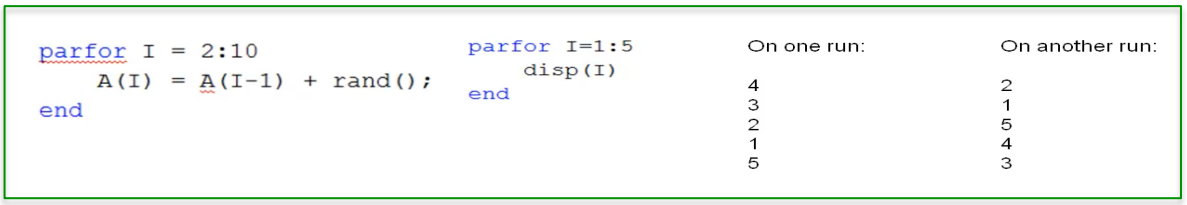


Figura 3.10. Mecanismo de funcionamiento de la instrucción parfor.

En este caso el script no se puede ejecutar, ya que la variable A requiere del conocimiento de su valor en la vuelta anterior. Esto quedará más claro si se entiende cómo funciona internamente el comando. Si se prueba a crear un sencillo bucle parfor, consistente en incluir el comando disp(I)[39], y siendo I un contador que vaya, por ejemplo, del 1 a 5, al ejecutarse se observará que cada vez salen en pantalla los números del 1 al 5 ordenados de distinta forma. Esto ocurre porque *parfor* calcula las iteraciones en cada vuelta con distinto orden[40].

En cambio, sí que existen algunas estructuras en las que, aunque MatLab nos indique que hay un problema, pueden resolverse mediante sencillos trucos. Se exponen a continuación:

---

[38] *Es posible encontrarse con bucles en los que, por su estructura de programado, directamente no sea posible implementar un parfor. Bien por incompatibilidad o porque requiera de una reescritura demasiado profunda, consumiendo tiempo y con riesgo de que el bucle pierda la funcionalidad para la que fue concebido. Como se verá, en este trabajo este tipo de paralelizado ha tenido escaso éxito.*

[39] *El comando disp() saca en consola el valor de la variable que tenga dentro del paréntesis.*

[40] *En este script parfor funciona muy bien porque a MatLab se le está solicitando que saque en pantalla números del 1 al 5. Y lo hace cada vez de un modo distinto, ya que internamente reparte el cálculo de forma aleatoria e independiente entre los núcleos del procesador. Si tiene 2 núcleos, por ejemplo, puede que 4 vueltas del parfor las esté haciendo uno de ellos y la que queda el otro. O tres y dos…etc.*

1) Bucles cuyo contador no es un número entero

```
parfor x = 0:0.1:1
```
⚠ The range of a PARFOR statement must be increasing consecutive integers. Details
```
end
```

```
xAll = 0:0.1:1;
parfor ii = 1:length(xAll)
       x = xAll(ii);
       Iterations depending on x
end
```

En este caso, el cuadro de advertencia no subraya de rojo el *parfor*. Pero esto se debe a que dentro de él no se halla ninguna variable que, por tanto, no da problemas. Pero igualmente, si se trata de ejecutar, MatLab dará error. La situación se solventa como se aprecia a la derecha: basta sacar fuera del bucle lo que antes se había incluido como contador de vueltas. Fuera del bucle ya no importa si no es entero. Al ejecutar esa línea generará un vector de números entre 0 y 1 con incrementos de 0,1. Ahora el *parfor* tiene de contador a un número entero pues, si se observa, éste va desde 1 hasta la longitud del vector xAll, que es 11. Por tanto, dará once vueltas, en las que x irá generando un vector con los valores que contiene el vector xAll situado fuera del bucle (pero no serán generados en orden natural).

2) Bucles anidados

```
parfor x = 1:5
    parfor y = 1:10
```
⊘ PARFOR or SPMD cannot be used inside another PARFOR loop.
```
    end
end
```

```
parfor x = 1:5
       myprog(x)
end

function myprog(x)
parfor y = 1:10
       ...
end
```

Al anidar dos sentencias *parfor* MatLab nos da un error, aunque no expresa con claridad cuál es el motivo exacto de no poder hacerlo. La solución, como se puede ver, es muy sencilla, basta con sacar uno de los bucles y declararlo como función. El otro, al ejecutar la invocará.

3) Convertir cuerpo del bucle en una función

```
data = rand(4,4);
means = zeros(1,4);
parfor I = 1:4
```
⊘ The PARFOR loop cannot run due to the way variable 'z' is used. Details
```
    means(I) = z.mean;
end
disp(means)
```

```
data = rand(4,4);
means = zeros(1,4);
parfor I = 1:4
    means(I) = computeMeans(data,I);
end
disp(means)

function meansI = computeMeans(data,I)
z.mean = mean(data(:,I));
meansI = z.mean;
```

A veces MatLab no sabe o no puede realizar el cálculo dentro del bucle por la forma en que una de las variables se usa. La solución es parecida a la anterior. Basta con crear una función que haga la media de los valores de cada columna de la matriz data, según el contador I tome del 1 a 4, y lo guarde en meansI. El bucle solo tiene que guardar el número contenido en meansI de cada cálculo e introducirlo en el vector de ceros de una fila y cuatro columnas.

Es importante recalcar que los casos ilustrados son objeto de cálculos muy sencillos. En este sentido, aunque valga para comprender en qué circunstancias será posible reescribir un bucle *parfor* para habilitarlo a nuestra conveniencia, en la inmensa mayoría de casos el problema con el que un usuario puede toparse será que, cuando se tienen algoritmos más o menos complejos, llevar a cabo los cambios que se sugieren en los casos 1, 2 y 3 puede ser una labor ardua, harto compleja e incluso antojarse impensable.

En general, cuando un nombre en un bucle *parfor* es reconocido como una referencia a una variable, se clasifica en una de las categorías que se muestran a continuación:

• **Variable "*Loop*":** - Sirve como índice del bucle para los arrays.

• **Variable "*Sliced*":** - Es un array cuyos segmentos son operados por diferentes iteraciones del bucle.

• **Variable "*Broadcast*":** - Es una variable definida antes del bucle, cuyo valor se utiliza dentro del mismo, pero no es asignado dentro del bucle.

• **Variable "*Reduction*":** - Es una variable que acumula un valor a través de las iteraciones del bucle, independientemente del orden de la iteración.

• **Variable "*Temporary*":** - Es una variable creada dentro del bucle pero, a diferencia de la variable del tipo "sliced", no estará disponible fuera del bucle.

Para que quede más claro, la Figura 3.11 muestra un ejemplo en un script:

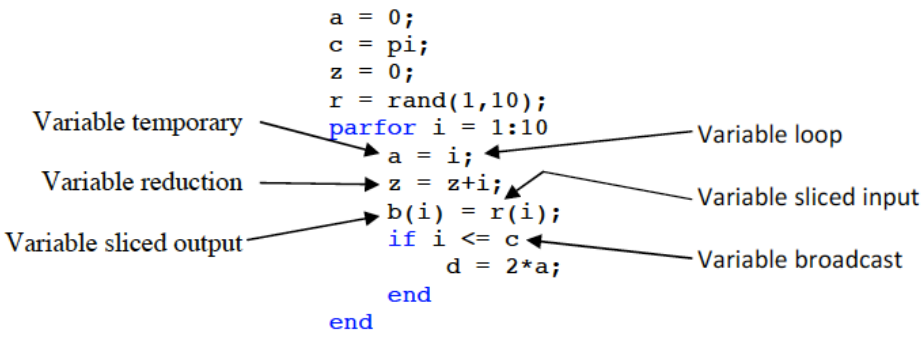


Figura 3.11. Tipos de variables en un bucle parfor.

Un bucle *parfor* generará un error si contiene cualquier variable que no puede ser categorizada únicamente en una clase de variable o si las variables violan sus restricciones de categoría al ser clasificadas. Por tanto, entender y asumir qué rol ejerce cada uno de estos tipos de variable en un bucle *for* ayudará a discernir si nuestro código a paralelizar cumple, en el tramo deseado, esta estructura, o a sopesar el esfuerzo que costaría reescribir las variables que generan problemas en una forma que sea aceptada por el analizador de código estático que tiene MatLab (el conocido cuadradito que está verde cuando el script no tiene fallos, naranja cuando ofrece sugerencias de mejora y rojo cuando hay error).

Dejando de lado los errores que puede dar el bucle *parfor*, antes de dar por cerrado el sub-apartado se citarán otra serie de limitaciones y que, esta vez, son intrínsecas al propio tipo de paralelizado (por tanto no atacables). Son las siguientes y rigen la velocidad de los bucles:

- **Paralelizar no acelera cuando el bucle es demasiado corto.**

- **Si la ejecución va más lenta que con un for convencional:** - Puede ser por limitación de RAM o de accesos a los archivos de variables (o datos) que están dentro del bucle.

- **Si el bucle está desbalanceado baja la velocidad:** - Hay que evitar que dentro del bucle existan cálculos muy cortos y otros excesivamente largos. MatLab lleva mal la gestión para priorizar núcleos.

Por otra parte, *MatLab* ya emplea hilos, de una forma similar a lo que hace la GPU, pero **solo** para ciertas operaciones y funciones preprogramadas **[20]**. Solo que la gestión la realiza de otro modo, más interno, sin la posible intervención del usuario. Por ello, se recomienda emplear el *Monitor de recursos* para examinar cómo evoluciona la carga de cómputo[41] .

Por último, dado que un PC de sobremesa rara vez sobrepasa los 8 núcleos, también es posible crear un clúster con N ordenadores y llevar a cabo paralelizados haciendo que cada ordenador aporte sus núcleos para realizar un mismo cálculo **[21]**. Con esto se consigue multiplicar enormemente el número de núcleos disponibles para procesar. En este caso se necesita la herramienta *MatLab Distributed Computing Server*.

Descrito ya todo el mecanismo de funcionamiento de la paralelización en CPU, así como sus posibilidades y limitaciones, concluir que el comando *parfor* resulta útil cuando se tienen iteraciones donde cada una de ellas tarda un gran tiempo en ejecutarse, ya que los "workers" (núcleos del procesador) pueden realizar estas iteraciones de forma simultánea. Cuando los bucles *for* en los que se implementa no cumplen esta condición, la comunicación entre núcleos consume más tiempo que el que lleva realizar las sencillas operaciones del bucle.[42]. Esto es precisamente lo que ha sucedido en el caso particular del algoritmo con el que se ha trabajado a lo largo del presente proyecto.

Se describirá y discutirá más adelante el cómo se trató de abordar inicialmente la paralelización del citado algoritmo mediante este método (Capítulo 4) y se analizarán posteriormente los resultados obtenidos (Capítulo 5).

---

[41] *Esto es muy interesante, puesto que podría estar tratándose de paralelizar un bucle que emplea funciones preprogramadas en MatLab que ya fueron implementadas para que, de forma automática, hagan uso de todos los núcleos del procesador. En tal caso, paralelizar no supone ninguna ventaja, de hecho enlentece. Para ver si un cálculo está haciendo uso de un porcentaje de cada núcleo próximo a cien, basta con abrir el monitor de recursos y examinar qué ocurre con las CPUs al ejecutar el bucle.*

[42] *Toda paralelización – y esto se verá que también ocurre con la GPU – lleva implícita una inevitable tarea de comunicación entre la CPU y los núcleos (o con la GPU al realizar la migración) que consume un cierto tiempo, debido a los paquetes de datos que se envían. Cuando el tamaño de los datos que se reciben son mayores (más pesados) que los que se envían, el paralelizado induce un enlentecimiento.*

Nota: - Si se tiene una matriz que es demasiado grande para la memoria de un ordenador dado, *Parallel Computing Toolbox* permite la distribución de la matriz entre varios núcleos, de modo que cada "worker" sólo contiene una parte de ésta, pudiéndose operar en todo el conjunto como una sola entidad. Cada "worker" sólo opera en su parte de la matriz, y de forma automática se transfieren datos entre sí cuando sea necesario, como por ejemplo, en la multiplicación de matrices. Se ha obviado su descripción debido a que ha dado problemas en el algoritmo de procesado de imágenes (produce bloqueos en MatLab).

3.4.2. Paralelización en GPU

3.4.2.1. En el entorno de programación de MatLab

Llevar a cabo cálculos paralelos con ayuda de la tarjeta gráfica en el entorno de programación de MatLab es lo que en este trabajo se ha hecho llamar paralelizado de nivel 2. Ello se debe a que, si bien continúa empleándose un lenguaje de programación de alto nivel, tan intuitivo y común en el ámbito ingenieril, el esfuerzo y dedicación que conlleva implementarlo bien lo hacen significativamente más complejo.

En el apartado 3.3 ya se introdujo cómo funciona una GPU basada en la arquitectura CUDA a nivel de hardware. Ahora es momento de profundizar en describir cómo un usuario puede emplear las herramientas, que MatLab ofrece en su *Parallel Computing Toolbox*, para paralelizar la parte (o partes) de su código que desea ejecutar más rápidamente, reduciendo al máximo el tiempo de cálculo y coste computacional global de su algoritmo. La construcción de un código CUDA requiere seguir los siguientes pasos **[23]:**

1. Identificar las partes con mayor potencial para beneficiarse del paralelismo.
2. Acotar el volumen de datos necesario para realizar dichas computaciones.
3. Transferir los datos a la GPU.
4. Hacer la llamada al kernel.
5. Establecer las sincronizaciones entre la CPU y la GPU.
6. Transferir los resultados desde la GPU a la CPU.
7. Integrarlos en las estructuras de datos de la CPU.

Los pasos 3 a 7 forman parte de una nueva estructura en el código original. Esto quiere decir que, si no se solicitara paralelizado, serían muchas líneas que no habría que introducir en el script. Por tanto, se ha de estar seguro (y realizar los estudios convenientes para ello) de que la ventaja que se puede extraer es suficiente. Ejecutar un trabajo en la GPU supone transferir, desde la memoria principal del ordenador a la memoria principal del dispositivo, los datos que se van a procesar. El Kernel solo trabajará en la GPU, por lo que, una vez finalizada la ejecución, el proceso es el inverso. Debido al elevado coste de tiempo que suponen estas transferencias de datos entre el Host y el Device es recomendable (si no imperativo) que en el caso de usar varios Kernel seguidos se reutilice la información ya copiada en la GPU y se evite, al máximo, el numero de transferencias.

La regla de oro en el ámbito de la paralelización reza así: ***Paralelizar solo cuando es posible, necesario y los cálculos del código convencional requieren mucho tiempo***.

Es importante señalar que NVidia diseñó la arquitectura CUDA pensando originalmente en el lenguaje C, que exhibe un nivel de programación más bajo que el de herramientas como MatLab. De hecho, su forma de programar es, como se verá en el sub-apartado siguiente, una pequeña variación de este lenguaje. No obstante, la compañía ha procurado que, gracias a la diseminación y popularización de su propuesta, el acceso al programado pueda realizarse con otros lenguajes y entornos de programación que, equipados con diferentes bibliotecas, hagan el proceso de la migración a la GPU mucho más sencillo e intuitivo, de manera que no sea preciso tener presentes en todo momento la arquitectura y características del dispositivo.

MatLab es un lenguaje de programación de alto nivel orientado al cálculo numérico, la visualización y la programación, muy empleado en el entorno de la investigación científica y en el campo de la ingeniería. Debido a que permite trabajar de manera interdisciplinar gracias a la cantidad de toolbox que ofrece[43] constituye la mejor opción para nuestro proyecto, ya que se tiene que aunar de manera eficiente el tratamiento de imágenes junto con la programación paralela. Por tanto será el principal encargado del tratamiento de los datos, reserva y alojamiento de memoria, inicialización de variables y transferencia de información entre dispositivos.

Se van a explicar a continuación las funciones, algoritmos y consideraciones más relevantes a la hora de programar en CUDA. Pero primero debe asegurarse que existe una comunicación ente la GPU y el entorno de programación. Para ello se invoca la función ***gpuDevice()***, que devuelve la información de los dispositivos conectados. En la Figura 3.12 se muestra un extracto de la información técnica de la *NVidia GeForce GTX 960* que saca en consola el programa cuando se introduce el citado comando.

```
CUDADevice with properties:

                      Name: 'GeForce GTX 960'
                     Index: 1
      ComputeCapability: '5.2'
         SupportsDouble: 1
          DriverVersion: 8
         ToolkitVersion: 8
    MaxThreadsPerBlock: 1024
      MaxShmemPerBlock: 49152
    MaxThreadBlockSize: [1024 1024 64]
           MaxGridSize: [2.1475e+09 65535 65535]
             SIMDWidth: 32
           TotalMemory: 2.1475e+09
       AvailableMemory: 1.7147e+09
  MultiprocessorCount: 8
         ClockRateKHz: 1240500
          ComputeMode: 'Default'
  GPUOverlapsTransfers: 1
KernelExecutionTimeout: 1
       CanMapHostMemory: 1
        DeviceSupported: 1
         DeviceSelected: 1
```

Figura 3.12. Información que saca en pantalla el comando gpuDevice().

[43] Como puede ser procesamiento de señales e imágenes, comunicaciones, sistemas de control y así un largo etc.

Como se describió en el apartado 3.2, la GPU consta de su propia memoria de video para la ejecución y el almacenamiento de los programas y datos necesarios. A la hora de ejecutar un kernel cada hilo necesitará acceso a la información que requiera la función. Este proceso lo gestiona, de forma autónoma, el propio MatLab. El usuario únicamente debe declarar que desea transferir un determinado objeto (dato, vector o matriz) a través de dos funciones:

A = gpuArray(X) **:** - Copia el valor de X en la GPU y devuelve un objeto A que puede ser manipulado por funciones que soporten el tratamiento paralelo. A esos objetos MatLab los etiqueta como de tipo gpuArray.

Result = gather(A) **:** - Copia el valor del objeto A desde la GPU a la variable Result de la CPU.

La estructura de un código que se desea programar en GPU consta de una parte del mismo que se ejecutará en la CPU y de otra parte que lo hará en cada hilo de la GPU. La ventaja que supone paralelizar con ayuda del toolbox es que el usuario no se tiene que preocupar de adaptar su script a lenguaje C (que puede ser bastante complejo si no se domina) ni de elaborar y compilar el Kernel que ordene la migración de esa parte del código de la CPU a la GPU por otra parte. Tan solo debe de procurar que lo que pretende llevar a cabo con su script (o con las líneas concretas que desea paralelizar) hagan uso de una serie de funciones que estén recogidas dentro del elenco de funciones que ya se encuentran preprogramadas en el *Parallel Computing Toolbox*. Son funciones que habitualmente se utilizan en la programación convencional en la CPU pero que se encuentran habilitadas para que soporten tratamiento en paralelo al admitir objetos de tipo gpuArray, de forma que la GPU las pueda procesar sin necesidad de que el propio usuario escriba, en un fichero aparte, el denominado Kernel que contiene las líneas de código que desea paralelizar – programadas en lenguaje C obligatoriamente – que la GPU debe seguir para paralelizar[44]. La Figura 3.13 muestra qué funciones tiene implementadas MatLab 2017 compatibles con la paralelización en GPU.

Muchas de ellas han sido empleadas en el algoritmo de procesado de imágenes con el que se ha trabajado en este proyecto mientras que, en algunos tramos del script, ha sido necesario reescribir algunas líneas de forma que se reemplazaran funciones utilizadas no recogidas en el elenco, por otras que sí lo están, en aras de buscar el máximo aprovechamiento del paralelizado (se verá y discutirá a lo largo de los Capítulos 4 y 5).

---

[44] *En el siguiente sub-apartado se verá que, para sacar el máximo potencial de la GPU (nivel 3) es preciso que el Kernel se encuentre implementado en lenguaje C (por ejemplo en un bloc de notas) y debidamente compilado (algo que MatLab hace automáticamente sin participación del usuario). Huelga decir que la complejidad – para un usuario asiduo de MatLab – de proceder por esta vía es significativamente superior. El motivo radica en que éste debe reescribir su algoritmo en C, buscando maneras alternativas de que su aplicación conserve la funcionalidad original en un nuevo lenguaje.*

abs	cond	flipud	isreal	permute	sprandsym
acos	conj	floor	isrow	pinv	spconvert
acosd	conv	fprintf	issorted	planerot	sph2cart
acosh	conv2	full	issparse	plot (and relatec	sprand
acot	convn	gamma	issymmetric	plus	sprandn
acotd	corrcoef	gammainc	istril	pol2cart	sprandsym
acoth	cos	gammaincinv	istriu	poly	sprintf
acsc	cosd	gammaln	isvector	polyarea	sqrt
acscd	cosh	gather	kron	polyder	squeeze
acsch	cot	ge	ldivide	polyfit	std
accumarray	cotd	gmres	le	polyint	sub2ind
all	coth	gradient	legendre	polyval	subsasgn
and	cov	gt	length	polyvalm	subsindex
angle	cross	hankel	log	pow2	subspace
any	csc	head	log10	power	subsref
arrayfun	cscd	histcounts	log1p	prod	sum
asec	csch	horzcat	log2	psi	superiorfloat
asecd	ctranspose	hsv2rgb	logical	qmr	svd
asech	cummax	hypot	lsqr	qr	svds
asin	cummin	idivide	lt	rad2deg	swapbytes
asind	cumprod	ifft	lu	rand	tail
asinh	cumsum	ifft2	mat2str	randi	tan
assert	deg2rad	ifftn	max	randn	tand
atan	del2	ifftshift	median	randperm	tanh
atan2	det	imag	mean	rank	times
atan2d	detrend	ind2sub	meshgrid	rdivide	toeplitz
atand	diag	Inf	min	real	trace
atanh	diff	inpolygon	minus	reallog	transpose
bandwidth	discretize	int16	mldivide	realpow	trapz
besselj	disp	int2str	mod	realsqrt	tril
bessely	display	int32	mode	rectint	triu
beta	dot	int64	movmean	rem	true
betainc	double	int8	movstd	repelem	typecast
betaincinv	eig	interp1	movsum	repmat	uint16
betaln	eps	interp2	movvar	reshape	uint32
bicg	eq	interp3	mpower	rgb2hsv	uint64
bicgstab	erf	interpn	mrdivide	roots	uint8
bitand	erfc	intersect	mtimes	rot90	uminus
bitcmp	erfcinv	inv	NaN	round	union
bitget	erfcx	ipermute	ndgrid	sec	unique
bitor	erfinv	isaUnderlying	ndims	secd	uniquetol
bitset	exp	isbanded	ne	sech	unwrap
bitshift	expint	iscolumn	nextpow2	setdiff	uplus
bitxor	expm	isdiag	nnz	setxor	vander
blkdiag	expm1	isempty	nonzeros	shiftdim	var
bsxfun	eye	isequal	norm	sign	vertcat
cart2pol	factorial	isequaln	normest	sin	xor
cart2sph	false	isfinite	not	sind	zeros
cast	fft	isfloat	nthroot	single	
cat	fft2	ishermitian	null	sinh	
cdf2rdf	fftn	isinf	num2str	size	
ceil	fftshift	isinteger	numel	sort	
chol	filter	islogical	ones	sortrows	
circshift	filter2	ismatrix	or	svds	
classUnderlying	find	ismember	orth	spconvert	
colon	fix	ismembertol	pagefun	sph2cart	
compan	flip	isnan	pcg	sprand	
complex	fliplr	isnumeric	perms	sprandn	

Figura 3.13. Funciones preprogramadas en MatLab 2017 que admiten objetos gpuArray.

Como ya conocerá el lector que está acostumbrado a trabajar en MatLab, las funciones *element-wise* permiten operar sobre cada elemento de un vector o una matriz cuando delante de la función (exponencial, logaritmo, raíz…etc) ,se coloca un punto ".". En la GPU esto se puede hacer utilizando el comando ***arrayfun***:

$$[B_1, \ldots B_m] = arrayfun(func, A_1, \ldots A_n)$$

Donde la función será una de las del repertorio de MatLab (o creadas por el propio usuario como se verá un poco más adelante), A_n son las entradas – objetos del tipo gpuArray guardados en memoria GPU – y B_m son las salidas. De este modo, cada núcleo de la GPU hará uso de sus múltiples hilos para realizar dicha operación sobre el elemento que corresponda. La ventaja es clara, la CPU necesita ir realizando la operación en cada elemento del vector o matriz uno por uno. En la GPU se hará muchísimo más rápido gracias a sus múltiples hilos[45].

[45] *Pero no será posible controlar que cada hilo realice solo la operación en un elemento del vector/matriz.*

A la función se la pasa con el clásico **@** delante. Lo bueno que tiene *arrayfun* es la flexibilidad que ofrece a la hora de elegir los parámetros de entrada y salida. Permite elaborar la función manualmente, algo que es muy útil para cálculos personalizados para una aplicación concreta. Así, si en la función creada se tienen tres argumentos de entrada, por ejemplo, al emplear el comando deberemos escribir *arrayfun(@nombrefuncion,arg1,arg2,arg3)*.Como limitaciones cabe decir que solo devuelve escalares, tomando operandos escalares (cada elemento del vector/matriz) y no tiene la posibilidad de acceder a elementos concretos de los argumentos de entrada, sino a los que corresponde para cada operación en la que un elemento del vector/matriz participa en una operación simple elemento a elemento, tal y como ocurre en la CPU convencional con el punto delante (suma, resta, división…etc).

Hermana pequeña de la anterior, más sencilla y con menos posibilidades, es la función ***bsxfun***. Básicamente hace lo mismo, salvo que en ella no se pueden solicitar las salidas que se quiera, sino que solo dará una, la que corresponde a operar elemento a elemento sobre los vectores/matrices que constituyen los dos únicos argumentos de entrada.

También se puede hacer uso de la llamada ***pagefun***, que permite aplicar funciones a cada página de una matriz (Matlab define como páginas los valores de la tercera coordenada de una matriz tridimensional). Es decir, se estaría trabajando con cada una de las matrices bidimensionales que componen la matriz global. En la Figura 3.14 se puede ver en qué consiste exactamente una matriz llamada "tridimensional". No es otra cosa que una matriz 2D convencional que tiene varias páginas.

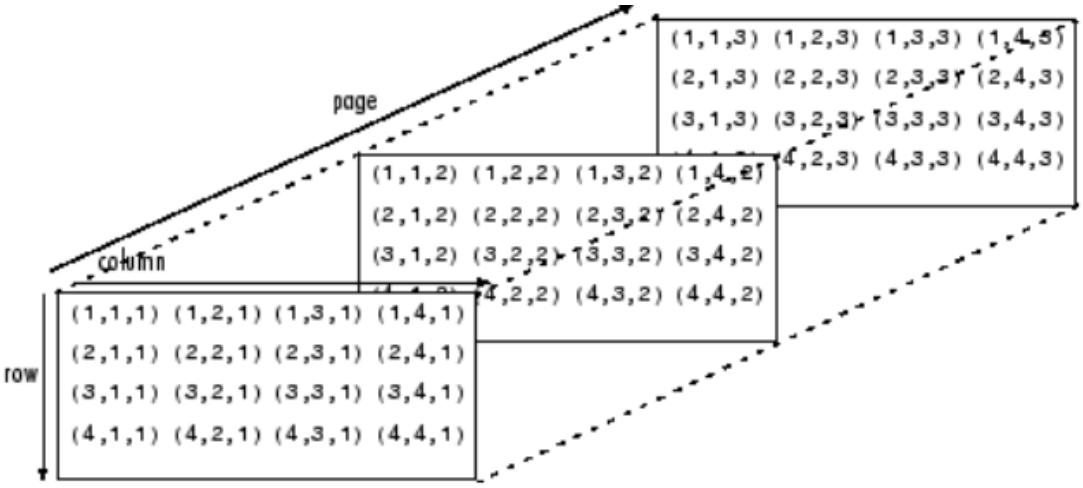


Figura 3.14. Representación gráfica de una matriz tridimensional.

En este caso, sí que se tiene control total sobre cada "sub-matriz" , permitiendo acceder a cualquier elemento de ésta. Además, no existen restricciones en los parámetros de entrada ni de salida, por lo que parece una solución más flexible que *bsxfun*. Sin embargo, tiene una desventaja con respecto a las dos anteriores que provoca que tampoco sea de gran interés y es el hecho de que acepta únicamente unas pocas funciones de Matlab. Esto quiere decir que el usuario no tiene la oportunidad de programar funciones arbitrarias para usarlas junto a esta llamada. Cabe aclarar que el resultado obtenido (argumento de salida), como antes también ocurría, será un vector de escalares y estará ordenado por páginas.

Como se puede ver, en *MatLab* la implementación de líneas de código que permitan

paralelizar cálculos no es en esencia complicada. El inconveniente que presenta proceder de esta forma es que se considera *paralelización de grano grueso*. Esto es porque, aunque para el usuario es intuitivo el poder hacer uso del elenco de funciones que la GPU sabe cómo ejecutar gracias al toolbox de MatLab[46], cuando las variables se hallan en la memoria de video no es posible controlar la configuración de los hilos. En consecuencia, al no implementar un Kernel en el que inicializar el número de hilos que se quiere declarar y asignar al cálculo, se pierde la capacidad de sacar el máximo provecho de la GPU[47] .

No obstante, en este trabajo se refiere a este tipo de paralelización como de nivel 2 porque, ahora, ya no se emplean los pocos núcleos que puede tener el procesador de un PC de sobremesa, sino que los cálculos paralelos los van a realizar miles de hilos que la GPU tiene a su disposición. Por tanto, pese a que no se pueda ejercer un control fino, el cambio es muy importante por el número de elementos que simultáneamente van a estar procesando un mismo cálculo. En este sentido, para un usuario ducho en MatLab, la dificultad de paralelizar en GPU no tiene el añadido de dificultad de tener que lidiar con la sintáctica de un lenguaje de programación poco usado a nivel ingenieril como C (fuera del ámbito de la informática), sino que reside en varios aspectos que lo envuelven y hacen crítica la toma de decisiones:

- **Conseguir adaptar el código original:** - Para que emplee en la medida de lo posible las funciones preprogramadas que el toolbox ofrece (puede ser que se usen en el código original funciones que no se encuentran en el elenco de la Figura 3.13).

- **Elegir bien las variables que se desean migrar de la CPU a la GPU (tamaño):** - Para ver si compensa el tiempo de cálculo adicional que suponen las transferencias (ida y vuelta).

- **Elegir bien en qué momento migrar es provechoso:** - Puede ser que el cálculo sea muy ligero y, pese a que se puede paralelizar, e incluso reducir el tiempo de cálculo de la/s línea/s de código en si, las transferencias CPU/GPU y GPU/CPU consuman más tiempo del que originalmente consumía/n sin paralelizar.

- **Emplear el monitor de recursos:** - A medida que se tracen los intentos de paralelizado por tramos, de manera que el avance sea siempre a mejor. Es importante recordar que MatLab ya emplea paralelización interna en algunas de sus funciones programadas.

- **Tener presente la programación heterogénea:** - Ésta hace alusión a que habrán cálculos con las variables guardadas en memoria GPU que se procesarán en la tarjeta gráfica y variables guardadas en el workspace de MatLab que se procesarán en la CPU. Por tanto, si se emplea una función no preprogramada en MatLab, que hace uso de datos guardados como GPUArray, será interesante introducirse en ella y tratar de paralelizarla también todo lo posible si no lo está.

---

[46] *MatLab lleva a cabo, de manera autónoma, todos los pasos comentados en el cuadro amarillo.*

[47] *De acuerdo al mecanismo de funcionamiento que ésta tiene y que ya fue explicado en el apartado 3.2.*

Antes de pasar a detallar brevemente en qué consiste la programación CUDA en su entorno natural en C, resulta interesante indicar que existen dos posibilidades más de explotar la capacidad de la GPU haciendo uso del *Parallel Computing Toolbox* de MatLab. Se va a describir en qué consiste cada una de ellas.

Funciones *mex*

La función ***mex*** permite añadir más funciones a las que ya dispone el toolbox – ver la Figura 3.13 – y emplearlas exactamente igual (no es más que una ampliación del repertorio de funciones que ya vienen de serie). El proceso es tedioso, pues hay que adaptar y compilar el código, siguiendo una serie de especificaciones concretas, para asegurar que éste entra a formar parte de la biblioteca como una función más con la que poder trabajar en GPU. Aclarar que no es un mecanismo restringido al procesamiento de la GPU ni mucho menos, sino que se extiende a cualquier código en C externo que se desee incorporar a MatLab.

Función *parallel.gpu.CUDAKernel* y *feval*

Con estas funciones, pertenecientes al *Parallel Computing toolbox*, es posible ejecutar un *kernel* programado en C, de manera similar a como se haría en CUDA, pero gestionando los recursos desde el entorno de MatLab. En primer lugar se crea un objeto en el workspace en el que se determinan ciertos parámetros importantes a la hora de ejecutar el código en GPU. En segundo lugar se ejecutan las funciones relacionadas con dicho objeto, introduciendo las variables y atributos de entradas que la función requiera.

La creación del objeto de tipo parallel.gpu.CUDAKernel se realiza con la siguiente función:

KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE, FUNC);

Esta función crea un objeto (KERN) introduciendo como parámetros de entrada:

PTXFILE: Archivo de extensión ***.ptx*** (*Parallel Thread Execution*). Contiene las instrucciones necesarias para ejecutar el programa enfocándolo a la ejecución sobre los hilos. Usa un lenguaje semejante a ensamblador y se puede obtener tras compilar el archivo ***.cu***.

CUFILE*:* Archivo de extensión ***.cu*** que contiene el kernel escrito en CUDA C.

FUNC*:* Nombre de la función dentro de CUFILE que queremos ejecutar.

Creado el objeto, es necesario modificar ciertos campos internos para personalizar la ejecución del kernel, como el número de hilos y bloques. Se realiza accediendo al objeto como una estructura y asignando un valor al campo *ThreadBlockSize* del siguiente modo:

NOMBRE DEL OBJETO.ThreadBlockSize=[]

La ejecución de la función relacionada con el objeto parallel.gpu se realiza mediante el uso de la función ***feval*** de la siguiente forma:

[y1, y2] = feval(KERN, x1, x2, x3)

Como siempre, un ejemplo es sinónimo de clarificación de conceptos. Con él se pretende exponer cómo compartir información con la GPU (tanto en envío como en recepción de datos una vez procesados) y la forma de asignar y usar los hilos disponibles para el cálculo. Se tomará el ejemplo propuesto de manera genérica (lenguaje neutro) introducido en el apartado 3.2 al explicar el funcionamiento de una GPU. Será crear un vector de 10 elementos con números aleatorios del 1 al 100 y sustituir por 0 en las posiciones en los que éstos sean par.

MatLab (Host)

```
% Se crea el objeto relacionado con el archivo inspeccion.cu

KERN=parallel.gpu.CUDAKernel('inspeccion .ptx', 'inspeccion.cu','indexado');

% Se crea un vector con 10 elementos que alberga números aleatorios del 1
% al 100. Se crea directamente en la memoria de la GPU:

v=ceil(100*rand(1,10),'gpuArray');
numero_elementos=length(v);

% NOTA: - Es importante cerciorarse de que las funciones empleadas fuera
% del Kernel estén en el repertorio que ofrece Parallel Computing Toolbox
% compatible con la paralelización en GPU (en este caso las tres lo están)

% Se asignan tantos hilos como elementos tiene el vector generado:

KERN.ThreadBlockSize=[numero_elementos,1,1];

% Se manda ejecutar el Kernel que contiene las instrucciones con las
% operaciones que se desean paralelizar sobre el vector v:

resultado_GPU=feval(KERN,v);

% Dado que esto podría ser únicamente un tramo de código insertado en otro
% más extenso, la CPU necesitará recuperar el dato en memoria para
% proseguir con la ejecución del resto de líneas que no se paralelizan:

resultado_CPU=gather(resultado_GPU);
```

El KERN al que invoca el script de MatLab tiene que estar implementado en lenguaje C, en un fichero (bloc de notas, por ejemplo) y guardado con extensión *.cu*.

GPU (Device)

```
_global_ void indexado (double * v)

{

 int idx = blockDim.x * blockIdx.x + threadIdx.x; // identificador de hilo

if (v[idx]%2 = 0) then

v[idx] = 0

end

}
```

Al ser una función del tipo *global*, se está indicando que la comunicación CPU/GPU sea bidireccional, esto es, que la información resultante será también rescatada por la CPU.

Lo que el código del Kernel está ordenando es, primero obtener el identificador de hilo, para cada uno de los diez hilos que fueron declarados en el script de MatLab en el cual, además de invocar al Kernel, se indicaron unos atributos. Almacena cada identificador en la variable idx. El método para asignar identificador a cada uno ya fue descrito y un resumen puede consultarse en la Figura 3.7. Por lo que allí hay reflejado, puede verificarse que la expresión (3.1), cualquiera que fuera el número de bloques en los que la GPU decidiera repartir sus hilos[48] (tanto si es uno con diez, dos de cinco…etc) se cumple correctamente.

Lo segundo que hace es, una vez obtenidos los identificadores de hilo por cada hilo, envía un puntero al vector v sobre el que trabajarán. Con ellos almacenados en el vector idx, todo lo que tiene que hacer es ir evaluando cada elemento del vector v con los elementos contenidos en idx y aplicar la división entre dos, que, si da resto cero, habrá quedado identificado como par. En tal caso, la siguiente instrucción consiste en sustituirlo por un cero. El resultado es exactamente el comentado en el ejemplo al que se hace alusión, esto es, un vector compuesto por números aleatorios del uno al cien (solo impares) en el cual determinadas posiciones serán ceros. Es importante recordar, una vez más, que la ejecución de los hilos no es ordenada.

Con esto queda cerrado el sub-apartado e introducidas las herramientas de paralelizado que se han empleado para optimizar el algoritmo que procesa imágenes con el que se ha trabajado en el presente proyecto. Para abordar los objetivos que rodean al TFG, objeto de esta memoria, los niveles de paralelizado de los que se han hecho uso han sido el 1 y el 2, con especial énfasis a éste último, ya que ha sido el que ha demostrado ser más eficaz con nuestro algoritmo y arrojado resultados fructíferos.

Inicialmente se estudió la posibilidad de aprender a programar en lenguaje C con objeto de poder hacer uso, tanto de las funciones *mex*, como de la *parallel.gpuCUDAKer*nel que, como se ha visto, son de fácil implementación en MatLab (con una estructura escueta, clara y concisa). El problema es que, tal y como se ha indicado, requiere de la inserción de un archivo compilado en el que se halle escrito el código en lenguaje C, que es un lenguaje de programación de más bajo nivel que el que emplea MatLab (mucho más sencillo e intuitivo).

Programar bien en C requiere tiempo, entrenamiento y, en muchas ocasiones, experiencia. Es algo que, quizá no queda patente viendo un ejemplo sencillo como el propuesto pero, cuando se describa el algoritmo de procesado de imágenes (Capítulo 4),que consta de más de 500 líneas de código, se verá que son su ingente cantidad de bucles los que constituyen el cuello de botella. Incluso tratar de paralelizar únicamente éstos, es hablar de muchas líneas con funciones y operaciones complejas que requeriría bastante más tiempo del asignado a un TFG convencional reprogramar, y con una sintaxis a la que, una persona asidua a entornos de programación como MatLab, no está acostumbrada.

---

[48] *Esto es así porque, el control preciso de cada hilo, de cara a asignarle un determinado cálculo, o de los bloques que se desean crear, junto a otra serie de muchos otros parámetros relativos a la arquitectura CUDA (lo que se conoce como paralelizado de grano fino) no es posible abordarlo desde un entorno de programación de alto nivel como es MatLab, siendo necesario acudir otro de más bajo nivel como es empleando lenguaje C.*

No obstante, los resultados obtenidos en la optimización han sido muy prometedores (Capítulo 5) y abren la puerta a líneas de investigación y futuras mejoras, especialmente si se concentran los esfuerzos en tratar de reescribir el código de una forma eficiente en lenguaje C (Capítulo 6, "Perspectivas y trabajos futuros").

3.4.2.2. En el entorno de programación de CUDA C

Resta todavía por describir en qué consiste la paralelización de nivel 3 o, dicho de otro modo, la que corresponde al uso de CUDA en su versión natural (fuera del entorno de Matlab) y que emplea lenguaje de programación C. Como se ha comentado unas líneas más arriba, no constituye la vía por la que se ha llevado a cabo la paralelización del algoritmo de procesado de imágenes con el que se ha trabajado. En este sentido, lo que perseguirá este sub-apartado será proporcionar una pequeña pincelada con las nociones más relevantes de este tipo de programación.

Ésta explota al completo, ahora sí, las capacidades de las GPUs que *NVidia* ofrece (incluida la nuestra), pero en fuentes tan reputadas como la propia *MathWorks* **[24],[25]** se hace alusión a ella como de nivel experto o avanzado, lo que da soporte y respalda la justificación de que, en el marco de tiempo estipulado para un proyecto de estas características, adentrarse en ella, haciendo uso de un código ya escrito y complejo como es el de procesado de imágenes, queda fuera de los objetivos que se marca la asignatura asociada al TFG.

En lo que concierne ya a los aspectos particulares de la inicialización, básicamente se trata de declarar las funciones en el Kernel y de asignar los identificadores de hilo (que ya se vio cómo hacerlo con anterioridad). Para declarar una función, se emplean los comandos ***void***, ***int*** y ***float***, entre otros, según sea el tipo del valor de retorno. En CUDA, además de disponer de dichas etiquetas, se debe de incorporar uno de los comandos siguientes, según el modo de ejecución de las funciones:

Para una ejecución en el host: - En el caso de realizar un volcado de datos desde el host para que el device los procese se utilizará el comando *_global_*

_global_void nombre _función (…)

Para una ejecución en el device: - En el caso de que el kernel necesite ejecutar alguna función auxiliar sin la necesidad de transferir la información al host se empleará el comando *_device_*

_device_void nombre _función _auxiliar(…)

Se puede apreciar que la estructura de estas declaraciones, como se vio al describir la sintaxis que empleaba la identificación de hilos, es homóloga al lenguaje de programación en C (es una variación). Introducido esto, el resto de la función del Kernel que se ejecutará en CUDA corresponde ya al propio manejo de la escritura en C y de la aplicación de las funciones y operaciones del script que el usuario tenga en mente. Tal y como se ha anticipado en el anterior sub-apartado, éstas son cuestiones ya de programación pura en las que no se va a profundizar, dado que es un nivel de paralelización que en este proyecto no se ha explorado.

4. METODOLOGÍA EXPERIMENTAL

4.1. Introducción

En este Capítulo que ahora se inicia va a procederse a la descripción de la metodología experimental, asociada a los ensayos y desarrollo del código base, que se ha empleado para llevar a cabo la optimización buscada. Se irá especificando cuál ha sido el planteamiento, el diseño de los mismos, los varios arreglos necesarios y los pasos seguidos para tener éxito. En este sentido, se comentarán también las necesidades y problemas acaecidos, así como las acciones mediante las cuales se han ido solventando y adaptando dinámicamente, explicando con detalle cuáles han sido las bases que soportan a las decisiones tomadas.

El inicio metodológico de este TFG, cuyo carácter predominantemente informático lo hace un tanto peculiar, parte de la base de una adecuada prospección bibliográfica. La primera cuestión que hubo que afrontar era la de estudiar qué hacía el algoritmo que procesa imágenes, tratando de indagar en los aspectos más incisivos con el fin de adquirir una buena idea global de su mecanismo de funcionamiento. Cabe recordar, llegados a este punto, que el citado algoritmo no ha sido elaborado en el marco del presente proyecto, siendo un programa que ya empleaba el grupo de Técnicas Ópticas en sus ensayos habituales.

La segunda, era la necesidad de adquirir nociones sobre aspectos como: Qué es CUDA y en qué consiste, qué herramientas de paralelización existen y cómo emplearlas, qué recursos se necesitan para ello, cómo realizar una implementación adecuada para un código ya confeccionado en MatLab, etc. En ese sentido, las referencias (*tutoriales*, *webinars*, *papers*, *tesis*, *TFGs*…etc) a las que se han ido haciendo alusión durante el Capítulo 3, han constituido un primer pilar básico sobre el que cimentar el desarrollo del código-base sobre el que trabajar

En este contexto, lo que se ha pretendido es, tomando como referencia a dicho código-base, plantear y abordar por etapas la búsqueda de su optimización. Resumidos esquemáticamente y en orden cronológico, los pasos han sido los siguientes:

1) Entendimiento del código base: - Adquirir una visión global de su funcionamiento.

2) Estudio y caracterización de las partes del script susceptibles de ser paralelizadas

3) Prospección bibliográfica: - Adquirir un "*feedback*" acerca de las herramientas de programación necesarias para llevar a cabo los paralelizados de nivel 1 y 2.

4) Realización de un pre-estudio para identificar los cuellos de botella que presenta el código: - Énfasis en los tramos que mayor porcentaje del tiempo de cálculo total consumen.

5) Iniciar la re-programación del código: - Modificar, sustituir o suprimir los tramos convenientes. Asegurar una trazabilidad para poder discernir si los caminos que se van tomando mejoran o empeoran el código provisional.

6) Obtención del código alfa (Código que se considera completamente optimizado)

7) Realización de estudios parámetricos: - Comparar el código base con los códigos paralelizados de niveles 1 y 2, extrayendo conclusiones robustas y consistentes.

Nota: - A lo largo de este capítulo se van a ir describiendo los siete pasos recién citados. Matizar que en el cuarto y quinto, dado que ha sido necesaria la realización de pequeños cálculos orientativos (sin consistencia estadística), se mostrarán algunas tablas y gráficas que justifiquen algunas tomas de decisiones. Los resultados obtenidos, tanto en tiempo como en coste computacional, así como los análisis y las discusiones, serán ya abordadas en el Capítulo 5. Será también en éste cuando se expongan los resultados arrojados por el planteamiento de diversos estudios paramétricos, que serán explicados en posteriores sub-apartados.

4.2. Código base

El código base se adjunta en el *Anexo A*, al final de la memoria. En él puede verse el código íntegro con el que se ha trabajado y que servirá como soporte a la descripción de sus ejecuciones. Para facilitarla un poco, ésta se realizará mediante secciones, tal como el propio código se encuentra implementado. Se reflejan también los ocho bucles en los que se ha dividido el paralelizado. Tras explicar el código se explicará por qué se han elegido éstos y no otros que quedan por el camino, o incluso líneas de cálculo que no corresponden a ningún bucle.

En lo que se refiere al algoritmo, éste se basa en procesar imágenes, adquiridas mediante la técnica LEI, para caracterizar y analizar la combustión en un motor MEC de dos tiempos. La idea concreta es la de cuantificar la cantidad de hollín presente en la llama que se produce en la cámara de combustión. Por secciones, el script sigue los siguientes pasos:

Inputs

Se especifica la dirección donde se encuentra la carpeta de la que se tienen que cargar las imágenes. Se fija el valor de distintos parámetros como la intensidad del LED, el número de imágenes por repetición que se toman antes de la inyección del chorro…etc. Los valores que se pueden ver en el Anexo A corresponden a un tipo de experimento determinado, lógicamente varían en función de lo que se pretenda estudiar y de sus condicionantes.

A continuación se inicia el gran bucle que acoge a todos los demás y que lo que va a hacer es recorrer un vector que contiene un fichero con fechas. Dentro del mismo se encuentra otro *for* que recorre otro vector donde se direcciona a carpetas que contienen un determinado número de casos [49], de forma que irá ejecutando caso a caso a cada vuelta de bucle.

[49] *Se encuentra "capado" de modo que solo realice una única vuelta. Esto es porque la optimización se ha llevado a cabo basándose en la mejora de un único caso (un ensayo en el que se procesa un determinado número de imágenes a lo largo del script). El motivo: si se logra una mejora ostensible, tanto en este bucle como en los más internos, ésta se verá multiplicada cuando el bucle de fechas tenga que procesar un alto número de casos por día y durante varios días.*

Controles de procesado

En el siguiente bloque se establecen una serie de controles que efectúan lecturas en diferentes ficheros. Aparecen sentencias condicionales cuyo principal objetivo es identificar si alguna repetición de un determinado caso (o éste en su totalidad) han sido previamente procesados en una anterior ejecución del código, algo que es especialmente útil cuando una carpeta alberga un elevado número de casos y se lanza con todo la ejecución.

Como es de esperar, en caso de que el programa detecte que existe ya un resultado guardado en el fichero (fruto de un anterior procesado) se salta un determinado número de líneas, lo que ahorra en tiempo de cálculo.

Procesado

Cuando está vacío el vector de las condiciones establecidas para la ejecución del caso, se ejecutan las líneas que se hallan dentro del comando *else* para establecerlas. Hace lo propio para el fichero que alberga la configuración de la cámara rápida que adquiere las imágenes.

A continuación se fijan el número de repeticiones (adquisición de imágenes por parte de la cámara; normalmente son treinta), el número de imágenes que adquiere la cámara por cada repetición, los frames por segundo y las dimensiones con las que se captarán las imágenes.

Detección de la posición de la tobera de inyección

Es un bloque de instrucciones, cuyas líneas de control revisan si existe ya un archivo que contenga las coordenadas x,y del inyector, en una imagen captada de la cámara de combustión cuando está vacía (en ausencia de chorro) y que sirve como referencia. En caso de que no esté generado dicho archivo *.mat*, carga la imagen y solicita al usuario que indique el punto sobre la pantalla donde aprecia visualmente el inyector, de forma que guarda las coordenadas y genera un archivo denominado *Inypos.mat*.

Si ya existiera, un anuncio en consola lo advertiría. Este es el caso del algoritmo tal y como se ha trabajado con él. Así, para buscar la optimización se ha dejado intacto el archivo (no se ha borrado) de modo que estas líneas se las salte y no tener así que presionar la posición del inyector en cada nueva probatura para examinar cambios en el código.

Determinación de los niveles de referencia del ruido y la luz existente en el ambiente

Se toman cuatro imágenes, por cada repetición, con la cámara de combustión iluminada por el LED y cuatro más en ausencia de luz. Se guardan en sendas matrices tridimensionales, donde el número de "páginas" de la matriz (ver Figura 3.14) es cuatro por ser el número de imágenes que se han tomado (recordar que las imágenes se procesan como matrices).

Esto se realiza debido a que en el ensayo se hace incidir luz sobre la muestra con una intensidad dada, de forma que la primera imagen que se captura es luminosa, enfocando a una cámara de combustión todavía carente de chorro. A continuación, el LED deja de iluminar, con lo que el siguiente fotograma que se recoge es en ausencia de luz y, por tanto, totalmente negro.

Se hace así porque la luz transmitida que le llega al detector no proviene exclusivamente del remanente que no absorbe o dispersa las partículas de hollín sino que, además, éstas también emiten luz al desexcitarse.

De esta forma, si se toman intermitentemente un fotograma iluminado y otro no, a intervalos de tiempo muy cortos y constantes, se podrá ir siguiendo el desarrollo del encendido y la formación del hollín, desde que se inyecta el chorro hasta que éste se produce. En consecuencia, se podrá cuantificar la cantidad de hollín presente a través del restado a cada imagen luminosa – variable en intensidad de unas a otras – su homóloga en ausencia de luz, puesto que la primera tiene en cuenta la atenuación al atravesar la luz el hollín y la segunda el aporte de luz por radiación que produce el hollín caliente (radiación de Planck).

También se realiza con el ruido de fondo y se elaboran sendos vectores con el promedio.

Detección de la combustión

Mediante una serie de operaciones con las matrices anteriores, se trata de examinar si en alguna de ellas se detecta el encendido. Para ello se evalúa el nivel máximo de intensidad de luz de cada imagen y se contrasta con el valor umbral del LED. En caso de que se detecte, se crea un vector *FireRepCount*, que va asignando un 1 y guardando aquellas donde exista combustión. En caso de que este vector quede vacío, aparecerá una advertencia en pantalla indicando que en el caso que se está procesando no existe combustión.

Los que se han denominado *Bucle 1* y *Bucle 2* en el script recogen tanto este paso como el anterior.

Procesado de imágenes de la técnica LEI *(Bucle 3)*

Aquí es ya donde se efectúa el verdadero procesado de las imágenes adquiridas. Lo hace a través de un bucle en el que se examina las imágenes con luz/no luz, las categoriza, las introduce en sendas matrices y les aplica un filtro.

Interpolación de tiempo en las imágenes *(Bucles 4 y 5)*

Busca las imágenes con y sin luz en las matrices anteriores e inicia una serie de contadores para vincular los inicios y los finales de luz a tiempos que permitan discriminar.

Cálculo de la cantidad de hollín (KL) y guardado *(Bucles 6 y 7)*

Es el paso último del procesado. Se producen las operaciones de restado, de las matrices que contienen imágenes no luminosas a las luminosas, para poder establecer la cantidad de luz transmitida y aplicar así la ley de *Lambert – Beer*, que ya se introdujo. Finalmente guarda los resultados en un fichero que crea a tal efecto y limpia las variables que contienen los resultados del KL de hollín (por repetición) de la memoria.

A continuación hace lo propio pero buscando ya obtener un promediado de concentración de hollín.

Construcción de gráficos con los resultados de los KL promedio *(Bucle 8)*

El paso final del código. Elabora una serie de *plots* en los que va mostrando la cantidad de hollín a cada tiempo, para ir siguiendo el transcurso de la combustión. Son esas gráficas las que al final le interesan al investigador, ya que arrojan información en 2D con mapas de concentración de hollín en la llama. De este modo, pueden medirse y evaluarse múltiples parámetros de la combustión (según interese por el experimento) como la penetración del chorro, la longitud líquida, las temperaturas…etc.

Terminado el "ploteo", se lanza un cartel en consola que lo advierte, se limpian todas las variables excepto las necesarias para la siguiente iteración del bucle principal (el que procesa un caso a continuación de otro) y se lanza otro aviso indicando que el procesado del caso ha concluido.

Una vez entendido el funcionamiento del código, el siguiente paso consistía en hacer una revisión más profunda (línea a línea) tratando de identificar qué partes podrían ser las más susceptibles de beneficiarse del aligeramiento de cálculo que ofrecen las distintas herramientas de paralelizado. Tales partes fueron rápidamente identificadas como los bucles del código, a los cuales la instrucción *parfor* (nivel 1 de paralelización) podría tratar de incorporárseles en primera instancia. También por lo que se ha estado describiendo sobre el funcionamiento de los hilos de una GPU, se apunta a los bucles como potenciales candidatos.

En este contexto, dado que también las líneas de código con operaciones sencillas son buenas elecciones para paralelizar[50], todo lo que resta es practicar un pequeño pre-análisis que arroje los principales cuellos de botella que ralentizan al algoritmo.

En el *Anexo B* se han incluido capturas de pantalla de la información de los ocho bucles que arroja MatLab cuando, en vez de "Run", se presiona el botón "Run and Time", el cual proporciona un perfil completo, describiendo todos los resultados del tiempo de ejecución del script, tanto por líneas como por funciones.

Antes de pasar a comentar brevemente qué han mostrado, cabe apuntar una serie de consideraciones que se han tenido en cuenta a la hora de optimizar el algoritmo:

- **Se ha decidido trabajar únicamente con los ocho bucles marcados en el Anexo A:** - Representan el 97,5 % del tiempo total de ejecución del código. De este modo, tanto las líneas que se encuentran antes del primer bucle, como las que se hallan entre ellos, representan el otro 2,5 % sumadas. La inmensa mayoría son líneas de definición de parámetros de control, de creación de vectores o matrices vacías, sentencias condicionales…etc, lo que les confiere un carácter independiente, no mereciendo la pena paralelizarlas individualmente.

- **Se han dejado fuera algunas líneas:** - Las asociadas a guardado de variables y a la configuración de las gráficas del Bucle 8, por consumir ambas un tiempo significativo dentro de los respectivos bucles y no ser paralelizables de ningún modo (desvirtúan las mejoras).

[50] según reporta la mayoría de la bibliografía citada y referenciada en esta memoria.

Apuntado esto, indicar que el código ejecutado para obtener las capturas de pantalla del *Anexo B* ha sido empleando un número de repeticiones de cinco[51] (que se introduce en el Bucle 1 y afecta al resto excepto al 8).

La Figura 4.1 muestra un extracto que *MatLab* también incluye en su "profiler". Se recoge el tiempo de cálculo total del script, así como una relación de las funciones y comandos, ordenados en orden decreciente, que más tiempo consumen.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
Codigo_base_puro	1	37.235 s	15.821 s	
imshow	246	12.900 s	4.377 s	
GetMovie	10	6.175 s	0.597 s	
imreadmraw	10	5.454 s	5.454 s	
initSize	246	3.096 s	0.163 s	
movegui	246	2.479 s	2.442 s	
newplot	492	2.427 s	0.463 s	
imrotate	246	1.728 s	0.774 s	
basicImageDisplay	246	1.460 s	1.090 s	
newplot>ObserveFigureNextPlot	492	1.267 s	0.019 s	
clf	245	1.248 s	0.058 s	
rot90	246	0.931 s	0.931 s	
isSingleImageDefaultPos	246	0.816 s	0.696 s	
imshow>getBorder	246	0.783 s	0.780 s	
graphics\private\clo	737	0.773 s	0.773 s	
graphics\private\clearscribe	245	0.739 s	0.006 s	
findall	245	0.733 s	0.678 s	
newplot>ObserveAxesNextPlot	492	0.648 s	0.044 s	
cla	492	0.604 s	0.045 s	
clearvars	5	0.551 s	0.540 s	
imageDisplayParseInputs	246	0.269 s	0.016 s	
imageDisplayValidateParams	246	0.231 s	0.031 s	
...:PositionUtils.setDevicePixelPosition	492	0.168 s	0.067 s	

Figura 4.1. Relación de funciones y comandos que más tiempo consumen en el script.

En el *Anexo B* puede verse como, sumando los tiempos de todas las líneas de código contenidas en los ocho bucles, e incluso omitiendo aquellas cuyos tiempos no llegan a 0,01 segundos, el tiempo total obtenido es de 36,297 segundos. Ello representa el 97,5 % del tiempo total que consume la ejecución del código. Por tanto, la consideración anteriormente introducida de "atacar" la mejora de estos bucles está completamente justificada.

Dentro de los ocho bucles, los de mayor orden de prioridad y que requieren la mayor concentración de los esfuerzos son, tal como muestra visualmente la Figura 4.2, los bucles 8 y 6 respectivamente (y con mucha diferencia). Seguidos de los bucles 1,3 y 7 que, más o menos, están a la par. El bucle 4 sería el siguiente y, los bucles 2 y 5, se aprecian insignificantes, por lo que serán los últimos en tratar de paralelizarse.

[51] Las cámaras rápidas habitualmente están configuradas para que realicen 30 repeticiones, si bien en el script se puede manualmente solicitar un número inferior, dado que las imágenes están guardadas (solicitar más obviamente no es posible).

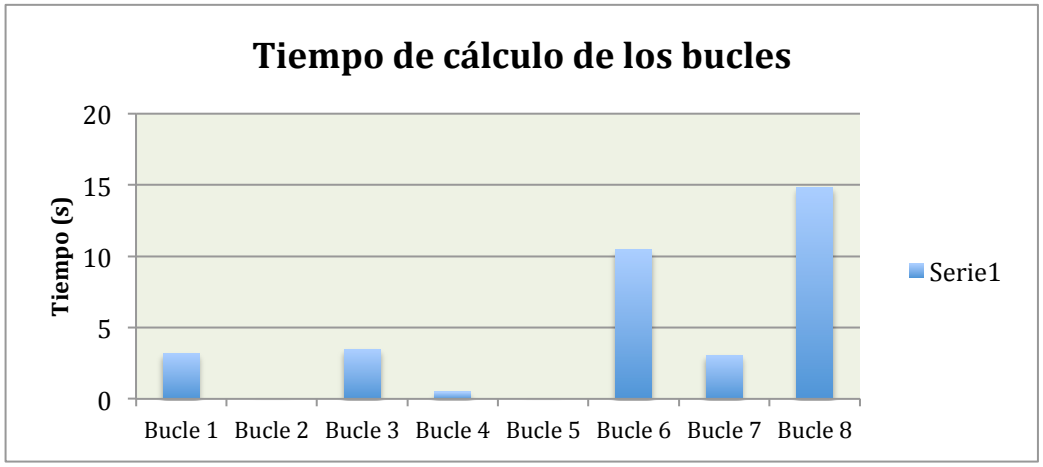


Figura 4.2. Gráfica que muestra los tiempos de cálculo de cada bucle en el código.

Nota importante: - Para optimizar del algoritmo se ha trabajado en todo momento con un número de repeticiones de 5. Se decidió así debido a la gran cantidad de veces que hubo que ejecutarlo para estudiar y definir los cambios que se fueron produciendo.

Anticipar que, en realidad, no era el bucle 8 el más importante pues, a diferencia del resto, el efecto de aumentar las repeticiones no le afecta para nada en el script sin paralelizar (no ocurre lo mismo con el paralelizado, como se discutirá en el Capítulo 5).

Descritos y justificados todos estos hechos, se pueden dar por concluidos los cuatro primeros pasos establecidos al inicio de este apartado. El quinto, que es la re-programación del script para buscar la mejora, se describirá separadamente en los dos próximos apartados, dado que proceder al paralelizado vía CPU, o hacerlo vía GPU, implica grandes diferencias.

4.3. Trazabilidad en la optimización del código

4.3.1. Paralelizado en la CPU

La búsqueda de una optimización del código de procesado de imágenes por medio de la paralelización haciendo uso del *Parallel Pool* permite – como se vio en el Capítulo 3 – el uso de todos los núcleos de que dispone el ordenador. Es algo sencillo, pues únicamente requiere de la inserción de comandos *parfor* a discreción allá donde sea posible. Y en los tramos donde no lo sea, tratar de emplear los "trucos" vistos, que permiten solventar algunos de los casos donde el citado comando origina problemas.

[52] El mismo código base que ha generado los resultados de las Figuras 4.1 y 4.2 , así como las capturas de pantalla del Anexo B.

La Figura 4.3 muestra el coste computacional asociado a cada uno de los 8 núcleos cuando se ejecuta el código base (sin paralelizar) con cinco repeticiones[52] . Ha sido recogido con el monitor de recursos *afterburner*.

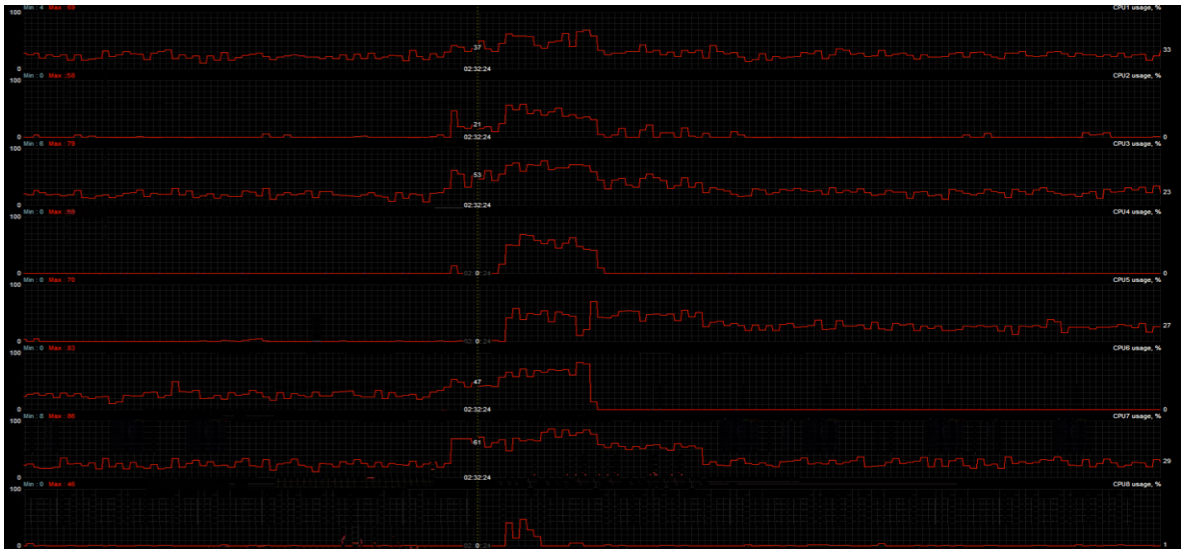


Figura 4.3. Monitor de recursos afterburner mostrando la carga computacional.

No se termina de apreciar con claridad, pero los porcentajes de recursos computacionales están ordenados, de arriba abajo, desde el núcleo 1 hasta el 8. Se pueden apuntar varias cosas:

• Antes iniciarse la ejecución (aproximadamente un poco antes de la raya vertical de trazos), los núcleos 2, 4, 5 y 8 estaban completamente inactivos. El resto tenían algo de actividad, que se debe a los procesos internos y tareas que el propio procesador realiza constantemente.

• Una vez iniciada, los cuatro núcleos inicialmente parados empiezan a registrar actividad de forma no sincronizada. Esto ocurre porque el procesador va recurriendo a ellos según las necesidades de cálculo. El núcleo 8 actúa durante un pequeño intervalo de tiempo y no se solicita más.

• En alguno de los momentos de la ejecución, los 8 núcleos están trabajando sin pausa a más del 50 % de su rendimiento en algún caso. Esto es indicador de que el código es pesado.

• Ninguno de los núcleos alcanza el 100 % de operación, lo que evidencia que un paralelizado por CPU difícilmente podría permitir un mayor uso de alguno de ellos.

A este último hecho se le puede sacar más punta si se razona adecuadamente. Teniendo en cuenta que durante la ejecución del código y, especialmente, en los tramos en los que se requiere un mayor número de recursos, los 8 núcleos trabajan (aunque cada uno se solicite y cese su actividad en momentos diferentes) esto parece indicar que el coste computacional que acarrea es perfectamente amortiguable. El dato de que ninguno de los núcleos deba emplearse al 100 % y que los ocho trabajen aproximadamente igual podría ser signo de que una eventual paralelización mediante el comando *parfor* raramente aportará mejoría. Quizá se pueda exprimir algo más al octavo núcleo, pero ciertamente el hecho de que no se le solicite más (igual que el hecho de que ninguno se tenga que emplear a más de un 60 – 70 %) parece mostrar que la ejecución del algoritmo puede verse bien cubierta de la forma actual.

Pese a ello, el primer objetivo era el de implementar un script basado en *parfor* y estudiar qué mejorías podía haber. El fin, tanto de este como del siguiente apartado, era el de desarrollar un código alfa, esto es, la mejor versión obtenida y que debería de ir soportada por los resultados del siguiente capítulo. Sin embargo, es preciso indicar que **con la CPU esto no ha sido posible**, debido a que el intento de implementación en cada bucle ha ido reportando dificultades en prácticamente todos ellos. Y en los pocos en los que ha sido posible, los resultados son extraordinariamente malos.

Por este motivo, se puede adelantar ya en este punto de la memoria, que **la paralelización usando la CPU no es una buena vía para el algoritmo de procesado de imágenes** que manejamos. Probablemente debido a su complejidad, pues el *parfor* es un comando que está mucho más indicado usar cuando los programas en los que se implementa son de corta extensión y con operaciones matriciales sencillas **[20], [24], [25]**.

Sin embargo, se va a comentar en qué líneas se ha logrado introducir para que, en el siguiente capítulo de resultados y discusiones, pueda comprobarse lo adversa que es su implementación en el código base, así como para tratar de averiguar finalmente qué sucede con los ocho núcleos de que dispone el ordenador. De todos modos, llegado a estas líneas, el lector ya puede intuir cuál ha sido el tipo de paralelización más exitoso (cosa que, por otro lado, quizá era esperable por la extensión del código, por su complejidad y por ser el nivel 2 un tipo de paralelizado considerablemente más profundo).

Sin más dilación, se pasa a comentar en qué puntos se ha implementado cada *parfor* (donde ha sido posible) dentro del script, así como su problemática:

Bucle 1: - No se puede emplear el tercer "truco" – visto en el apartado 3.4 del Capítulo 3 – en elementos que direccionan posiciones en matrices/vectores. Tampoco es posible emplear el comando ***break*** dentro de un *parfor* (ni un eventual ***return***). Con todo, se logra introducir uno en la línea 185.

```
parfor kk=1:length(Inoise(n,:))
                    Inoisesum_rep=Inoisesum_rep+double(Inoise{n,kk});
                end
```

Bucle 2: - Se implementa uno en la línea 226 y otro en la 235.

```
parfor i=1:size(Iback,2)*size(Iback,1)
                if ~isempty(Iback{i})
                    Ibacksum=Ibacksum+double(Iback{i}(:,:));
%sum1=sum1+double(Iback1{i}(:,:));
                    Iback_count=Iback_count+1;
                end
            end
```

```
parfor i=1:size(Inoise,2)*size(Inoise,1)
                if ~isempty(Inoise{i})
                    Inoisesum=Inoisesum+double(Inoise{i}(:,:));
%sum1=sum1+double(Iback1{i}(:,:));
                    Inoise_count=Inoise_count+1;
                end
            end
```

Bucle 3: - En principio, el analizador de código estático no arroja ningún error cuando se inserta un *parfor* en la línea 264, pero a la segunda vuelta del bucle *for* que lo contiene, el programa da un error desconocido (***parfor consume***). Por tanto, no se puede.

Bucle 4: - Exactamente mismo problema que en el anterior.

Bucle 5: - No es posible emplear el tercer "truco" para solventar la situación (como ocurría en el Bucle 1, pese a que ahora sí que se trata de matrices).

Bucle 6: - Se implementa uno en la línea 378 de forma aparentemente exitosa (no da errores). Es un bucle idóneo en el que hacerlo ya que, como se vio en la Figura 4.2, se trata de uno de los bucles que más tiempo cuesta ejecutar.

```
parfor j=NumBI+1:NumPerRep
                        %
Flamepos=find(TimeBLK(i,:)==TimeBLKord(i,j));
                        %             mask2=zeros(ysize,xsize);
                        %             mask2=Iflame1{i,Flamepos}>60;

                        %             diff=(double(Itotalint{i,j})*mask)-
(double(Iflame1{i,Flamepos}*mask2));
                        a=double(ItotalFINAL{i,j});
                        b=double(IflameFINAL{i,j});
%                                     c=double(b);
                        diff=double(a-b);
                        KL_rep{j}=log(complex((IbackavgFin./diff)));
                        KL_Axis_rep{j}=KL_rep{j}(1:70*pixmm,xinjector);
                        KLsat_rep{j}=log((IbackavgFin./diffSat).*mask);
%saturated KL value
                        KLsat_Axis_rep{j}=KLsat_rep{j}(1:70*pixmm,xinjector);


                    end
```

Bucle 7: - Se implementa uno en la línea 415.

```
parfor j= NumBI+1:NumPerRep;
                sum1 = zeros(ysize,xsize);
                sum2 = zeros(ysize,xsize);
                for r = 1:length(Rep2Process);
                    i=Rep2Process(r);
                    sum1 = sum1+double(ItotalFINAL{i,j}(:,:));
                    sum2 = sum2+double(IflameFINAL{i,j}(:,:));
                end
                Itotalsum{j} = sum1;
                Itotalavg{j} =
mask.*double(Itotalsum{j})./length(Rep2Process);%average total illumination
with LED and flame
                Iflamesum{j} = sum2;
                Iflameavg{j} =
double(Iflamesum{j})./length(Rep2Process);%average flame illumination
                mask2 = zeros(ysize,xsize);
                mask2 = Iflameavg{j}>60;
                Iflameavg{j} = Iflameavg{j}.*mask2;

                %calculate the KL of soot
                diff = double(Itotalavg{j})-double(Iflameavg{j});
                KL_avg{j}=log(complex(IbackavgFin)./diff);
                KLsat_avg{j}=log((IbackavgFin./diffSat).*mask);%saturated KL
value
                Time(j)= 1000000/Framerate*j;%unit [us]
            end
```

Bucle 8: - En principio se implementa uno en la línea 533 que, de hecho, rebaja en torno a un 20 % el tiempo que le cuesta ejecutarse (de 15 a 12 segundos). El problema radica en que dejan de aparecer los gráficos en pantalla (tanto el chorro combustionando como las propias gráficas cuando se descomentan para ver qué sucede).

Podría pensarse que tal vez se produzca un conflicto entre núcleos por ver cuál de ellos exactamente tiene que gestionar la aparición de cada gráfica (pues cabe recordar que en el comando *parfor* las acciones se realizan completamente independizadas, pudiendo aparecer una gráfica que va después antes que otra que debería precederla). Quizá todo esto la CPU no sepa bien cómo gestionarlo. Por tanto, la implementación no es válida ya que, directamente, le resta una funcionalidad vital al algoritmo. Éste deja de funcionar como corresponde, luego el paralelizado en este tramo carece de sentido.

Con todo, se ha conseguido nutrir al código con un total de **cinco parfor**. Será en el Capítulo 5 cuando se analicen los resultados que arroja, tanto en tiempo de cálculo como en coste computacional. Como ya se ha dicho, han sido muy malos.

4.3.2. Paralelizado en la GPU

Con ánimo de no perder de vista el esquema de pasos o etapas introducido en el apartado 4.1 del presente capítulo, conviene recordar que nos encontramos en el quinto paso para alcanzar el sexto. Esto es, el desarrollo de un **código o script alfa**.

En este orden de cosas, para programar en MatLab con ayuda de su *Parallel Computing Toolbox*, no se necesita más que lo que se ha aprendido y reflejado a lo largo del Capítulo 3. Dado que ahora son los hilos de la gráfica *NVidia* los que van a procesar los cálculos, al practicar un pre-estudio con el monitor de recursos para ver cómo se está comportando la GPU, se obtiene que la actividad de ésta es muy pequeña, pero no nula. Probablemente porque participa en procesos como el propio visionado de lo que aparece en pantalla.

Nota: - Esta vez no se introducirá la gráfica pertinente porque únicamente se trata de un registro y, con lo apuntado, se puede adquirir la idea general. No obstante, de nuevo en el Capítulo 5 se introducirán los estudios que permitan comparar cómo se comporta la carga de la GPU en ambos códigos: base y paralelizado.

Apuntado esto, es momento de ir introduciendo los fragmentos de código – por bucles, como antes – y describiendo lo que se ha hecho, cómo se ha efectuado y en base a qué fueron finalmente consolidados dichos bucles. Esto es, del mismo modo que se ha hecho anteriormente con la CPU, reflejar en la presente memoria la trazabilidad de la que se ha hecho uso para alcanzar el código óptimo.

Bucle 1

```
   tic;
for n=1:5
    data = GetMovie_paralelizada(0,n,'Photron','mraw',NumPerRep,Framerate);
    data = gpuArray(data);
    m=1;
    g=1;
    for c=1:NumBI
        if max(max(data(yinjector:yinjector+200,:,c)))>LEDcontrol
            Iback{n,m}=data(:,:,c);
            m=m+1;
        else
            Inoise{n,g}=data(:,:,c);
            g=g+1;
        end
    end
    if size(Iback,2)~=size(Inoise,2)
        if size(Iback,2)<size(Inoise,2)
            for d=NumBI+1:NumPerRep
                if max(max(data(yinjector:yinjector+200,:,d)))>LEDcontrol
                    Iback{n,m}=data(:,:,d);
                    break
                end
            end
        else
             for d=NumBI+1:NumPerRep
                if max(max(data(yinjector:yinjector+200,:,d)))<LEDcontrol
                    Inoise{n,size(Inoise(n,:),2)}=data(:,:,d);
                    break
                end
             end
        end
    end
    %% Combustion Detector
    Inoisesum_rep=gpuArray(zeros(ysize,xsize));
%     Inoisesum_rep=gpuArray(Inoisesum_rep);
    for kk=1:length(Inoise(n,:))
        Inoisesum_rep=Inoisesum_rep+double(Inoise{n,kk});
    end
    Inoiseavg_rep=Inoisesum_rep./length(Inoise(n,:));
    InoiseRef_rep=max(max(Inoiseavg_rep(ysize/2:end,:)));
    if InoiseRef_rep==0
        InoiseRef_rep=minimum_noise;
    end
    for j=NumBI:NumPerRep
        if max(max(data(yinjector:yinjector+200,:,j)))<LEDcontrol
            if max(max(data(ysize/2:end,:,j)))>InoiseRef_rep*FlameControl;
                FireRepCount(n)=1;
                break
            end
        end
    end
end
```

Las líneas paralelizadas se muestran en naranja y han sido dos. Una de ellas ha consistido únicamente en definir una matriz de ceros, con dimensiones las de las capturas de imagen de la cámara. La otra es una ventaja muy importante pues, si volvemos a la Figura 4.1, se puede ver que la función ***GetMovie*** original es, ni más ni menos, la segunda de las funciones que más tiempo de cálculo conlleva dentro del código base.

La función *GetMovie* es una función personalizada que fue creada en *el CMT – Motores Térmicos*. Es ampliamente utilizada cada vez que se llevan a cabo ensayos en los que hay que procesar imágenes procedentes de alguna cámara rápida (en este script se procesan imágenes que fueron tomadas con la cámara rápida *Photron*, pero existen otras).

Por tanto, la mejora que aporta la nueva función ***GetMovie_paralelizada*** es algo que se aprovechará de forma permanente, por parte de aquél que desee procesar imágenes y su ordenador de trabajo esté dotado de una GPU que permita cálculos en paralelo. Se muestra a continuación un pequeño extracto de la función (no toda) en la que se realizó la modificación que ha permitido optimizarla haciendo uso de los hilos de la GPU.

```
case 'photron'

 % patch added to read mraw movies
      if strcmpi(input.ImageExtension,'mraw') %%
         Nameraw=getarchivos('*.mraw'); %De momento solo funciona para un solo archivo
de pelÌcula por carpeta. %%
         Nameraw=Nameraw{1}(1:end-5); % Hay que quitar la extensiÛn para imreadraw. %%
         Imgs = imreadmraw_paralelizada(Nameraw,[ImgNums(1)+input.NumPerRep*(RepNum-
1),ImgNums(end)+input.NumPerRep*(RepNum-1)],input)/16;%/16; % /16 to level values to 12
bit range
         %        figure
%           imshow(Imgs(:,:,end));

      else %%   % end of patch
```

Lo que se ha hecho ha sido mejorar a su vez otra nueva función, creada manualmente, llamada ***imreadmraw***, que también ha sido sustituida por su versión paralelizada, como se aprecia en el resalto en naranja. Cabe recordar que, si la anterior constituía la segunda función más costosa de ejecutar de todo el código, esta es la tercera. La Tabla 4.1 muestra los resultados arrojados por un rápido cálculo estadístico, extraído de ejecutar cuarenta veces un bucle con la instrucción correspondiente, de modo que se recojan los valores del tiempo que ha costado ejecutar esa línea y los incluya en un vector de cuarenta elementos que, posteriormente, se guarda.

Medidas GetMovie (s)				Medidas de tiempo en GetMovie_paralelizada (s)			
0,67020	0,63337	0,62149	0,61068	0,54132	0,51442	0,52904	0,52908
0,62278	0,62611	0,60952	0,61555	0,53977	0,52195	0,53630	0,53081
0,61986	0,62341	0,65202	0,62033	0,51793	0,51668	0,52618	0,52200
0,62247	0,62530	0,61333	0,62406	0,53050	0,51762	0,53249	0,52736
0,63813	0,61212	0,63186	0,60943	0,54523	0,51283	0,52560	0,52180
0,62217	0,62561	0,62185	0,60606	0,53076	0,52908	0,52542	0,52657
0,62172	0,62056	0,62142	0,60897	0,52754	0,52730	0,52419	0,52126
0,61520	0,61713	0,61436	0,61072	0,53547	0,53067	0,53200	0,52955
0,62305	0,62633	0,62024	0,61172	0,52741	0,52856	0,52131	0,52714
0,62118	0,62006	0,62080	0,62391	0,52485	0,52827	0,53636	0,52778

Promedio	**0,62188**	**0,52751**
Desv. St.	0,01159	0,00692
Mejora (%)	**15,17440**	

Tabla I. Resultados arrojados por el cálculo rápido para las funciones GetMovie.

En la Figura 4.1 se vio que la función *GetMovie* original costaba de ejecutar, efectivamente, en torno a 0,6 segundos. Por tanto, una mejora de más del 15 % es algo considerable y de agradecer, pues además de que es una función empleada en más de un bucle, las ganancias se espera que vayan "in crescendo" cuando se aumenten el número de repeticiones.

Bucle 2

No ha sido paralelizado porque se trata de un bucle que dura unas centésimas de segundo (ver *Anexo B* y Figura 4.2).

Bucle 3

```
for r=1:length(Rep2Process)
                n=Rep2Process(r);
                data =
GetMovie_paralelizada(0,n,'Photron','mraw',NumPerRep,Framerate);
                data=gather(data);
```

Se ha incorporado únicamente la parte del bucle que se ha paralelizado (ver *Anexo A*). De nuevo, la mejora ha venido de la mano de la nueva función *GetMovie_paralelizada*. Como se puede apreciar, a diferencia del Bucle 1, ahora ha sido preciso devolver la variable *data* a la memoria CPU tras cada iteración en el bucle *for*. El motivo radica en que, de lo contrario, la consola hacía saltar un error indicando que la memoria video de la GPU había sido desbordada.

Es importante apuntar que, la mejora que reporta este bucle (como se verá en el capítulo de resultados) es menor que en el anterior, fruto de que se pierde un tiempo en cada vuelta de bucle por realizar una transferencia que antes no hacía falta con la función *GetMovie* original. Pero era necesario para evitar que la memoria video de la GPU colapsara.

Bucle 4

Este bucle ha resultado imposible de paralelizar. Por una parte lo que se ha probado con los *TimeLED* y *TimeBLK* no ha dado frutos. Y por otra, los *Itotalint* e *Iflameint* son de clase ***cells*** (direccionan a posiciones de matrices y vectores) y no está permitido su categorización como objetos *gpuArray*.

De todos modos, si se echa un vistazo al *Anexo B* y a la Figura 4.2 puede asumirse que no es algo realmente grave, puesto que se trata del antepenúltimo bucle en importancia, siendo para el algoritmo de cinco repeticiones, apenas medio segundo de cálculo.

Bucle 5

Este bucle, como ocurría con el 2, no tiene sentido paralelizarlo, dado que su tiempo de ejecución es muy pequeño. Por tanto, se decidió dejarlo como ya estaba.

Bucle 6

Se trata del bucle en el que más mejora se consiguió (hasta un 40 %) y una de las mayores claves del éxito de este proyecto, ya que en la Figura 4.1 se destapa como de los más importantes. De hecho (como se verá en el siguiente apartado) es el más costoso.

```
diffSat=gpuArray(diffSat);
for r=1:length(Rep2Process)
                    i=Rep2Process(r);
                    for j=NumBI+1:NumPerRep
                        %             Flamepos=find(TimeBLK(i,:)==TimeBLKord(i,j));
                        %             mask2=zeros(ysize,xsize);
                        %             mask2=Iflame1{i,Flamepos}>60;

                        %             diff=(double(Itotalint{i,j})*mask)-
                       (double(Iflame1{i,Flamepos}*mask2));
                        a=gpuArray(double(ItotalFINAL{i,j}));
%                         a=gpuArray(a);
                        b=gpuArray(double(IflameFINAL{i,j}));
%                         b=gpuArray(b);
%                                     c=double(b);
                        diff=double(bsxfun(@minus,a,b));
                        KL_rep{j}=log(complex((IbackavgFin./diff)));
                        KL_Axis_rep{j}=KL_rep{j}(1:70*pixmm,xinjector);
                        KLsat_rep{j}=log((IbackavgFin./diffSat).*mask); %saturated KL
value
                        KLsat_Axis_rep{j}=KLsat_rep{j}(1:70*pixmm,xinjector);


                    end
                     %save(strcat('KL_LEI_Rep_',num2str(i),'.mat'),'KL_rep',
'KLsat_rep', 'KL_Axis_rep', 'KLsat_Axis_rep', '-v7.3');
                    clearvars KL_rep KLsat_rep KL_Axis_rep KLsat_Axis_rep;

  end
```

Puede verse como, con apenas tres líneas a modificar dentro del bucle se ha podido obtener una mejora muy grande. Esto ha sido gracias a que el *Parallel Computing Toolbox* tiene implementadas funciones como ***double***, ***bsxfun***, ***minus***, ***log*** o ***complex***, que permiten que las operaciones en las que se tengan como variables a al menos una que sea de clase *gpuArray* basta para poder extender el cálculo en los hilos de la GPU. Cabe observar que fue necesario introducir una nueva línea para declarar en memoria de la GPU a una variable (fuera).

Bucle 7

```
    for j= NumBI+1:NumPerRep;
                    sum1 = zeros(ysize,xsize);
                sum2 = zeros(ysize,xsize);
                for r = 1:length(Rep2Process);
                    i=Rep2Process(r);
                    sum1 = sum1+double(ItotalFINAL{i,j}(:,:));
                    sum2 = sum2+double(IflameFINAL{i,j}(:,:));
                end
                Itotalsum{j} = sum1;
                Itotalavg{j} = mask.*double(Itotalsum{j})./length(Rep2Process);%average
total illumination with LED and flame
                Iflamesum{j} = sum2;
                Iflameavg{j} = double(Iflamesum{j})./length(Rep2Process);%average flame
illumination
                mask2 = zeros(ysize,xsize);
                mask2 = Iflameavg{j}>60;
                Iflameavg{j} = Iflameavg{j}.*mask2;
                mask=gather(mask);

                %calculate the KL of soot
                diff = gpuArray(double(Itotalavg{j})-double(Iflameavg{j}));
%                             diff=gpuArray(diff);
                KL_avg{j}=log(complex(IbackavgFin)./diff);
                KLsat_avg{j}=log((IbackavgFin./diffSat).*mask);%saturated KL value
                Time(j)= 1000000/Framerate*j;%unit [us]
            end
```

En esta ocasión, con tan solo un par de líneas modificadas ha sido posible obtener mejoras que, al menos en el código de cinco repeticiones (el que se ha usado para realizar todo el proceso de optimización), ha oscilado entre el 18 y el 27 % de mejora.

Bucle 8

```
for j= NumBI+1:NumPerRep
                          %PLOTS

                          h1=figure(1);
                          KLAVG=KL_avg{j};
                          KLAVG=single(KLAVG);

                          KLrot{j}=imrotate(KLAVG,90);
%                           KLrot{j}=imrotate(KL_avg{j},90);
                          imshow( KLrot{j}(((xsize-xinjector)-
round(12*pixmm)):((xsize-
xinjector)+round(12*pixmm)),yinjector:round(70*pixmm)),[0,3.5]);
                          KLrot{j}=gather(KLrot{j});
  end
```

En este último bucle, la estrategia consistió en comentar el comando ***imrotate***, que inicialmente operaba sobre las imágenes promediadas de la celda *KL_avg{j}* – el cual sirve para rotar las imágenes 90 grados – e implementar varias líneas nuevas, introduciendo una nueva variable denominada *KLAVG*.

La razón la encontramos en que, la variable *KL_avg*, que como se ha dicho es una celda (direcciona posiciones), es de clase *gpuArray* debido a tal y como fue obtenida en el Bucle 7. Lo más intrigante es que este tipo de variables no son retornables a la memoria de la CPU una vez se han generado de forma directa en la GPU, pues formalmente no son matrices. Por ello, la acción que se propuso fue la de crear esa nueva variable *KLAVG* que MatLab la guarda de una forma que podemos aplicarle el comando ***single*** (recogido en el Toolbox, tal y como muestra la Figura 3.13). De esta forma, se le pueden aplicar ya los comandos ***imrotate*** e ***imshow*** a un objeto gpuArray, aunque no estén contempladas en la Figura 3.13. Finalmente se introduce una línea con el comando *gather* para que la variable *KLrot{j}* retorne a la memoria GPU, pues fue la única manera de evitar que la memoria video de la GPU colapsara.

Antes de cerrar este apartado que ha dado paso a la obtención del código alfa, merece la pena realizar dos apuntes: el primero, notar que la función *imshow* es, nada más y nada menos, que la función que más tiempo de cálculo consume, de acuerdo con lo visto en la Figura 4.1. Lo segundo, recordar de nuevo que el fragmento del Bucle 8 que sirve para configurar las gráficas de KL que saca MatLab está comentado (no forma parte del bucle) porque requiere de un tiempo que no es insignificante y no se puede paralelizar ninguna línea. Por ese motivo se prefirió trabajar, tanto con el código base como con los candidatos a código alfa, con este tramo inactivo (se pueden apreciar igual las mejoras a nivel relativo).

Con todo, se ha visto como el código alfa, desarrollado mediante programación heterogénea (parte en la CPU parte en la GPU) se sostiene en buena parte gracias a la mejora de las tres funciones más pesadas en el algoritmo (representan un 66 % en la ejecución de un caso, como se puede apreciar en la Figura 4.1).

4.4. Comparativa CPU/GPU

Lo primero que cabe decir para evitar posibles confusiones es que, esta comparativa que se planifica ahora (de cara a la obtención de los resultados que se publicarán en el Capítulo 5) no son entre el paralelizado efectuado en la CPU y el implementado en la GPU.

Tal y como ya se ha apuntado con anterioridad, el código "alfa" obtenido para la CPU no merece tal denominación, puesto que ha sido mantenido única y exclusivamente para mostrar dos cosas. Una la de comprobar que, efectivamente, es una mala vía para tratar a este algoritmo en particular[53]. La otra, es que todavía puede merecer la pena estudiar qué sucede con el coste computacional.

Como se introdujo al inicio de esta memoria, en el apartado de objetivos del Capítulo 1, la variable crítica que se persigue optimizar no es el coste sino el tiempo de cálculo computacional. Pero, saber cómo reaccionan los núcleos con el código paralelizado, puede resultar interesante ante eventuales situaciones en las que prime más tener una mayor cantidad de recursos disponibles en el ordenador de trabajo, para poder hacer correr a otras aplicaciones, pese a que ello implique un mayor tiempo de procesado (como todo, es cuestión de sopesar, del experimento y de la circunstancia).

4.4.1. Estudios paramétricos

Para cerrar este cuarto capítulo, se va a proceder a establecer la estrategia a seguir para la obtención de una serie de estudios paramétricos que puedan arrojar luz acerca del comportamiento del código alfa – obtenido y explicado en el anterior apartado – frente a distintas variaciones, tratando de compararlo en la medida de lo posible con lo que representa en su estado natural el código base con el que se tuvo que trabajar a lo largo del proyecto.

4.4.1.1. Sensibilidad frente a número de repeticiones

Este estudio paramétrico va a consistir en elaborar un bucle, en el script de *MatLab* del código alfa (el paralelizado), que permita recoger cuarenta muestras del tiempo que cuesta ejecutarse a cada uno de los ocho bucles, así como el total del código, haciendo variar el número de repeticiones de cinco en cinco (hasta 30, el máximo) con las que el Bucle 1 adquiere las imágenes de los ficheros (tomadas por la cámara rápida *Photron*) y las va guardando en la matriz tridimensional *data*.

[53] *No significa que paralelizar mediante el comando parfor sea nocivo, en otros casos más simples sí funciona.*

De este modo, se espera **elaborar un total de 6 tablas** (una por serie) que incluyan los citados tiempos para llevar a cabo, a posteriori, los análisis estadísticos oportunos que permitan analizar el comportamiento observado, así como la inclusión de gráficas que hagan más visuales los resultados obtenidos. De ahí se espera poder sacar conclusiones confiables, amparadas en la robustez estadística que confiere el obtener más de 30 medidas[54].

A continuación mostramos pequeños fragmentos explicativos de cómo se han implementado los bucles que permiten la obtención de la totalidad de las muestras en las 40 ejecuciones del código necesarias para cada serie.

Primero se han inicializado una serie de vectores, de una fila y cuarenta columnas, en los que serán introducidos cada uno de los valores de tiempo asociados a cada bucle en cada nueva ejecución del script al completo:

```
muestras=40;
tiempo_Bucle1=ones(1,muestras);
tiempo_Bucle2=ones(1,muestras);
tiempo_Bucle3=ones(1,muestras);
tiempo_Bucle4=ones(1,muestras);
tiempo_Bucle5=ones(1,muestras);
tiempo_Bucle6=ones(1,muestras);
tiempo_Bucle7=ones(1,muestras);
tiempo_Bucle8=ones(1,muestras);
tiempo_calculo_total=ones(1,muestras);
```

A continuación, justo antes de que se inicie el bucle de casos – "capado" para que solo ejecute un único caso – se introduce un bucle al que se le ordena dar cuarenta vueltas:

```
for x=3:length(date)
      date_path=strcat(mainpath, '\', date(x).name);
      cd(strcat(date_path,'\Photron'))
      cases=dir;
      for k=1:muestras
    tic;
     for t=3:3 (bucle que carga cada caso)
```

Se ejecuta el resto del script de la misma forma que habitualmente, solo que al terminar cada bucle, se le agrega una sentencia para que vaya guardando cada resultado en el correspondiente vector ya inicializado. Por ejemplo, para el primer bucle quedaría como sigue:

```
Bucle1=toc;
tiempo_Bucle1(k)=Bucle1;
```

---

[54] *Permite por ejemplo el uso, en análisis discriminante, de la prueba t de student, en la que el estadístico sigue esta conocida distribución cuando la hipótesis nula es cierta. Se aplica cuando se sabe que la población sigue una distribución normal pero el tamaño muestral es demasiado pequeño como para suponer que el estadístico en el que se basa la inferencia sigue dicha distribución. De este modo, la prueba t de student permite emplear la desviación típica en lugar del valor real.*

Finalmente, no hay más que guardar todos los vectores rellenos en un fichero ***.mat***:

```
        save('Tiempos_5Rep.mat','tiempo_Bucle1','tiempo_Bucle2',
'tiempo_Bucle3','tiempo_Bucle4','tiempo_Bucle4','tiempo_Bucle5',
'tiempo_Bucle6','tiempo_Bucle7','tiempo_Bucle8','tiempo_calculo_total');
```

Este estudio interesa para averiguar si podría ocurrir algún efecto positivo en el paralelizado que permitiera que, con más cantidad de datos adquiridos, la GPU acelerara su velocidad de cálculo con respecto a la CPU. [*J.Straus*], **[20]** muestra que cuantos más bytes ocupan las variables transferidas de la CPU a la GPU, exponencialmente más se acelera el cálculo. Esto es algo que puede apreciarse en la Figura 4.4. Se aprecia como a partir de un cierto valor de elementos en la matriz (evaluado con el comando ***numel***) la velocidad de cálculo en la GPU se empieza a disparar, en contraposición con la de la CPU, que esencialmente se mantiene constante.

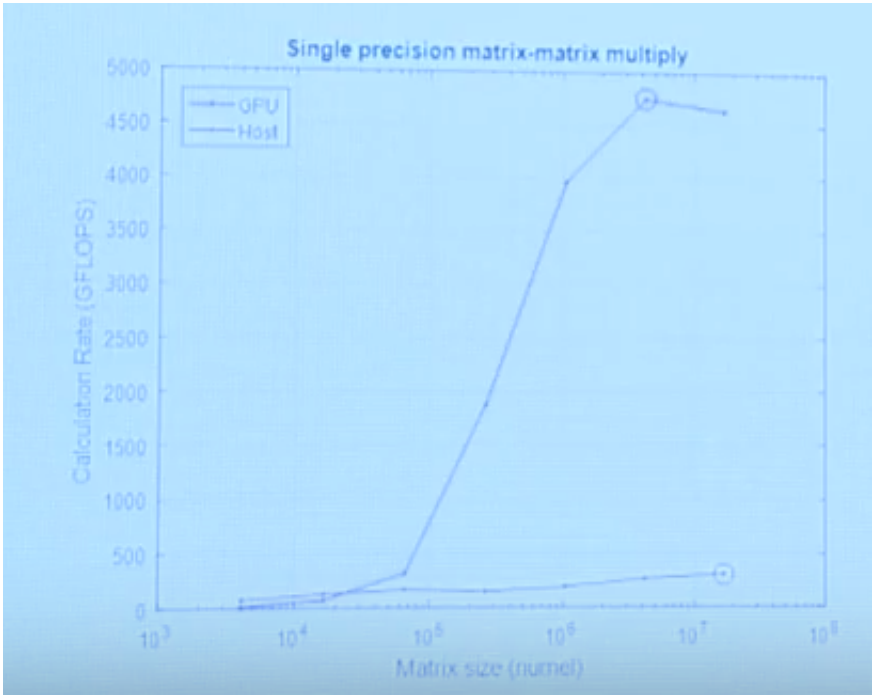


Figura 4.4. Variación de la tasa de cálculo en GPU y CPU con el tamaño de la matriz.

4.4.1.2. Sinergias en el paralelizado de bucles internos

El planteamiento de este estudio es muy sencillo. Una vez que el anterior estudio haya arrojado luz acerca de cómo se comporta nuestro código alfa cuando se le somete a incrementos en el número de las repeticiones que tiene que efectuar en el cargado de las imágenes – en lugar de las cinco con las que se ha desarrollado y trabajado desde el inicio – se elegirá aquél código más optimizado para realizar un estudio en el cual dilucidar si los tiempos de cálculo de cada bucle en el seno del script (ejecutado en continuo) difieren significativamente de los que se obtienen cuando se ejecutan individualmente por separado.

De nuevo, aquí la contribución de la estadística jugará un papel preponderante, pues obteniendo cuarenta muestras de tiempo por cada bucle ejecutado de forma aislada, se podrán inferir conclusiones que permitan explicar lo que pueda estar ocurriendo. La

construcción de los vectores que contienen los valores de tiempo en cada bucle se lleva a cabo de una forma similar a la expuesta en el anterior punto.

En lo que se refiere a los tiempos de cada bucle cuando éstos han sido ejecutados en continuo, serán los que ya se hayan obtenido en el punto anterior (para la serie que corresponda, esto es, la de 5 repeticiones, 10, 15, 20, 25 o 30, pues no será necesario el estudio en distintas, ya que lo que ocurra en una será extrapolable a lo que ocurriría en otra con un diferente número de repeticiones).

En este sentido, lo que se espera obtener son **sendas tablas** (una conteniendo los tiempos de cada bucle en continuo y otra en aislamiento) para poder comparar los promedios y graficar aquellos resultados que puedan ilustrar lo que ocurre.

4.4.1.3. Sensibilidad al tipo de paralelizado

Este será el estudio que ponga punto final al apartado de resultados y que dé paso a la formulación sobre cuál de los dos niveles de paralelizado ha resultado ser más eficaz y eficiente. En este caso, se tratará de realizar una comparativa entre los resultados arrojados por la CPU y los arrojados por la GPU, tanto en términos de tiempo de cálculo como de coste computacional.

Nota: - Anteriormente se habrán evaluado los resultados obtenidos en los respectivos códigos alfa – desarrollados en el apartado 4.3 de este capítulo – también tanto en términos del tiempo como del coste, comparándolos con los que exhibe el código base. Esto es algo que merece la pena tener en consideración, pues los estudios paramétricos, por su naturaleza, no pueden tener cabida antes de la propia exposición de los resultados que arroja cada tipo de código manipulado (base, paralelizado por CPU y paralelizado por GPU, respectivamente).

5. RESULTADOS EXPERIMENTALES

5.1. Introducción

En este capítulo se van a presentar los resultados correspondientes a los estudios realizados en este trabajo, describiendo y justificando cada aspecto convenientemente. El orden de presentación de los mismos se va a efectuar siguiendo el orden lógico y cronológico con el que fueron planteadas las experiencias y desarrollos a los cuales van asociados.

En los siguientes tres sub-apartados se van a mostrar los resultados obtenidos para cada una de las series de números de repeticiones. De este modo, se tendrán ya expuestos todos los datos directamente vinculados, tanto al código original, como a los paralelizados CPU y GPU, de manera que puedan formularse unas primeras conclusiones en vista de su evolución. El propósito principal es el de tenerlos recogidos todos, con el fin de poder llevar a cabo los estudios paramétricos que se abordarán ya en el último apartado de este capítulo.

5.2. Código base

5.2.1. Tiempo de cálculo

La necesidad de explorar cuellos de botella, que mostraran criterios de prioridad a la hora de elegir los caminos más prometedores para el establecimiento de un código alfa, llevó a la obtención de resultados de tiempo para el código configurado con cinco repeticiones. La *Tabla 5.1* contiene también los obtenidos para el resto de series. Notar que la suma de los tiempos de cada bucle constituye el 97,5 % del tiempo total, por ese motivo no coincide con el medido.

5 Repeticiones		**10 Repeticiones**		**15 Repeticiones**	
Bucle	**Tiempo (s)**	**Bucle**	**Tiempo (s)**	**Bucle**	**Tiempo (s)**
1	3,208	**1**	6,256	**1**	9,314
2	0,010	**2**	0,014	**2**	0,020
3	3,463	**3**	6,682	**3**	10,157
4	0,491	**4**	0,911	**4**	1,372
5	0,016	**5**	0,036	**5**	0,061
6	10,447	**6**	20,265	**6**	30,408
7	3,054	**7**	3,723	**7**	4,259
8	14,851	**8**	15,201	**8**	15,057
Total	**36,452**	**Total**	**54,449**	**Total**	**72,459**
Desv.St	0,318	Desv.St	1,217	Desv.St	1,049
20 Repeticiones		**25 Repeticiones**		**30 Repeticiones**	
Bucle	**Tiempo (s)**	**Bucle**	**Tiempo (s)**	**Bucle**	**Tiempo (s)**
1	12,538	**1**	16,043	**1**	18,924
2	0,026	**2**	0,030	**2**	0,035
3	13,812	**3**	17,318	**3**	20,829
4	1,857	**4**	2,445	**4**	3,015
5	0,096	**5**	0,143	**5**	0,167
6	41,222	**6**	51,519	**6**	61,705
7	5,169	**7**	6,219	**7**	6,976
8	15,113	**8**	15,570	**8**	15,415
Total	**92,135**	**Total**	**112,090**	**Total**	**130,324**
Desv.St	1,127	Desv.St	2,533	Desv.St	2,831

Tabla 5.1. Resultados de los tiempos de ejecución para el código base en función de RepNum.

Únicamente visualizando los resultados, en términos de tiempo de cálculo para el código base, resulta un poco prematuro realizar un análisis que conduzca a la formulación de conclusiones. Los resultados de tiempo pueden ser (y de hecho lo han sido) determinantes de cara a elegir un camino u otro en la paralelización. Pero, por si solos, únicamente constituyen características intrínsecas al código original, tal y como éste se empezó a utilizar. En este sentido, lo razonable es que, los resultados presentados a lo largo de este apartado, sirvan como presentación de lo que es el propio código base, dejando para posteriores los análisis más detallados, junto con el planteamiento de las discusiones que conducirán a la inferencia de dichas conclusiones. No obstante, viendo la Tabla I llaman la atención un par de aspectos:

• **Todos los bucles, excepto el octavo, se ven a afectados por el aumento de las repeticiones:** - Así, el bucle 6 ya es el más representativo en tiempo a partir de la serie con 10 repeticiones.

• **Las dispersión de los datos se dispara con 25 y 30 repeticiones:** - La serie de cinco apenas presenta dispersión, mientras que las de quince y veinte mantienen un valor comparable.

Las dispersiones en las series con un mayor número de repeticiones (mayor tiempo total por tanto) son las que tienen un valor más alto en términos absolutos. Sin embargo, lo que interesa es evaluar los porcentajes que representan con respecto al promedio. Así, se puede apreciar como se produce un salto importante entre la primera y la segunda serie. También entre la cuarta y la quinta. En contrapartida, de la segunda a la tercera y, de ésta a la cuarta, se produce una mejoría. Estos hechos denotan que, en general, en los algoritmos con mayor número de repeticiones, el tiempo de cálculo total está sujeto a una mayor fluctuación de los bucles y líneas de código que lo componen, dando como respuesta una mayor variabilidad en las ejecuciones entre una vuelta y la siguiente.

Por otro lado, el hecho de que a más repeticiones se tengan tiempos de ejecución más elevados es muy alentador, gracias a que el código optimizado ha sido el correspondiente a un número de repeticiones de cinco. Por tanto, las mejoras obtenidas en él (en porcentaje) van a ser arrastradas para las series con un número de repeticiones más alto, lo cual repercute en unos tiempos en valor absoluto cada vez más notorios y provechosos. Especialmente interesante es el hecho de que, el bucle que más se ha conseguido mejorar (el Bucle 6), sea el que empieza a predominar a partir de las 10 repeticiones. De todo esto se hablará más adelante con detenimiento, justificando cada explicación que se aporte con datos obtenidos experimentalmente[55] .

[55] *Los experimentos en este proyecto son de carácter informático. Ello quiere decir que los datos se obtienen mediante ensayos en los algoritmos obtenidos a partir de la implementación de las herramientas de paralelizado sobre el código base original.*

5.2.2. Coste computacional

Este análisis se introdujo, en parte, durante la exposición de la trazabilidad asociada al código alfa del paralelizado CPU. Las gráficas que exhiben el comportamiento de los ocho núcleos del procesador fueron insertadas y analizadas ya convenientemente en la Figura 4.3[56].

A continuación se exponen los resultados – tratados previamente mediante la aplicación *Excel* – que arroja el monitor de recursos, sobre el algoritmo para cada serie (distinto número de repeticiones). De este modo, se puede estudiar cómo se comportan de forma natural la *CPU* (total), la *GPU*, la *RAM* y la *memoria física*, con objeto de tener ya recogido, en este primer apartado, todo lo que concierne a las características inherentes al código base. Las gráficas obtenidas servirán para el apartado 5.5, de cara a realizar el estudio comparativo frente a la GPU. Los resultados arrojados por *afterburner* se muestran en las Figura 5.1 y 5.2.

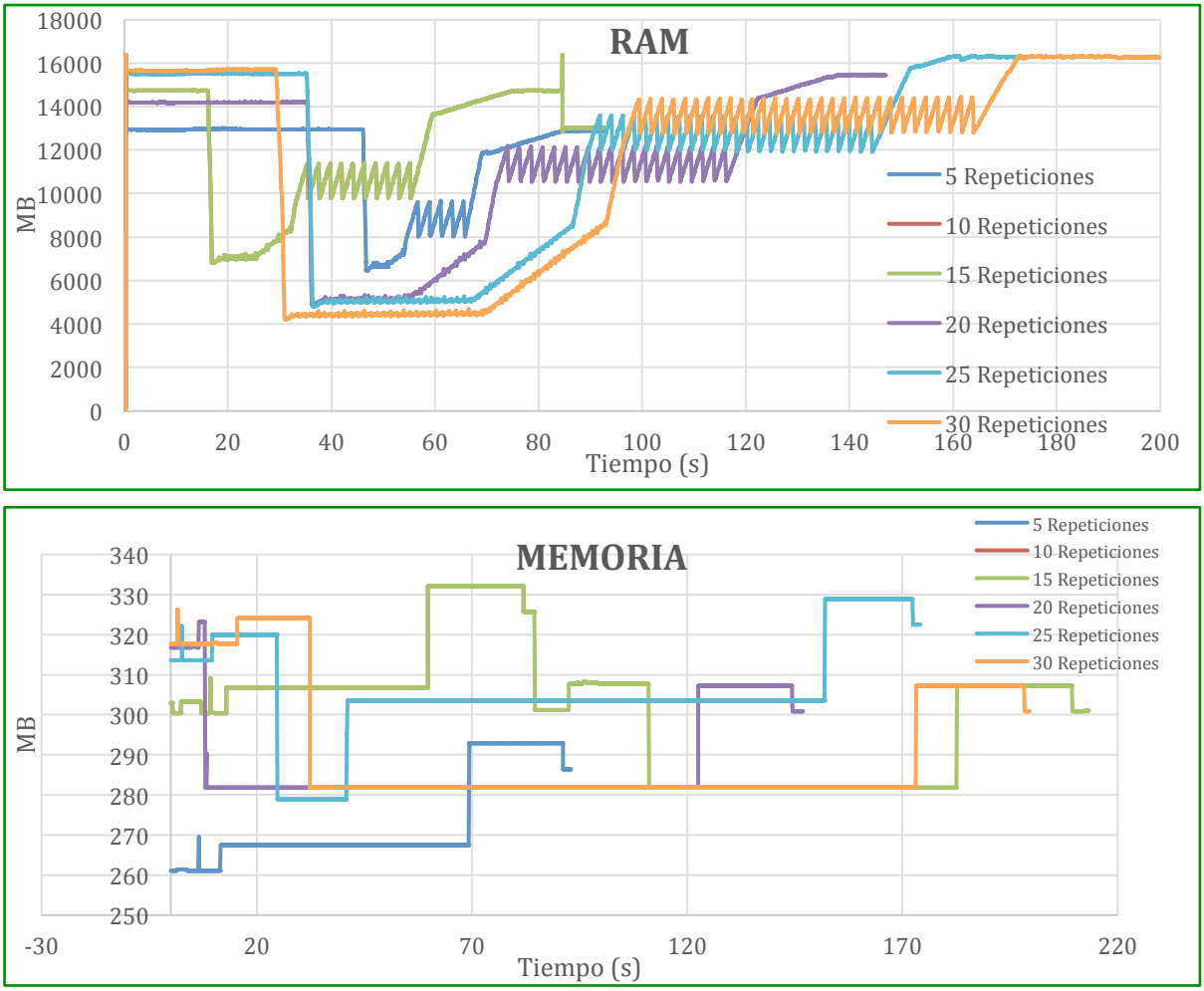


Figura. 5.1. Gráficas frente al tiempo para: a) RAM, b) Memoria física.

[56] *Formalmente no constituían un resultado. De hecho, no se manejaron cifras. Lo que se perseguía en aquél punto de la memoria era mostrar de qué manera se podría prever una hipotética mejoría a la hora de implementar una paralelización, mediante el uso del comando parfor. Dado que el pre-análisis no arrojaba resultados prometedores, el paralelizado por esa vía se encaró ya sabiendo que las posibilidades de mejora eran mínimas.*

Figura. 5.2. Gráficas frente al tiempo para: a) GPU, b) CPU.

Las cuatro gráficas han sido obtenidas a partir de los datos del cuaderno que aporta la aplicación de monitor de recursos *afterburner* y reconstruyendo las señales en una hoja *excel*. En este apartado se va a comentar, a nivel cualitativo, el comportamiento observado de los cuatro elementos de estudio en el ordenador, así como tratar de examinar qué diferencias se observan entre series. El estudio "fino" y numérico se deja para el apartado *5.5. Comparativa CPU/GPU*, donde se indicarán también las partes de las señales que corresponden a cada bucle. Se arrojarán datos sobre áreas bajo las curvas como medida aproximativa del valor promedio de cada señal, así como se procederá a un restado de señales, procedente de los códigos paralelizado y sin paralelizar, que trate de evidenciar a nivel básico cuáles son las diferencias más significativas que se observan en los comportamientos de uno y otro código.

Pasando a comentar ya lo que se aprecian en las figuras, si se observan las dos gráficas que se recogen en la Figura 5.1, se pueden realizar los siguientes análisis:

RAM

• Aparece un primer tramo en el que, con carácter general, todas las señales decaen bruscamente a valores tanto más pequeños cuanto mayor es el número de repeticiones. Esto se puede achacar al uso de RAM que, lógicamente, cuanto más largo y cargado es el script, más se solicita por parte de la CPU la creación de este tipo de memoria de origen rápido y

aleatorio. Como se vio en la Figura 3.8, el ordenador tiene 16384 MB de RAM utilizables. Por eso en la gráfica se aprecia a cuánto disminuye la disponibilidad para otros procesos, en el momento en el que se inicia el código (al darle a "Run"). Incluso para el caso de 5 repeticiones, el valor decae a menos de la mitad.

• Con el script ya empezado a ejecutar, poco a poco la memoria RAM va recuperándose con una rampa de igual pendiente para todas las series, lo que muestra que se están ejecutando las mismas líneas de código (por tanto, las señales adoptan formas equivalentes). La mayor diferencia se halla en que cada pendiente tiene una longitud distinta siendo, naturalmente, las señales asociadas a códigos con mayor número de repeticiones las mayores.

• Transcurrido el bucle o bucles a los que se asocia esa pendiente, las señales entran en una fase en la que adoptan una forma de dientes de sierra, sufriendo continuos ciclos de carga y descarga. De nuevo, las mayores longitudes se encuentran para las series de más repeticiones. Esto indica que todavía se encuentran ejecutando partes del código previos al último bucle (el Bucle 8 [57]).

• El último tramo es una pendiente de subida, la cual se puede atribuir ya al último bucle, pues se aprecia que las longitudes son las mismas en las cuatro series, tal como era de esperar. El hecho de que durante la ejecución de éste, se empiece a recuperar linealmente, parece mostrar que se trata de un tramo de código que, pese a tener una duración importante, no requiere de mucha RAM (tiene bastante sentido, ya que es un bucle de "plots" y no se guardan variables).

Nota: - No hay que prestar demasiada atención al momento en que se inician unas señales con respecto a otras. Este desfase temporal se debe a que cada una ha sido tomada abriendo *afterburner* de forma independiente cada vez. En este contexto, los inicios de las ejecuciones del script se pueden producir con unos pocos segundos de diferencia de unas a otras (primero se abre *afterburner*, después *MatLab* y, cuando está todo preparado, se pulsa "Run").

Como conclusión del análisis de la RAM puede decirse que, el código base en estado natural, ya constituye un proceso muy relevante dentro de las tareas en las que ésta puede colaborar. Tanto es así que, si eventualmente se abriera *MatLab* por duplicado en dos ventanas diferentes, y se ejecutara el algoritmo para únicamente 5 repeticiones, la memoria RAM colapsaría. En este sentido, cabe esperar que el paralelizado la descongestione bastante.

Memoria física del ordenador

• La señal empieza con un pequeño pico para cada señal. Cuanto mayor es el número de repeticiones, en general la memoria reservada para la ejecución del código se inicia desde un valor más alto (como excepción, la de 25 lo hace a un valor más bajo que la de 20).

[57] *Tal como se vio en la Tabla 5.1, el tiempo de cálculo que lleva asociado no cambia significativamente.*

• Reservada la memoria, todas las señales entran en otra etapa en la que se produce una subida en forma de escalón (abrupta) y que es de la misma magnitud. Pese a que se aprecia una variabilidad en el comportamiento para las siguientes etapas (unas decaen y se recuperan y otras no o que, señales como la naranja y la verde, decaen hasta el mismo valor antes de enfilar la recta) todas tienen en común el aspecto de "garfio" con el que terminan. Esa subida que da entrada a la forma de "garfio" debe corresponderse con el bucle 8 puesto que, de nuevo, se observa que las longitudes son equivalentes. Puede notarse que la señal verde parece como si hubiera participado en dos ciclos consecutivos (dos ejecuciones del script) ya que, de otro modo, se hace difícil explicar su comportamiento diferente.

Con todo, en este caso se puede concluir lo que, de forma esencial, se aprecia en la gráfica: - La reserva de memoria para llevar a cabo la ejecución difiere considerablemente entre las primeras series, siendo mucho menor la diferencia entre las tres últimas. Ello parece mostrar que, la unidad central de cálculo, requiere de un umbral mínimo para procesar adecuadamente el script. Sin embargo, a partir de ahí cada número de repeticiones que se adicione requiere menos reserva de memoria añadida a la anterior.

CPU

• En todas las series se rebasa el 50 % de uso en periodos de tiempo notables, no hallándose ninguna significatividad en el hecho de que unas requieran más que el resto.

• Las series con mayor número de repeticiones tienen el fragmento del código, donde la ejecución conlleva un alto porcentaje de uso de la CPU, más amplio. Teniendo en cuenta que el ensanchamiento se produce en un tiempo intermedio, probablemente corresponda a la ejecución del Bucle 6 que, como se mostró en la Tabla 5.1, varía fuertemente. Así, por ejemplo para la serie de 30 repeticiones (en naranja), esto casa bastante bien con los algo más de sesenta segundos que indica la tabla. Ello podría confirmarse contrastando con la gráfica homóloga para el código paralelizado por la GPU. De ser exitosa la citada identificación, se estaría ante una gran oportunidad para que la tarjeta gráfica descongestione la carga de la CPU. Lo que sí debería de verse seguro es dicho tramo acortado.

Para la CPU se puede concluir que el script, en todas sus versiones, requiere el uso de entre el 50 y el 60 % de la CPU. Además, no se evidencian diferencias significativas en el nivel de dicho uso, apreciándose todas estadísticamente igual. Donde sí se ha detectado una diferencia clara es en la duración en la que la CPU hace uso del promedio que rebasa el 50 %, notándose que cuanto más largo es el script, más larga es esa banda. Es altamente probable que pueda asociarse con la ejecución del bucle 6, puesto que es el que, de forma predominante, crece de una serie a otra. En ese sentido, el contraste con la figura homóloga en el caso paralelizado arrojará luz a esta cuestión.

GPU

En lo que respecta a la GPU, el comportamiento de todas las series es muy parecido y las diferencias se advierten poco claras, sobretodo teniendo en cuenta el desfase temporal que se ha comentado que existe en la captación de unas señales a otras. Lo más importante que se desprende del análisis en *excel* es que, éste, arroja que ninguna tiene un promedio de más del 15 % de uso. Cuando se analicen las gráficas homólogas en el código alfa, obtenido con el

paralelizado en la GPU, se debería de verificar que dicho promedio asciende claramente, con especial énfasis en aquellos bucles en los que existen líneas con mayor reducción de tiempo.

Nota: - Se podría haber realizado un análisis más exhaustivo, aportando datos numéricos sobre las series con las que se han elaborado las gráficas e incluso concretando qué promedio de la magnitud analizada se asocia a cada bucle individualmente[58] . Pero esta es una labor que se ha visto más interesante desarrollar en el apartado 5.5, donde se comparará detalladamente el paralelizado GPU con el código base sin paralelizar.

5.3. Paralelizado en la CPU

Como ya se pudo ver en el apartado 4.3 del capítulo anterior, el código paralelizado obtenido haciendo uso de los núcleos de la CPU ha distado mucho de ser candidato a código alfa. Durante su implementación, ya se pudo dejar constancia de toda la problemática que envolvía a la acción de sembrar con bucles *parfor* al grueso del script. Es momento ahora de reflejar los resultados que ha arrojado para cada una de las seis series – como ya se expuso en el código base – tanto en tiempo de cálculo, como en coste computacional.

5.3.1. Tiempo de cálculo

La Tabla 5.2 recoge los valores de tiempo, tanto por bucle como totales, que se ha obtenido mediante un procedimiento de toma de muestra análogo al llevado a cabo para analizar el código sin paralelizar.

SP 5 Repeticiones		P 5 Repeticiones		SP 10 Repeticiones		P 10 Repeticiones		SP 15 Repeticiones		P 15 Repeticiones	
Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora
1	3,208	6,066	-89,082	1	6,256	12,926	-106,617	1	9,314	20,978	-125,239
2	0,010	0,349	-3339,145	2	0,014	0,357	-2381,481	2	0,020	0,452	-2120,949
3	3,463	3,564	-2,909	3	6,682	7,060	-5,671	3	10,157	10,728	-5,626
4	0,491	0,683	-39,195	4	0,911	0,976	-7,143	4	1,372	1,561	-13,745
5	0,016	0,016	1,663	5	0,036	0,035	3,604	5	0,061	0,065	-6,692
6	10,447	27,054	-158,958	6	20,265	53,587	-164,431	6	30,408	80,493	-164,713
7	3,054	28,145	-821,586	7	3,723	178,331	-4690,568	7	4,259	522,124	-12157,972
8	14,851	15,436	-3,937	8	15,201	19,242	-26,580	8	15,057	22,247	-47,756
Total	36,453	81,313	-44,860	Total	54,449	275,121	-220,672	Total	72,459	663,340	-590,881
Desv.St		2,540		Desv. St		34,818		Desv. St		42,250	
Mejora	-123,065			Mejora	-405,281			Mejora	-815,469		

Tabla 5.2. Resultados de los tiempos de ejecución para el código alfa CPU en función de RepNum. En rojo se muestran resaltados los tres bucles que no han sido objeto de paralelización.

[58] *Pues se han generado, durante la toma de muestra en MatLab, vectores que contienen las horas de inicio/final de cada bucle (horas, minutos y segundos), lo que permite identificarlos gracias a que afterburner toma medidas en continuo y sincronizado con el reloj interno de la CPU.*

Los resultados hablan por si solos, pero merece la pena incidir en varios aspectos que se van a ir ordenando por dejar claras las ideas:

• La primera serie (5 repeticiones) es unas dos veces más lenta que su homóloga del código base. Por su parte, la segunda serie se ejecuta unas cinco veces más lentamente y, la tercera, nueve. Esto deja patente, ya desde un primer momento, que el enlentecimiento que promueve una paralelización del código base, mediante el uso de los ocho núcleos del ordenador, es de carácter exponencial.

• Los bucles 3 y 4 (no paralelizados) sufren también empeoramientos. Especialmente el cuarto. En cambio, el Bucle 5 no se altera significativamente. El octavo se daña considerablemente.

• De los bucles paralelizados (el resto), el seis y el uno los sufren pero daños moderados mientras que, el dos sufre enlentecimientos severos.

Mención aparte a los anteriores análisis merece el bucle 7, que es el gran responsable de que la instrucción *parfor* perjudique tanto a nuestro algoritmo de procesado de imágenes y, cuyos aumentos en tiempos de ejecución son tan exponenciales, que ni siquiera ha sido posible adquirir datos para las series de 20, 25 y 30 repeticiones, puesto que *MatLab* proporcionaba los errores que recoge la Figura 5.3.

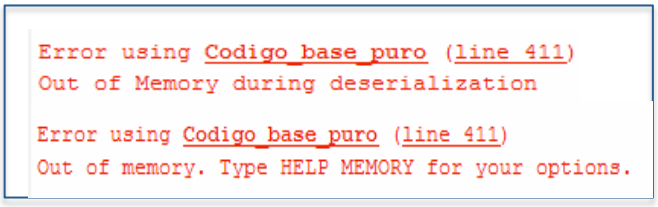


Figura. 5.3. Anuncios que emite MatLab en consola advirtiendo de la aparición de un error.

Ambos errores son desconocidos para profanos en materia de programación CUDA en el entorno de MatLab, pero parecen indicar que el procesador tiene un conflicto con la memoria, que le impide gestionar adecuadamente cómo deben de repartirse los cálculos sus ocho núcleos. Algo que, a la vista de la Tabla 5.3 y, dada la exponencialidad con la que aumentan los tiempos de ejecución del séptimo bucle, es completamente entendible.

La conclusión que se puede formular ante todas estas lecturas es clara y rotunda, sin que sean necesarios otros tipos de análisis, estudios o comparaciones: **El paralelizado por vía CPU es inviable, tanto en términos lógicos como en prácticos**. En el ámbito de la experimentación mediante Técnicas Ópticas, es mucho más habitual emplear un número de repeticiones próximo a treinta que a cinco, siendo que, con éstas últimas ya se doblan los tiempos de cálculo por procesado de un caso. Por tanto, esta vía está completamente desaconsejada para el algoritmo con el que se procesan las imágenes captadas por la técnica LEI.

5.3.2. Coste computacional

Todo y que ha quedado claro que la paralelización por CPU no es una opción para el algoritmo con el que se está elaborando el presente proyecto, en el apartado 4.3 (Figura 4.3) se hizo una descripción cualitativa acerca de cómo se comportaban los núcleos en el código base, con el fin de indagar hasta qué punto un paralelizado mediante el uso del comando *parfor* podría ofrecer un beneficio neto.

Ya se vio, de hecho, que no era muy alentador debido a la buena gestión que, de forma natural, ya estaba ejerciendo el propio procesador para ejecutar el código base. Se vio que no era necesario sobrepasar un 60 % del uso en ninguno de los núcleos, pero que todos trabajaban, lo que hacía sospechar que raramente iba a repartirse mejor la ejecución paralelizando por esta vía. Sin embargo, resultaría al menos curioso, inspeccionar qué ha sucedido definitivamente con la gestión de los recursos en el código "alfa" CPU. Para apreciarlo se ha hecho uso, nuevamente, del monitor de recursos *afterburner*, pero empleando únicamente la serie de 5 repeticiones pues, de poder llegar a ser interesante (caso de que el enlentecimiento se pudiera compensar con una hipotética mayor cantidad de recursos) con la serie siguiente, esto es, la de 10 repeticiones, sería ya harto complicado hallar punto de equilibrio alguno. La Figura 5.4 recoge la señal que se ha captado con *afterburner*.

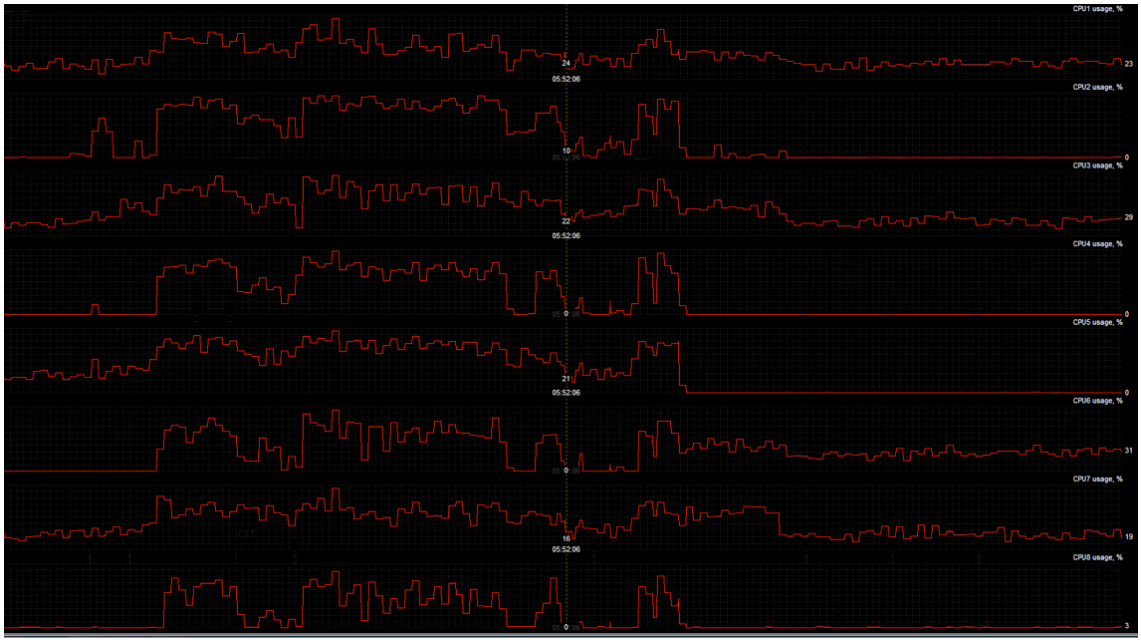


Figura. 5.4. Monitor de recursos afterburner mostrando la carga computacional.

Sin entrar demasiado en detalle (aunque este tipo de imágenes son para llevar a cabo análisis cualitativos saca también información numérica en pantalla), el cambio que se aprecia con el que se obtuvo en la Figura 4.3 es ostensible. En aquella ocasión, siete de los ocho núcleos trabajaban ininterrumpidamente a poco más del 50 %. En cambio, ahora todos sin excepción, lo hacen en muchos tramos del código a casi el 100 %, por lo que incluso están más exigidos de lo que lo estaban ejecutando el código base. Esto permite concluir que, artificialmente, el comando *parfor* los hace trabajar más de lo necesario, por lo que esa hipotética situación de contar al menos con más recursos, se desvanece por completo.

5.4. Paralelizado en la GPU

A diferencia de lo que se acaba de ver para la paralelización con ayuda de la CPU (los ocho núcleos), la conseguida a expensas de la tarjeta gráfica *NVidia GeForce GTX 960* ha sido bastante exitosa.

5.4.1. Tiempo de cálculo

En la Tabla 5.3 se recogen los resultados obtenidos para los distintos tiempos de cálculo, tanto por bucle como totales, junto con los que ya se incluyeron en la Tabla 5.2 para el código base, de forma que se pueden apreciar las diferencias a golpe de vista[59] . Por su parte, la Figura 5.5 muestra los tiempos totales, comparando el código alfa con el código base.

SP 5 Repeticiones		P 5 Repeticiones		SP 10 Repeticiones		P 10 Repeticiones		SP 15 Repeticiones		P 15 Repeticiones	
Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora
1	3,208	2,879	10,276	1	6,256	5,720	8,576	1	9,314	8,543	8,273
2	0,010	0,009	7,895	2	0,014	0,012	15,278	2	0,020	0,014	29,951
3	3,463	3,310	4,427	3	6,682	6,627	0,823	3	10,157	9,905	2,483
4	0,491	0,461	5,946	4	0,911	0,917	-0,662	4	1,372	1,401	-2,092
5	0,016	0,016	1,247	5	0,0360	0,034	5,453	5	0,061	0,060	0,768
6	10,447	6,149	41,139	6	20,265	11,896	41,300	6	30,408	18,084	40,529
7	3,054	2,135	30,081	7	3,723	2,805	24,646	7	4,259	3,540	16,893
8	14,851	12,980	12,599	8	15,201	13,137	13,58	8	15,057	13,349	11,340
Total	36,453	28,657	7,796	Total	54,449	42,202	12,247	Total	72,459	56,304	16,155
Desv.St	0,318	0,125		Desv. St	1,217	0,288		Desv. St	1,049	0,320	
Mejora	21,386			Mejora	22,493			Mejora	22,296		

SP 20 Repeticiones		P 20 Repeticiones		SP 25 Repeticiones		P 25 Repeticiones		SP 30 Repeticiones		P 30 Repeticiones	
Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora	Bucle	Tiempo (s)	Tiempo (s)	Mejora
1	12,538	11,345	9,513	1	16,043	14,488	9,691	1	18,924	17,225	8,981
2	0,026	0,015	42,313	2	0,030	0,017	45,165	2	0,035	0,020	42,375
3	13,812	13,284	3,823	3	17,318	16,961	2,062	3	20,829	20,274	2,661
4	1,857	1,836	1,14	4	2,445	2,495	-2,06	4	3,015	3,068	-1,778
5	0,096	0,097	-1,849	5	0,143	0,146	-1,816	5	0,167	0,179	-6,988
6	41,222	24,218	41,249	6	51,519	30,955	39,916	6	61,705	38,715	37,257
7	5,169	4,258	17,626	7	6,219	5,340	14,124	7	6,976	6,410	8,116
8	15,113	13,784	8,790	8	15,570	17,191	-10,407	8	15,415	18,515	-20,106
Total	92,135	70,602	21,532	Total	112,09	89,839	22,251	Total	130,324	104,406	25,917
Desv. St.	1,127	1,396		Desv. St	2,533	1,349		Desv. St	2,831	1,692	
Mejora	23,37			Mejora	19,851			Mejora	17,832		

Tabla 5.3. Resultados de los tiempos de ejecución para el código alfa GPU en función de RepNum. En rojo se muestran resaltados los tres bucles que no han sido objeto de paralelización.

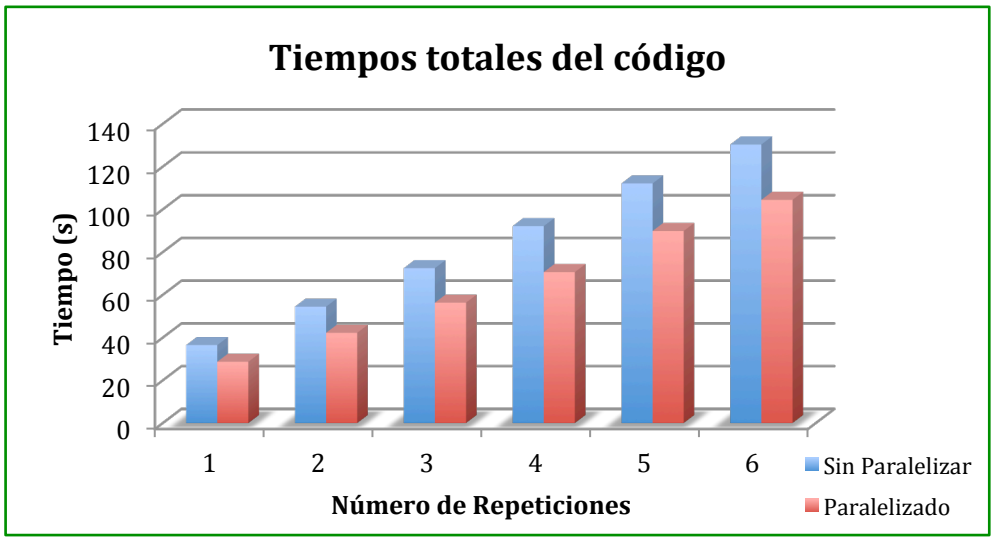


Figura. 5.5. Gráfica que muestra los tiempos de cálculo totales para cada serie.

Se quiere incidir una vez más en que, este primer bloque de apartados que preceden al ya mencionado varias veces, *apartado 5.5. Comparación CPU/GPU*, únicamente pretende reflejar los resultados que se han obtenido durante los ensayos con cada código (el base, el paralelizado CPU y el paralelizado GPU). A tal efecto, las tablas que se han ido introduciendo serán las que darán soporte a los estudios más concretos, que serán desarrollados para arrojar la mayor cantidad de información posible que permita obtener una serie de conclusiones férreas en este trabajo. Sin perjuicio de ello y, anticipando lo que se discutirá en profundidad posteriormente, del rápido análisis de la Tabla 5.3 y de la Figura 5.3, se aprecia que:

• Los tiempos de cálculo totales del código base se han conseguido mejorar entre un 20 y un 23 %, en el caso de la serie con 20 Repeticiones[60]. Demuestra que el paralelizado es factible.

• De los bucles que no han sido paralelizados, el Bucle 2 reporta mejoras "importantes" con el aumento de las repeticiones. Por su parte, el Bucle 4 exhibe un comportamiento más errático, pero en términos generales se identifica un ligero empeoramiento (nunca más del 2,1 %). El Bucle 5 mejora ligeramente en las tres primeras series para, a continuación, empeorar. Estos resultados indican que el paralelizado pudiera estar afectando también a algunas de las líneas que no se han modificado. No obstante, se tratan de bucles cuyos tiempos de ejecución son tan pequeños, que pueden verse afectados por la dispersión que se produce entre una toma de muestra y la siguiente (recordar que se han tomado 40 tiempos de cada bucle y 40 totales).

---

[59] *Hay que aclarar que, los resultados recogidos en la Tabla 5.3, fueron adquiridos en el marco del estudio paramétrico de la sensibilidad del paralelizado al número de repeticiones, que se expondrá en el apartado 5.5. Por tanto, son objetivos del presente apartado el reportar qué resultados ha arrojado el paralelizado y justificar convenientemente a qué líneas de código se atribuyen (para demostrar que, efectivamente, se han conseguido reducciones de tiempo robustas y no fruto de la aleatoriedad). Será en el citado apartado cuando se lleven a cabo análisis y valoraciones más detalladas.*

[60] *Como se discutirá en el apartado 5.5, constituye el número de repeticiones idóneo con el que trabajar para procesar imágenes en el algoritmo desde la perspectiva de un paralelizado óptimo.*

• De los bucles que sí han sido paralelizados (el resto), el Bucle 6 se destapa como el que mayor mejoría aporta (más del 40 % en las 4 primeras series). En las dos últimas sufre un desplome ligero que le lleva al 37 %. Pero, en términos generales, es el bucle que reporta mejoras más estables. El siguiente en importancia corresponde al Bucle 7. Se aprecia como va reduciendo su notable mejoría (que empieza en el 30 % en la primera serie) con el aumento del número de repeticiones. A continuación viene el Bucle 8 que es, como se aprecia en la Tabla 5.3, el que muestra un comportamiento más extraño. Como se vio en la Tabla 5.1, se trata de un bucle al que no le afecta el aumento de repeticiones cuando se ejecuta en el seno del código base. En este sentido, su ejecución en el código alfa GPU demuestra que se obtiene una buena y estable mejoría para las cuatro primeras series. Sin embargo, su tiempo de ejecución se dispara alarmantemente en las dos últimas provocando que, en lugar de mejorar con el paralelizado, empeore drásticamente. Finalmente, los Bucles 1 y 3, cuyas mejoras están bastante consolidadas, oscilan alrededor del 9 % y 3 % respectivamente. Pese a que todo esto se puede analizar a partir de la simple inspección visual de los datos incluidos en la Tabla 5.3, la Figura 5.6 muestra un gráfico 3D equivalente adjuntando los tiempos parciales.

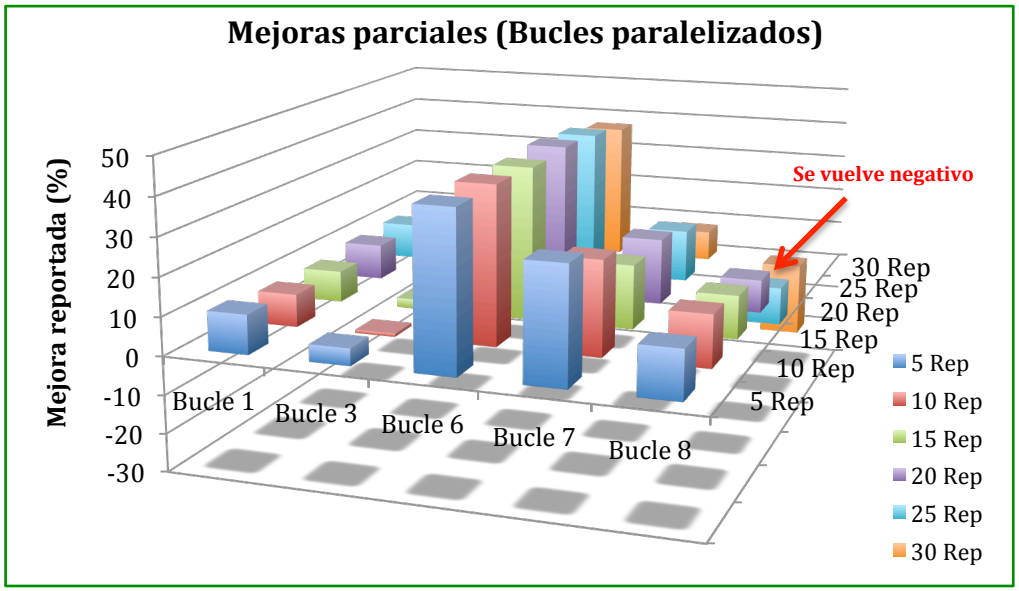


Figura. 5.6. Gráfica que muestra los tiempos de cálculo parciales para los bucles paralelizados.

• Las mejoras consolidadas por las cuatro primeras series (superiores al 20 %) se ven drásticamente reducidas en las dos últimas. Esto es esencialmente atribuido al aumento del tiempo de ejecución – de más de 3 segundos – que sufre el Bucle 8.

Analizados superficialmente todos estos datos, la conclusión preliminar a la que se llega es la siguiente: - el paralelizado con GPU mejora ostensiblemente el algoritmo que, de acuerdo a lo que se vio en el apartado 5.2, es complejo, largo y pesado. La citada mejora viene principalmente de la mano de la que se consigue con el bucle más representativo del código, que es el número seis, bucle en el que fueron concentrados los mayores esfuerzos por aplicar la paralelización. No obstante, es conveniente realizar un estudio más profundo asociado al deterioro del comportamiento del Bucle 8, pues merma de forma relevante en las dos últimas series las mejoras anteriormente consolidadas.

Dicho estudio se abordará en profundidad ya en el apartado 5.5. Lo que sí se va a presentar a continuación es una prueba fehaciente de que, efectivamente, las mejoras totales y parciales que se observan, vienen acompañadas justificadamente de la reducción de tiempo de cálculo en las líneas que, en el apartado 4.3 del Capítulo 4, se mostraron paralelizadas dentro de cada bucle. Para ello, se irán mostrando extractos, obtenidos de *MatLab* en su modo de ejecución "Run and Time", para cada uno de los bucles objeto de paralelización. El lector mismo puede contrastar, en cada caso, con lo que el *Anexo B* incluía para el código base.

Nota: - Por no extender este resultado varias páginas, se empleará únicamente una de las series. En este caso, la de 5 repeticiones pues, como ya se explicó en partes anteriores del documento, es la serie con la que se ha trabajado para desarrollar y obtener el código alfa.

Bucle 1

```
                152              % BUCLE 1
< 0.01        1  153              tic;
< 0.01        1  154              for n=1:5
  2.87        5  155                  data = GetMovie_paralelizada(0,n,'Photron','mraw',NumPerRep,Framerate);
                156 %                  data = gpuArray(data);
< 0.01        4  157                  m=1;
< 0.01        4  158                  g=1;
< 0.01        4  159                  for c=1:NumBI
  0.01       16  160                      if max(max(data(yinjector:yinjector+200,:,c)))>LEDcontrol
< 0.01        8  161                          Iback(n,m)=data(:,:,c);
< 0.01        8  162                          m=m+1;
< 0.01        8  163                      else
< 0.01        8  164                          Inoise(n,g)=data(:,:,c);
              8  165                          g=g+1;
< 0.01       16  166                      end
< 0.01       16  167                  end
< 0.01        4  168                  if size(Iback,2)~=size(Inoise,2)
                169                      if size(Iback,2)<size(Inoise,2)
                170                          for d=NumBI+1:NumPerRep
                171                              if max(max(data(yinjector:yinjector+200,:,d)))>LEDcontrol
                172                                  Iback(n,m)=data(:,:,d);
                173                                  break
                174                              end
                175                          end
                176                      else
                177                           for d=NumBI+1:NumPerRep
                178                              if max(max(data(yinjector:yinjector+200,:,d)))<LEDcontrol
                179                                  Inoise(n,size(Inoise(n,:),2)+1)=data(:,:,d);
                180                                  break
                181                              end
                182                          end
                183                      end
                184                  end
                185                  %% Combustion Detector
< 0.01        4  186                  Inoisesum_rep=gpuArray(zeros(ysize,xsize));
                187 %                  Inoisesum_rep=gpuArray(Inoisesum_rep);
< 0.01        4  188                  for kk=1:length(Inoise(n,:))
< 0.01        8  189                      Inoisesum_rep=Inoisesum_rep+double(Inoise(n,kk));
```

La reducción del tiempo de cálculo viene de la mano de la línea 155, gracias a la mejora de la función *GetMovie_paralelizada*, que es una de las tres líneas que se paralelizaron. El resto no llega a los 0,01 segundos y cuestan lo mismo que en el bucle para el código base.

Bucle 3

La mejoría proviene esencialmente de la línea 271, gracias también a la función *GetMovie_paralelizada*, pero el comando *gather* – necesario para que la memoria video de la GPU no colapse en este bucle – desluce dicha mejora. El resto de líneas no cambian significativamente, como puede verse.

```
                267          % BUCLE 3
< 0.01        1 268           tic;
< 0.01        1 269           for r=1:length(Rep2Process)
< 0.01        5 270               n=Rep2Process(r);
  2.90        5 271               data = GetMovie_paralelizada(0,n,'Photron','mraw',NumPerRep,Framerate);
  0.21        5 272               data=gather(data);
                273
< 0.01        5 274               for Nimg=1:NumPerRep
  0.12     1250 275                   if max(max(data(yinjector:yinjector+100,:,Nimg)))>LEDcontrol; %check if there is LED in the "f" Image
< 0.01      625 276                       LEDcount(n,Nimg)=1;
                277                       %                       BLKcount(n,Nimg)=0;
< 0.01      625 278                       TimeLED(n,Nimg)=1000000/Framerate*Nimg;
                279                       %             if Nimg>NumBI
                280 %                        TimeLED=gpuArray(TimeLED);
  0.16      625 281                       Itotal(n,Nimg)=data(:,:,Nimg);
                282                       %             end
< 0.01      625 283                   else
                284                       %                       LEDcount(n,Nimg)=0;
< 0.01      625 285                       BLKcount(n,Nimg)=1;
< 0.01      625 286                       TimeBLK(n,Nimg)=1000000/Framerate*Nimg;
                287                       %             if Nimg>NumBI
                288 %                       TimeBLK=gpuArray(TimeBLK);
  0.16      625 289                       Iflame(n,Nimg)=data(:,:,Nimg);
                290                       %             end
< 0.01     1250 291                   end
< 0.01     1250 292               end
< 0.01        5 293           end
```

Bucle 6

```
                409          % BUCLE 6
< 0.01        1 410           tic;
                411           % // Este for estoy consiguiendo mejorarlo un 40 %
< 0.01        1 412           for r=1:length(Rep2Process)
< 0.01        4 413               i=Rep2Process(r);
< 0.01        4 414               for j=NumBI+1:NumPerRep
                415                   %             Flamepos=find(TimeBLK(i,:)==TimeBLKord(i,j));
                416                   %             mask2=zeros(ysize,xsize);
                417                   %             mask2=Iflame1(i,Flamepos)>60;
                418
                419                   %             diff=(double(Itotalint(i,j))*mask)-(double(Iflame1(i,Flamepos)*mask2));
  1.20      908 420                   a=gpuArray(double(ItotalFINAL(i,j)));
                421 %                   a=gpuArray(a);
  0.93      908 422                   b=gpuArray(double(IflameFINAL(i,j)));
                423 %                   b=gpuArray(b);
                424 %                               c=double(b);
  0.50      907 425                   diff=double(a-b);
  2.04      907 426                   KL_rep(j)=log(complex((IbackavgFin./diff)));
  0.13      907 427                   KL_Axis_rep(j)=KL_rep(j)(1:70*pixmm,xinjector);
  1.11      907 428                   KLsat_rep(j)=log((IbackavgFin./diffSat).*mask); %saturated KL value
  0.10      907 429                   KLsat_Axis_rep(j)=KLsat_rep(j)(1:70*pixmm,xinjector);
                430
                431
< 0.01      907 432               end
                433 %               save(strcat('KL_LEI_Rep_',num2str(i),'.mat'),'KL_rep', 'KLsat_rep', 'KL_Axis_rep', 'KLsat_Axis_rep', '-v7.3');
  0.14        3 434               clearvars KL_rep KLsat_rep KL_Axis_rep KLsat_Axis_rep;
< 0.01        3 435           end
```

En este caso puede apreciarse como la mejora viene repartida entre muchas más líneas que en los casos anteriores, pese a que solo se paralelizaron tres líneas. Se trata de muchas operaciones matriciales, que es donde los hilos de la GPU más pueden incidir. No obstante, las mejoras más acusadas han venido de la mano de las líneas 426 y 428, respectivamente[61] .

[61] *Esto demuestra que, la mejoría de la paralelización, va más allá de las simples líneas que se modifican. Guardadas determinadas variables clave en la memoria de video de la GPU, muchas operaciones se ejecutarán en los hilos, lo que hace que algunas modificaciones trasciendan positivamente al resto de las líneas del código (esto, que aquí es beneficioso, origina algunas pérdidas de tiempo en otros casos).*

Bucle 7

Las líneas que más se reducen son la 474 y la 475. Llama poderosamente la atención ver que, la línea 472, que sí se paraleliza, dobla su tiempo de ejecución. Pero es gracias a ella que, las citadas con anterioridad, se benefician de la rapidez con la que los hilos de la GPU realizan las operaciones matriciales.

```
                451             % BUCLE 7
< 0.01        1 452             tic;
                453             % // Consigo mejorar el for entre un 18 y 27 % aprox.
< 0.01        1 454             for j= NumBI+1:NumPerRep;
< 0.01      199 455                     sum1 = zeros(ysize,xsize);
< 0.01      199 456                 sum2 = zeros(ysize,xsize);
< 0.01      199 457                 for r = 1:length(Rep2Process);
< 0.01      995 458                     i=Rep2Process(r);
  0.37      995 459                     sum1 = sum1+double(ItotalFINAL(i,j)(:,:));
  0.38      995 460                     sum2 = sum2+double(IflameFINAL(i,j)(:,:));
< 0.01      994 461                 end
< 0.01      198 462                 Itotalsum(j) = sum1;
  0.16      198 463                 Itotalavg(j) = mask.*double(Itotalsum(j))./length(Rep2Process);%average total illumination with LED and flame
< 0.01      198 464                 Iflamesum(j) = sum2;
  0.11      198 465                 Iflameavg(j) = double(Iflamesum(j))./length(Rep2Process);%average flame illumination
  0.02      198 466                 mask2 = zeros(ysize,xsize);
  0.03      198 467                 mask2 = Iflameavg(j)>60;
  0.11      198 468                 Iflameavg(j) = Iflameavg(j).*mask2;
  0.01      198 469                 mask=gather(mask);
                470
                471                 %calculate the KL of soot
  0.22      198 472                 diff = gpuArray(double(Itotalavg(j))-double(Iflameavg(j)));
                473 %                       diff=gpuArray(diff);
  0.46      198 474                 KL_avg(j)=log(complex(IbackavgFin)./diff);
  0.23      198 475                 KLsat_avg(j)=log((IbackavgFin./diffSat).*mask);%saturated KL value
< 0.01      198 476                 Time(j)= 1000000/Framerate*j;%unit [us]
< 0.01      198 477             end
```

Bucle 8

```
                571             % BUCLE 8
< 0.01        1 572                 tic;
< 0.01        1 573                 for j= NumBI+1:NumPerRep
                574
                575                     %PLOTS
  0.17      233 576                     h1=figure(1);
< 0.01      233 577                     KLAVG=KL_avg(j);
  0.16      233 578                     KLAVG=single(KLAVG);
  0.21      233 579                     KLrot(j)=imrotate(KLAVG,90);
                580 %                     KLrot(j)=imrotate(KL_avg(j),90);
 12.32      233 581                     imshow( KLrot(j)(((xsize-xinjector)-round(12*pixmm)):((xsize-xinjector)+round(12*pixmm)),yinjector:round(70*pixmm)
  0.42      232 582                     KLrot(j)=gather(KLrot(j));
                583 %                     tic;
                584 %                     axis ([0 (70*pixmm-yinjector)  0 24*pixmm]);
                585 %                     axis on;
                586 %                     set(gca,'Xtick',[0:pixmm*10:(70*pixmm-yinjector)],'Ytick', [0:pixmm*4: 24*pixmm],...
                587 %                         'YTickLabel',('12' '8' '4' '0' '4' '8' '12'),'XTickLabel',('0' '10' '20' '30' '40' '50' '60' '70' '80'),...
                588 %                         'fontsize',22,'FontWeight','bold');
                589 %                     xlabel('Spray axis[mm]','Fontsize',22,'FontWeight','bold');
                590 %                     ylabel(' [mm]','Fontsize',22,'FontWeight','bold');
                591 %                     colormap(jet);
                592 %                     colorbar('location','EastOutside','Fontsize',14);
                593 %                     text(15,30,strcat(num2str(round(Time(j))),'us'),'color','k','BackgroundColor','w','Fontsize',22,'Fontweight','b
                594 %                     set(h1,'PaperPositionMode','auto');
                595 %                     name=strcat(results,'\',conditions,'_AVG_',num2str(round(Time(j))),'us.png');
                596 %                     print(h1,'-dpng',name);
                597 %                     close;
                598 %                     toc;
                599 %                     fclose all
```

En este último bucle puede apreciarse como la paralelización ha incidido especialmente sobre la función *imshow* y sobre las líneas que conducen a la construcción de la variable *KLrot{j}*, a través del comando *imrotate*. En el *Anexo B* puede verse que, las líneas 577, 578 y 579, antes constituían una sola (la 532 en el anexo). Sin embargo, la suma de estas tres consumen casi la quinta parte. Apuntar también que, de nuevo la introducción del comando *gather*, contrarresta lo que podría haber sido una mejora más notable todavía, aportando 0,42

segundos que antes no existían. Pero, tal como se comentó en el apartado 4.3 del anterior capítulo, su inclusión no es fruto de un capricho, sino que en su ausencia la memoria video de la GPU colapsaba, sin terminar de ejecutar las vueltas que contiene el bucle (lanzaba error).

5.4.2. Coste computacional

En este sub-apartado que ahora empieza, se va a proceder a discutir qué es lo que ocurre con los recursos de los elementos que ya se estudiaron en el sub-apartado 5.2.1 de este mismo capítulo. A saber: la *CPU*, la *memoria física*, la *GPU* y la *RAM*. A estos cuatro se le adiciona ahora la *memoria de video*, que también resultará interesante averiguar cómo se comporta. Las Figuras 5.7 y 5.8 recogen los resultados obtenidos.

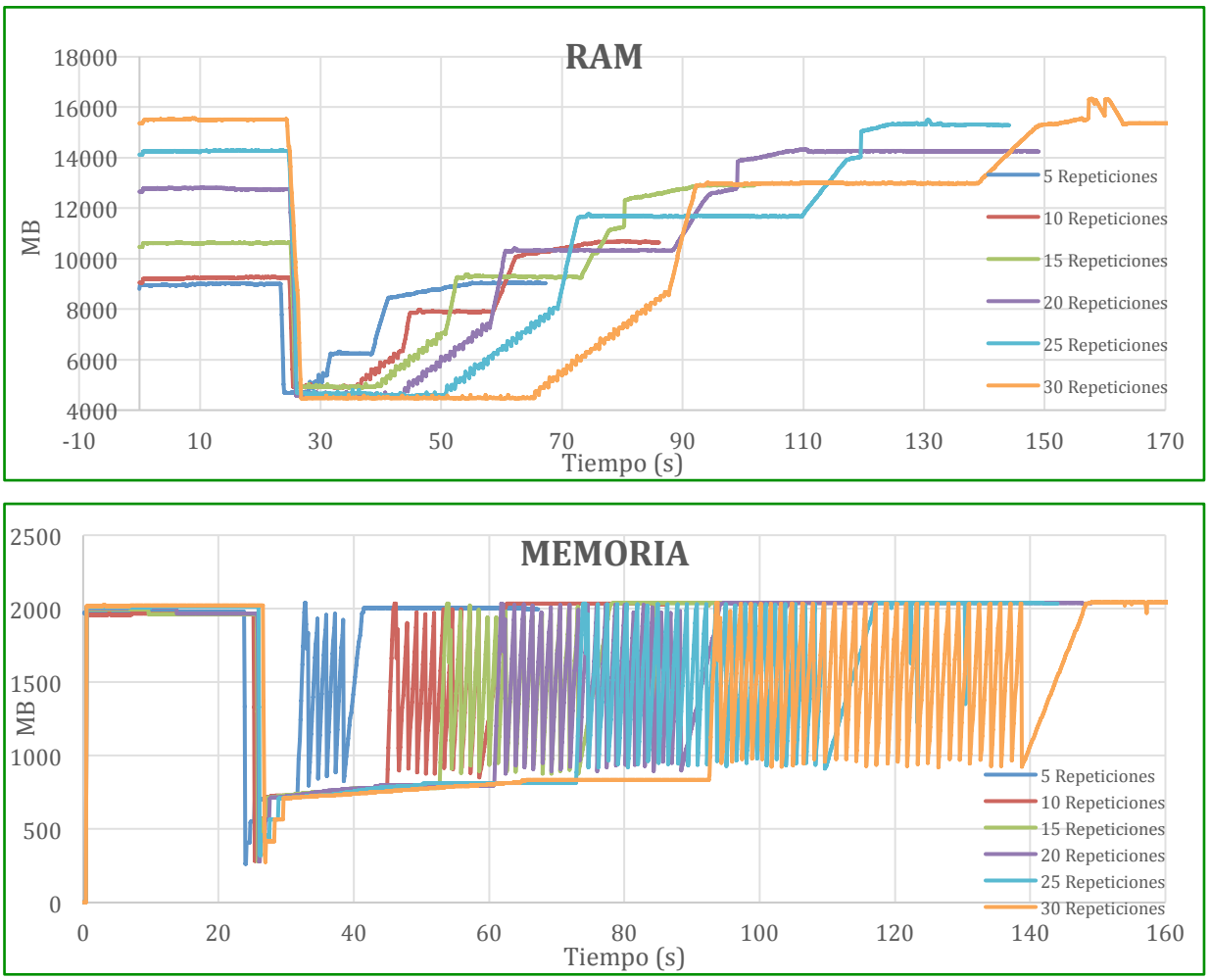


Figura. 5.7. Gráficas frente al tiempo para: a) RAM, b) Memoria física.

Nota: -La gráfica de la memoria video no se ha incluido porque el registro fue nulo de forma constante para todas las series. Debe de haber algún problema de intercambio de datos entre el monitor de recursos afterburner y el procesador, ya que no debería de ser nula.

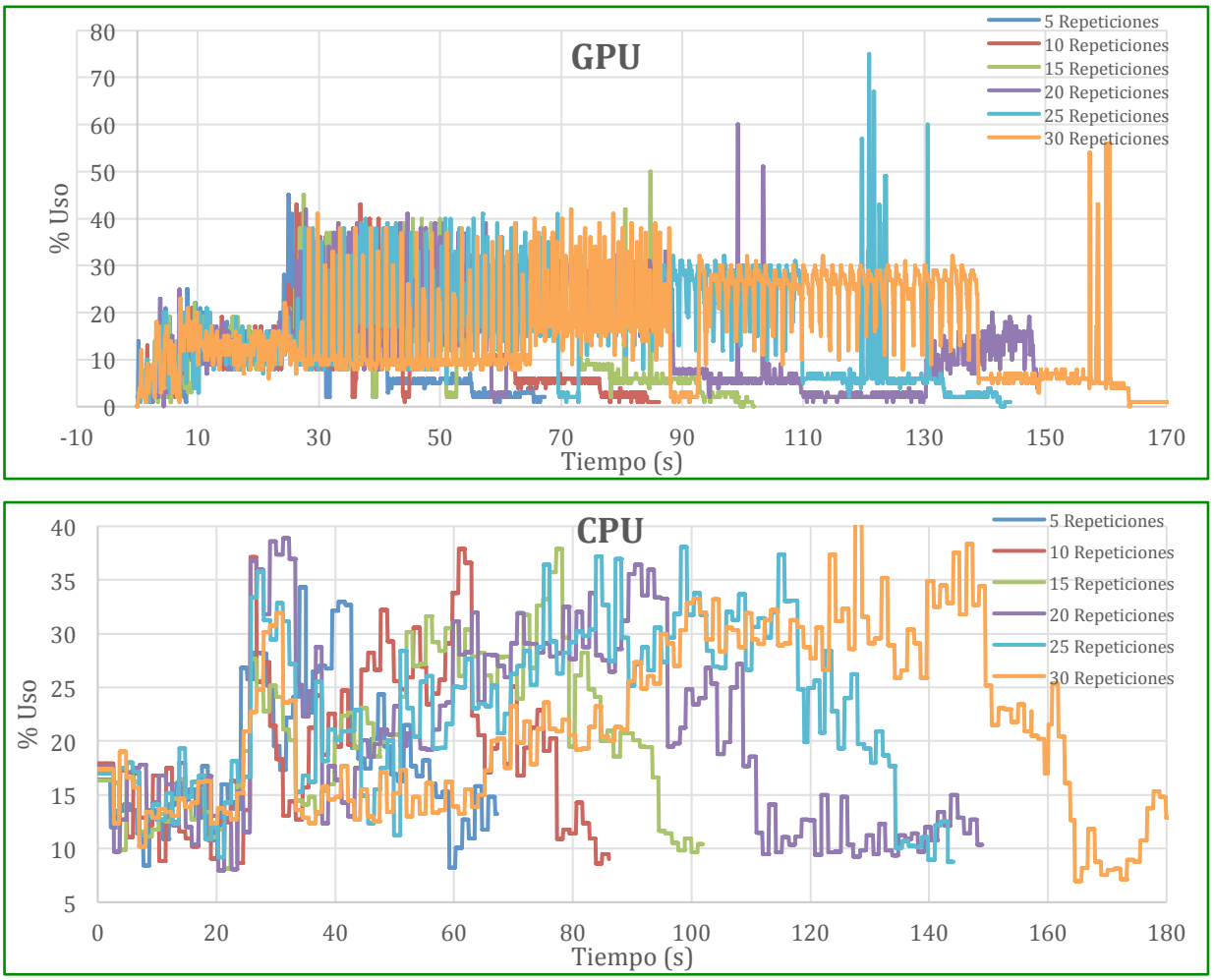


Figura. 5.8. Gráficas frente al tiempo para: a) GPU, b) CPU.

Ordenados los datos ya en forma de gráficas, es momento de iniciar los análisis, del mismo modo que se hizo en el apartado 5.2 con el código base.

RAM

• Las primeras zonas son exactamente como las obtenidas en la Figura 5.1 para el código base.

• El inicio, esta vez, se sincronizó en 24 segundos para todas. De esta forma, ahora sí que cada serie termina más tarde que la que le precede.

• Las caídas de RAM ahora son más pequeñas. Cabe tener en cuenta que se han unificado para que el final de la caída de todas las series (el "suelo") sea común. Pero, si se toma como referencia, por ejemplo, la serie de 5 repeticiones (azul oscuro) ésta cae desde los casi 9000 MB a los unos 4500 MB. En la Figura 5.1 se vio que esta caída era desde unos 13000 MB hasta unos 7000. Se puede comprobar esta tendencia en todas las series examinando las figuras, pero es más acusado en las primeras series. Que sea menor es razonable, pues al iniciar el código, el primer bucle ya está paralelizado. En este sentido, se cumplen las previsiones.

• Las zonas de las rampas que cambian de pendiente son exactamente igual en comportamiento que lo que se describió para el código base. De nuevo, las longitudes indican que el bucle/s asociado/s son ejecutados en algoritmos con un mayor o menor número de repeticiones (y de ahí la discrepancia en extensión, pues son cada vez códigos más pesados y largos).

• El tramo que antes era "dentado" ahora ha pasado a ser una línea recta. Es decir, la RAM ahí descansa y no necesita aumentar. Esa etapa correspondía al Bucle 6, que es el que va ganando en extensión con mayor notoriedad al aumentar el número de repeticiones, tal y como se vio con anterioridad. Aquí se puede hallar la explicación de uno de los mayores éxitos del paralelizado, pues la GPU ha conseguido relajar a la memoria RAM del procesador que, si recordamos, en este tramo sufría una serie de ciclos de carga/descarga. Por confirmar este hecho (aunque de la GPU se hablará a continuación), si se toma como referencia, por ejemplo, la serie de 30 repeticiones (la naranja) se observará que, el tramo en el que tiene lugar esta "relajación" de la RAM coincide con una fuerte actividad de la GPU, que llega incluso al 30 % de uso, cuando en el código base su promedio no sobrepasaba el 15 % de uso. De hecho, el promedio en este tramo es del 25,11 % de uso (aunque no se muestre, se ha obtenido de *Excel*). Además, este tramo asociado al Bucle 6 dura menos, con lo cual la RAM inicia la rampa que se asocia al Bucle 8 mucho antes (a partir de ese instante, es aprovechable una linealmente creciente cantidad de RAM que antes no estaba disponible).

• El tramo final, que corresponde al Bucle 8, se aprecia algo más corto pero con una pendiente menor en las series de 25 y 30 repeticiones. En las de más bajas repeticiones, en cambio, se aprecia que las pendientes son más cortas lo que, unido a que no se aprecian cambios de pendiente significativos comparando con sus homólogas de la Figura 5.1, parece evidenciar que el bucle se reduce en tiempo, tal y como en la Tabla 5.3 podía verse reflejado.

Con todo, la valoración que se puede hacer de la RAM es que, claramente, en los bucles paralelizados sufre un relajamiento, de forma que la diferencia de ésta que se gana (tanto en cantidad como por el tiempo en el que actúa gracias al acortamiento de los bucles) es aprovechable por otros procesos que pueda tener en marcha el ordenador en ese momento.

Memoria física

• En el tiempo que duran las ejecuciones de los scripts desde que se da comienzo (a unos 24 segundos) se aprecia con claridad que la señal realiza unos ciclos de carga y descarga fuertes que antes no realizaba. Además, si tomamos como referencia la serie de 30 repeticiones por ejemplo, en la ventana de 90 – 135 segundos (Bucle 6) es donde se aprecian estas fuertes oscilaciones. Parece indicar que el nivel de reserva en memoria en el procesador se afecta mucho más por las constantes migraciones de líneas de la CPU a la GPU (cabe recordar que se trataba de un bucle con bastante paralelización, no solo por las líneas que se modificaron, sino por el hecho de que también se llevaban a cabo gran cantidad de posteriores operaciones con objetos de tipo *gpuArray*).

• El resto de los bucles son ejecutados a memoria esencialmente constante (poca variación).

• Los niveles de memoria disponibles, para todas las series, tras el primer pico en el que decaen las señales tras el inicio de los algoritmos, son considerablemente mayores que los que se tenían para el código base. Esto parece denotar que la paralelización estaría descargando buena parte de la memoria física del ordenador que, de nuevo, dispondrá de mayor cantidad para poder utilizar en otros procesos (las señales en el código base se movían en un orden de magnitud de 300 MB cuando ahora lo hacen por los 800 MB). Aparte de esto, en los picos de acción-recuperación que experimenta la memoria en el transcurso de la ejecución del Bucle 6, van produciéndose alternancias entre la recuperación total de la memoria de que dispone el ordenador (carga) y la caída que ocasionan algunas líneas que requieren de su uso (descarga).

Las conclusiones que se extraen son parecidas a las de la RAM. En esta ocasión, los resultados arrojan que la mayor parte del algoritmo consume algo más de la mitad de la memoria física del ordenador (unos 1200 Mb, por eso quedan 800 libres para usar en otras tareas) pero, en el Bucle 6, se producen unos ciclos de carga y descarga que, eventualmente, recuperan toda la memoria disponible por breves y cíclicos lapsos de tiempo (que era algo que sin paralelizado no ocurría).

GPU

• Los niveles promedio de uso de la GPU en todo lo que dura la ejecución del algoritmo ahora ascienden a 19,16 %, más los picos asociados al Bucle 8, que alcanza incluso el 70 % de forma puntual en la serie de 25 repeticiones.

• De nuevo, las series se comportan todas de forma completamente análoga, variando como siempre en las longitudes de los tramos.

• El tramo asociado al Bucle 6 no es el que mayor porcentaje punta alcanza, pero se sitúa a un promedio elevado del 25,1 %, como ya se indicó antes. Eso concuerda con lo que se explicitó al comentar el comportamiento de la RAM.

• Mientras que en el código base, los niveles de uso de la GPU (sus oscilaciones continuas) se situaban en torno al 10 -15 % de forma permanente, en el algoritmo paralelizado se aprecia como, tras el Bucle 6, el 7 y el 8 se ejecutan con muy poquito uso. Concretamente a una media cercana al 5 % bastante estable, pero el código termina con una serie de picos discretos (de dos a cuatro frecuentemente), muy elevados (por encima del 50 % y llegando incluso al 75 % en la serie de 25 repeticiones) que, con toda probabilidad, puede asociarse a la ejecución del Bucle 8. En ese sentido, es muy posible que la disfunción que se observe para las dos últimas series en esta parte del código (que, como se vio, empeora ostensiblemente en lugar de mejorar) esté íntimamente relacionado con este hecho. La prueba la pueden aportar las primera series, que funcionan bien y, a excepción de la de 20 repeticiones, ninguna de las tres anteriores sobrepasa el 50 %.

Con todo, la conclusión que se puede extraer del análisis de la GPU es que, ésta, se emplea durante el código con mucha más solicitación de la que lo hacía en el código base. Los niveles promedio aumentan y explican en buena parte por qué tanto la memoria, como la

RAM disminuyen en tramos clave del algoritmo. Además (aunque esto se discutirá mejor a continuación), concuerda con la disminución de los niveles de uso de la CPU, como se puede apreciar en la comparación entre las Figuras 5.2 y 5.6. Finalmente, permite atisbar y vislumbrar posibilidades de explicación a por qué se ralentiza tanto el código en el octavo bucle para las dos últimas series lo que, a la postre, repercute en un deslucimiento de la mejora que consolidaron las precedentes.

CPU

• Los promedios de uso (que son todos muy similares, sea cual sea el número de repeticiones) rondan los poco más de 21 %, frente al 32 % del código base. Además, para el tramo en el que se ejecuta el Bucle 6, mientras que en el código base el promedio es del 50 % y con picos aislados que sobrepasan el 60 %, en el paralelizado es del 30 % y el pico más alto (que se puede encontrar en la serie de 30 repeticiones) no llega al 45 %. Con estos datos, está claro que la CPU se beneficia de una "relajación" por la migración de parte de las líneas más pesadas a la GPU.

• El resto de tramos y las formas son muy similares en ambos códigos, el paralelizado y el sin paralelizar.

La conclusión que se extrae de todo lo anteriormente analizado es que, **tanto en la CPU como en la RAM y en la memoria física, el paralelizado beneficia sus solicitaciones,** permitiendo que el ordenador de sobremesa requiera usarlos en menor medida y, por ende, puedan dedicarse más recursos a otra serie de procesos que tenga abiertos también en ese momento (o, en caso de no tenerlos porque no era viable, es posible plantearse hacerlo cuando se paraleliza, siempre y cuando se tengan en cuenta qué niveles de uso y de memoria, respectivamente, hay disponibles para el usuario).

5.5. Comparativa CPU/GPU

El elenco de estudios que se va a introducir en los siguientes sub-apartados ha sido varias veces mencionado a lo largo de la memoria, con especial énfasis en lo que se lleva de capítulo. Antes de abordar el primero de ellos es importante matizar de nuevo que, cuando se haga alusión a la expresión "Comparativa CPU/GPU", el término CPU hará referencia al código base, según éste es procesado por la CPU. No significa que se vayan a comparar más las ventajas que pueda ofrecer el paralelizado CPU frente a las que lleva implícitas el paralelizado vía GPU. Esa cuestión ha quedado ampliamente dilucidada en los análisis de los tres anteriores apartados.

Nota: - En principio sí que existía una intencionalidad de llevar a cabo estudios comparativos entre ambos tipos de paralelizado, pero con todo lo que se ha ido viendo, la comparación (y la obtención del criterio de selección) se ha producido de forma natural debido a los muy pobres resultados que ha sido capaz de ofrecer el paralelizado vía CPU, que ni aporta mejoras en términos temporales ni las aporta en términos de recursos. En ese sentido, cualquier tipo de estudio que se pudiera realizar para comparar cualidades de los dos tipos de paralelización únicamente reportaría resultados triviales para el lector.

5.5.1. Sensibilidad frente a número de repeticiones

Tal y como ya se ha adelantado en un momento anterior a este punto, el estudio de sensibilidad del tiempo de cálculo, frente al número de repeticiones con las que trabaja el algoritmo de procesado de imágenes, interesa llevarlo a cabo para averiguar si podría tener lugar algún efecto positivo que permitiera que, con más cantidad de datos adquiridos, la GPU acelerara su velocidad de cálculo con respecto a la CPU, lo cual se ha evidenciado ya en otros trabajos **[20]** (ver Figura 4.4 y punto 4.4.1.1). Por ello, para este estudio paramétrico, se van a hacer referencias continuas a las tablas que se introdujeron en los tres apartados anteriores ya que, de hecho, dichas tablas se extrajeron de practicar este mismo estudio.

La idea es la de trabajar con los datos ya expuestos en las Tabla 5.3 y sacar todo el jugo que la gran cantidad de información alberga implícitamente. De este modo, se espera obtener conclusiones sólidas acerca de las cuestiones que se empezaron a plantear en el apartado 5.4, así como tratar de justificar por qué determinados bucles (como por ejemplo el 8) se comportan de esa manera y no de otra.

La Figura 5.9 muestra los tiempos totales de ejecución de los códigos base y paralelizado por GPU, respectivamente. La curva de tiempos totales asociada al algoritmo sin paralelizar muestra una correlación lineal muy fuerte. Esto indica que el aumento de las repeticiones no afecta en absoluto a la CPU (ni acelera ni enlentece su ejecución). Es decir, el efecto es aditivo, el tiempo de cálculo total crece a la par que las vueltas de los bucles por el aumento de las repeticiones. En general esto también se observa para el algoritmo paralelizado, si bien se aprecia como, a partir de 20 repeticiones, la pendiente se hace ligeramente más positiva, lo que muestra que el cálculo se enlentece (se pierde un poco esa divergencia que estaban exhibiendo las rectas)

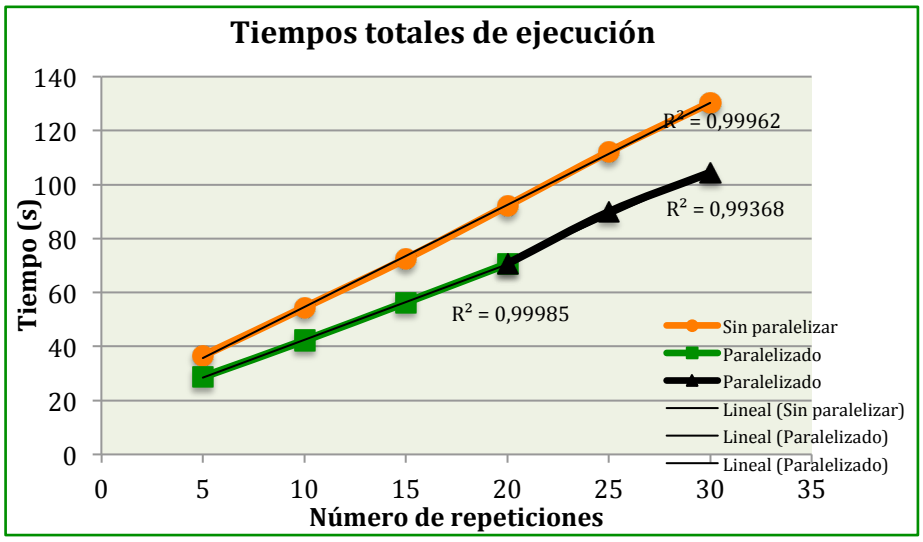


Figura. 5.9. Gráfica del tiempo total de ejecución frente a número de repeticiones para código base y paralelizado GPU.

Así pues, se tienen dos tramos: uno primero en el que la correlación es fuertemente lineal positiva, que va de las 5 a las 20 repeticiones y, otro, que va desde las 20 repeticiones hasta 30, en el cual la correlación es también muy lineal pero menos (sobretodo teniendo en cuenta que la primera recta está constituida por cuatro puntos y la segunda por tres).

Si se analiza ahora la diferencia de tiempos totales (lo que corresponde a las distancias entre cada par de puntos de las rectas en las 6 series de repeticiones), representada en la Figura 5.10, dado que primero las rectas van divergiendo marcadamente, la pendiente hasta las 20 repeticiones es muy alta. Luego decae abruptamente porque la divergencia de las rectas pasa a ser mucho más sutil.

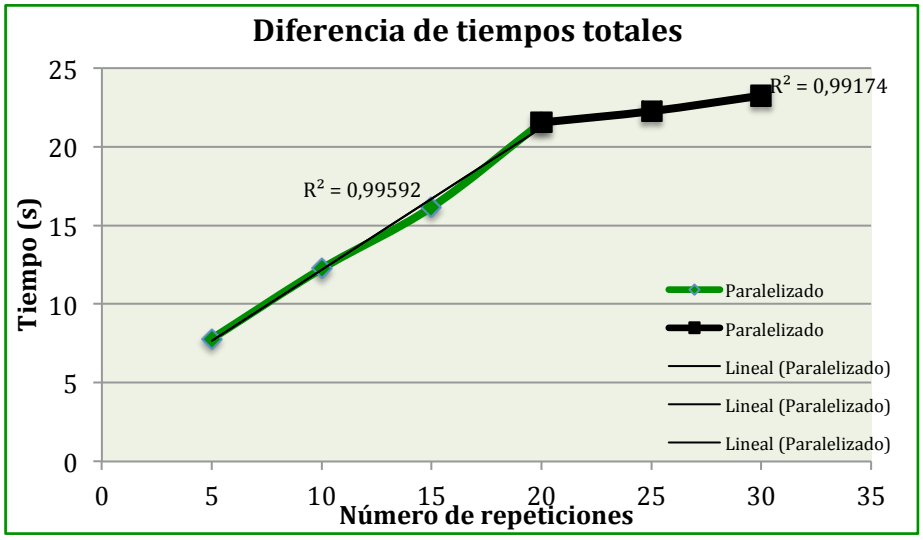


Figura. 5.10. Gráfica de la diferencia de tiempo total frente a número de repeticiones para código paralelizado GPU.

En ambos tramos existe una correlación lineal positiva, lo que confirma que el número de repeticiones no hace aumentar la diferencia de tiempos en cada tramo, sino que únicamente constituye un efecto aditivo: - A más repeticiones, se ejecuta más veces el primer bucle *for* para adquirir imágenes y construir la matriz *data* con una información visual más precisa (no varían sus dimensiones). De haber existido un efecto acelerador o de frenado, ello se habría traducido en curvas en lugar de rectas en la gráfica de la Figura 5.10. Y en la gráfica de la Figura 5.9, la recta del paralelizado divergiría de una forma no rectilínea, es decir, alejándose con una trayectoria curva cerrada de la recta naranja. O bien acercándose y llegando a un punto en el que, la mejora arrojada por la primera serie (5 repeticiones) se anulara, produciéndose un punto de corte. Dado que nada de esto se observa, está claro que no existe influencia.

Si se analizan ahora las mejoras en porcentaje que se consiguen en el código alfa, representadas en la gráfica 5.11, puede observarse que todo lo anteriormente descrito concuerda.

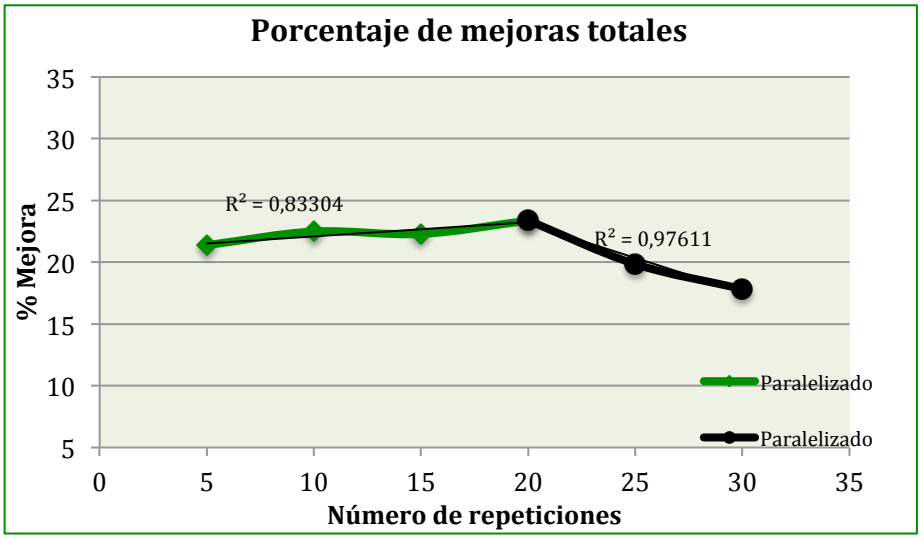


Figura. 5.11. Gráfica del porcentaje de mejoras totales frente a número de repeticiones para código paralelizado GPU.

En el primer tramo la mejora oscila alrededor de valores próximos al 22 % pero, esencialmente, se muestra constante. A partir de las 20 repeticiones se produce ese punto de inflexión que lleva a un rápido y corto decaimiento. En este sentido, el factor que hace pasar a la pendiente de positiva a negativa se halla claramente en el Bucle 8, por lo que merecerá la pena estudiar su evolución y discutir qué le ha podido suceder para invertir su comportamiento de tal forma.

En este orden de cosas, el siguiente paso en el estudio va a consistir en analizar las cosas, no desde el punto de vista de magnitudes totales, las cuales son absolutas y muchas veces ocultan información, sino que hay que ir a buscarlas ahora en las parciales, que se asocian a cada tipo de bucle individualmente. Solo así se podrá entender por qué la Tabla 5.3 arroja los resultados que arroja y, por supuesto, elucidar qué sucede con el bucle que estropea las mejoras ya consolidadas.

Se empezará por analizar las mejoras parciales, esto es, cómo evolucionan las contribuciones de cada bucle a la mejora global que se acaba de analizar en la figura anterior. En la Figura 5.12 pueden verse las mejoras asociadas a los bucles paralelizados y a los que no lo han sido. El hecho de que se hayan separado responde a que los comportamientos son similares entre cada homólogo, por lo que es más fácil describirlos.

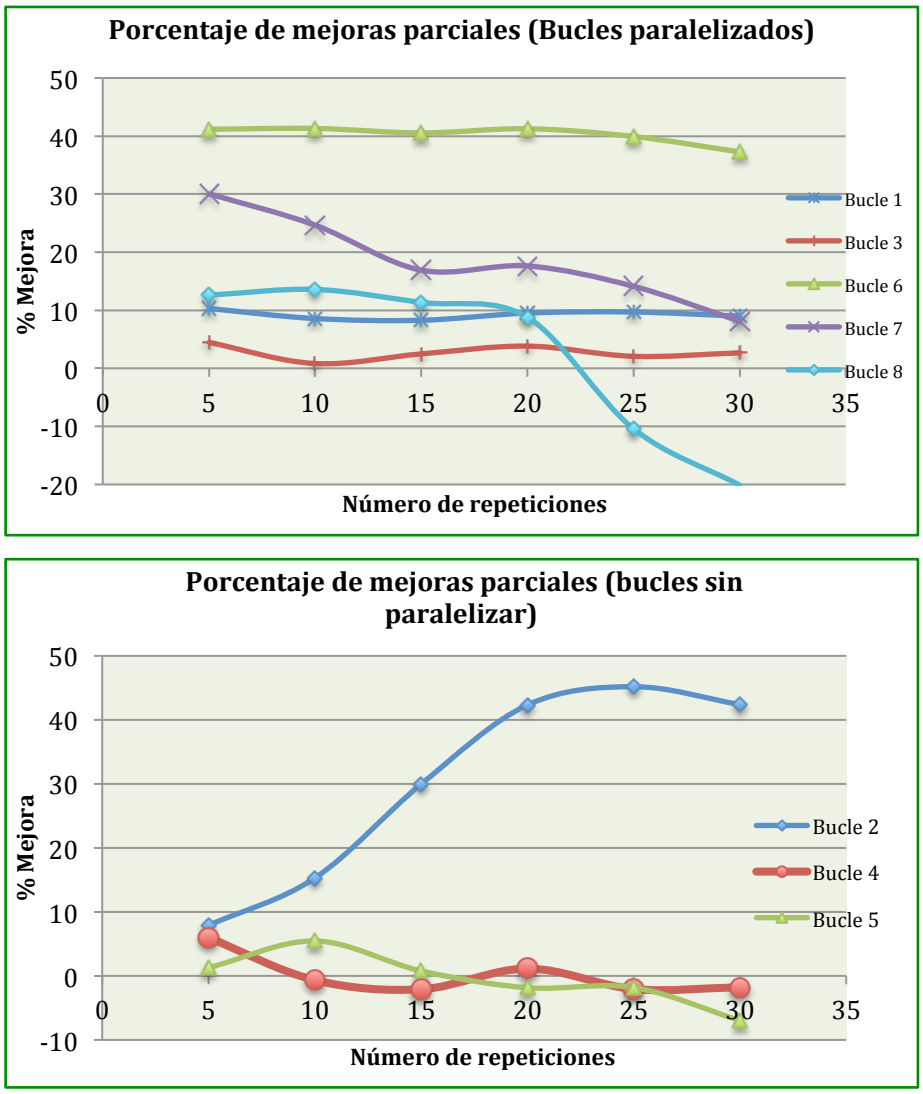


Figura. 5.12. Gráfica del porcentaje de mejoras parciales de: a) Bucles paralelizados. b) Bucles sin paralelizar.

Se pueden describir los siguientes comportamientos:

- **Bucle 1:** - Está paralelizado y no le afecta aumento de repeticiones.
- **Bucle 2:** - No está paralelizado y le afecta aparentemente el aumento de repeticiones.
- **Bucle 3:** - Está paralelizado y significativamente no le afecta el aumento de repeticiones.
- **Bucle 4:** - No está paralelizado y le afecta levemente el aumento de repeticiones
- **Bucle 5:** - No está paralelizado y le afecta levemente el aumento de repeticiones.
- **Bucle 6:** - Está paralelizado y no le afecta el aumento de repeticiones (el más estable).
- **Bucle 7:** - Está paralelizado pero exhibe comportamiento extraño.
- **Bucle 8:** - Está paralelizado pero exhibe un comportamiento extraño.

A la vista de las gráficas, puede verse visualizarse lo que en la Tabla 5.3 se comentó: los bucles 7 y 8 (sobretodo este último) exhiben un comportamiento anómalo. El Bucle 7 va disminuyendo paulatina pero preocupantemente a lo largo de las series, pero el Bucle 8 hace saltar las alarmas y requeriría de una explicación que hiciera comprensible por qué permanece tan estable hasta la cuarta serie y, a partir de la quinta, decae de esa forma tan estrepitosa, haciendo que el código alfa cojee con respecto al código base en las ejecuciones de ese tramo del script.

Para completar el compendio de gráficas de lo que los datos numéricos han arrojado, se mostrarán a continuación los tiempos parciales de forma separada: por un lado se analizarán primero los paralelizados que tienen un comportamiento lineal (en la Figura 5.12 mostraban una mejora estable, aproximadamente constante). El resto de los bucles se analizará por separado, debido a la diferencia de escala. Se muestran en la Figura 5.13 los tiempos parciales de cada dupla paralelizado (P, código alfa) y original o no paralelizado (SP, código base).

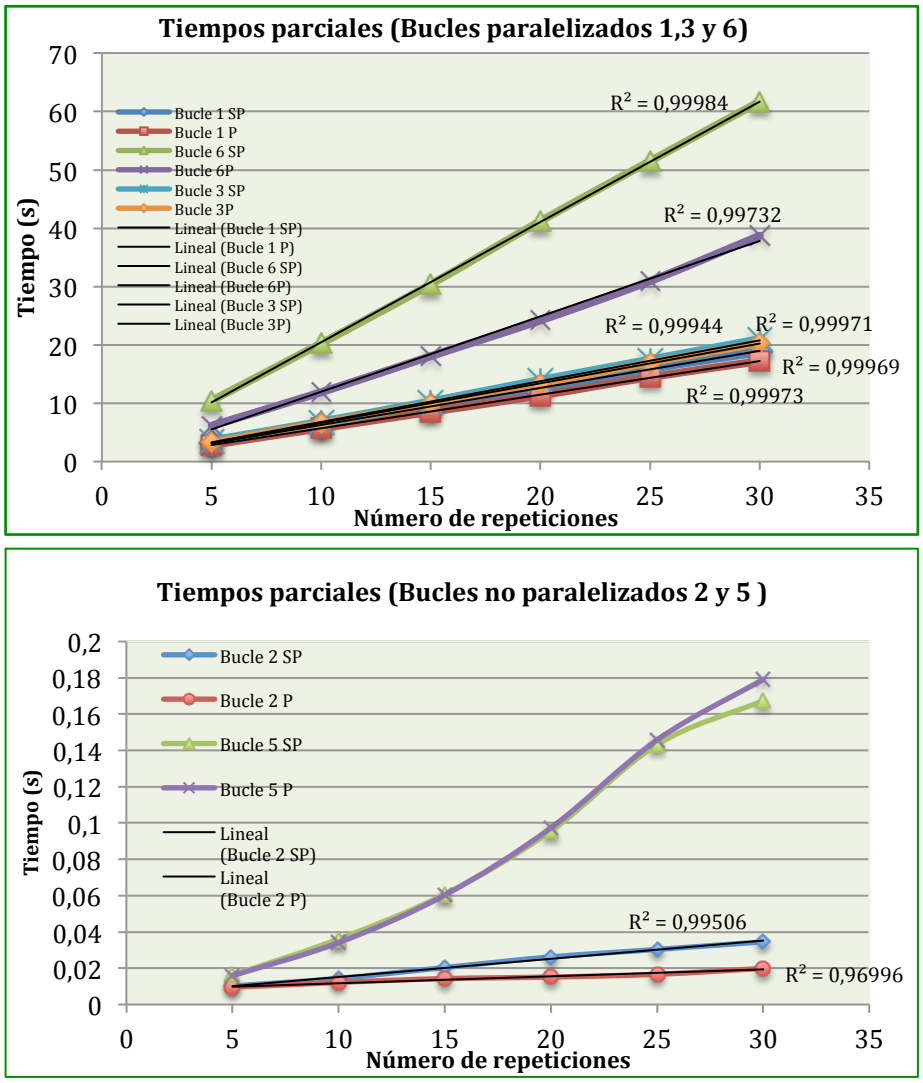


Figura. 5.13. Gráfica de tiempos parciales para: a) Bucles paralelizados con comportamiento lineal . b) Bucles sin paralelizar 2 y 5.

En la Figura 5.12 a) no se llegan a apreciar con claridad las rectas que se encuentran apiladas en la parte inferior por exhibir similares pendientes. Se trata de dos pares de rectas, las asociadas a los bucles 1 y 3 paralelizados junto a las de sus homólogos sin paralelizar. Se aprecia claramente como los tiempos parciales de este terceto de bucles tienden a aumentar linealmente con el número de repeticiones, tanto en el código base como en el código alfa. Que sean rectas, diverjan y con tanta linealidad (especialmente en el Bucle 6 paralelizado) confirma que, como se veía en la gráfica de mejoras parciales, no se produzca en estos bucles una mejora al aumentar las repeticiones (pues entonces se hubieran obtenido curvas logarítmicas en las paralelizadas, de forma que la divergencia no fuera constante).

Por su parte los bucles 2 y 5 no se han paralelizado y, pese a que el primero exhibía una evolución en la mejora preocupante, por los órdenes de magnitud en los que se mueven ambos no merece la pena profundizar demasiado en su análisis. Además, en la Figura 5.13 b) se aprecia como, en el caso del Bucle 2, su comportamiento es claramente lineal, lo que muestra que, efectivamente, las llamativas variaciones que experimenta en ocasiones de una serie a otra pueden asociarse a que, por ser un valor tan pequeño, podría estar sujeto a aleatoriedades. En el caso del Bucle 5, que ya no es tan lineal, los bucles que le preceden en el código, y que sí se han paralelizado, podrían estar actuando sobre variables que, insertadas luego en éste, afectan. Su pequeño valor está sujeto a que las fluctuaciones le haga variar mucho. Parece que el aumento de las repeticiones ejerza un aumento en el tiempo de cálculo, si bien lo inducen por igual en el código sin paralelizar como en el paralelizado, con lo cual no hay mejora ni deterioro (ya que las líneas se comportan exactamente igual y están muy próximas, casi solapadas).

Finalmente, en la Figura 5.14 se muestran los tiempos parciales que arrojan el resto de los bucles que quedan por analizar, entre los que se encuentra el más extraño de todos: el ocho.

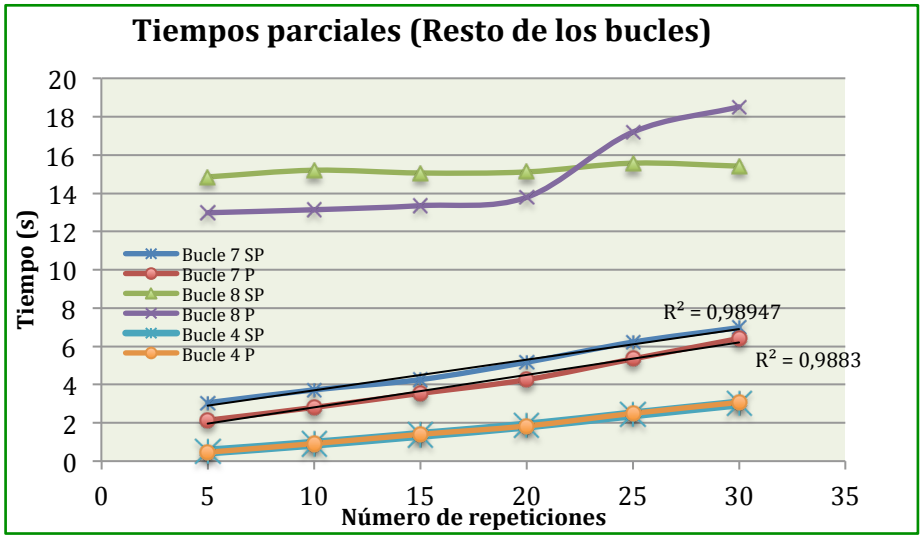


Figura. 5.14. Gráfica de tiempos parciales para los bucles 4, 7 y 8.

La mejora del Bucle 4 se vio que cruzaba el eje x varias veces, como sucede también con el 5 (de hecho muestran comportamientos especulares). No obstante, se aprecia , para todas las series, las evoluciones en el código base y en el alfa son coincidentes (como también ocurría con el 5). Además existe linealidad, lo que confirma que no está influenciada por el aumento

de las repeticiones. Que oscile ligeramente su mejora responde al hecho de que también se mueve en órdenes de magnitud pequeños y, por tanto, sujetos a aleatoriedades que introduce *MatLab* en el cálculo de una ejecución a otra. Para el bucle 7 se puede suscribir exactamente lo dicho para el 4.

Finalmente, en el tan esperado análisis del Bucle 8 se aprecia que sucede algo curioso, y es que hasta la repetición 20 el aumento, no es que no produzca una aceleración de los cálculos en el paralelizado sino que, de hecho, ni siquiera modifica el tiempo (ni mejora ni empeora).

Esto se explica porque es un bucle de plots, y no importa cuántas repeticiones se hayan tomado para captar las imágenes, pues el número de ellas que hay que representar es el mismo. Es decir, el Bucle 8 está completamente al margen de lo que se haga con el número de repeticiones. Ahora bien, por alguna razón, en el algoritmo paralelizado se aprecia con claridad como, a partir de esas 20 repeticiones, el tiempo del bucle empieza a dispararse, hasta el punto de que con 25 ya supera al tiempo que costaba la ejecución en el código base.

Aunque este viraje en el comportamiento es claro y rotundo, sí que sería de interés realizar estudios más concretos para averiguar si este factor de aumento en el tiempo del bucle sigue una tendencia alcista (en tal caso ver si es lineal, exponencial o cuadrática) o, si por el contrario, ha sido un bajón puntual que ha dado el rendimiento de la GPU y que poco a poco se va amortiguando, de forma que llegue un punto en el que también se estabilice (aunque sea a un valor de tiempo alto). Desafortunadamente, no se puede llevar a cabo porque la cámara rápida solo toma 30 repeticiones, que es con lo que, por otra parte, se suele trabajar en los experimentos. Por tanto es algo que, al no estar destinado a ser utilizado, tampoco representa un aspecto crítico.

En cuanto al motivo que pueda asociarse a este raro comportamiento, como se ha dicho el bucle de plots queda al margen en principio de lo que haga el que Bucle 1 y el resto con el aumento de repeticiones, si bien es posible que, incluir más repeticiones en la adquisición de las imágenes que se guardan en la matriz *data*, influya de alguna manera sobrecargando la memoria vídeo con las variables que se manejan en algoritmos cada vez más pesados.

Completado todo el análisis de los datos que se aportaron en la Tabla 5.3, la conclusión a la que se puede llegar es esencialmente la que se empezaba a atisbar al examinar las magnitudes totales (tiempos y mejoras): **Aumentar el número de repeticiones únicamente arrastra la mejora que se consigue en el algoritmo paralelizado, pero no ha quedado probado que significativamente hayan variaciones**.

El hecho de que la diferencia de tiempos sea cada vez mayor es engañoso, pues esto ocurre en términos absolutos. Sencillamente lo que ocurre es que, cuanto mayor es el número de repeticiones, a escala humana nosotros percibimos al paralelizar una diferencia de tiempo cada vez más representativa y útil de cara a la realización de otras labores. Por ilustrar un poco toda esta situación y materializar en qué se podrían traducir todas estas implicaciones, se propondrá el siguiente ejemplo:

- Supongamos que el grupo de Técnicas Ópticas tiene un acumulo de imágenes a procesar de una semana (5 días de ensayo en el *CMT*) con 40 casos por día. Como se ha visto, para 20

repeticiones (que serían las óptimas para conseguir la máxima mejora con el paralelizado), la diferencia de tiempos obtenida ha resultado ser de 21,53 segundos por caso (ver Tabla 5.1).

Por tanto se tendrá que, haciendo la sencilla operación: 21,53*40*5= 4306 segundos (71'77 minutos). Ese es un tiempo que, en términos sensoriales humanos, puede ser muy importante: - El ordenador se podrá tener libre una hora antes para poder realizar otras labores que requieran toda la RAM, carga de CPU y/o memoria disponible. O ponerse a analizar los resultados arrojados por las imágenes una hora antes y empezar a evaluar la cantidad de hollín, la penetración y demás tipo de parámetros importantes de la combustión, para extraer las conclusiones que se buscaban al plantear los experimentos. O bien ponerse a procesar la siguiente tanda de casos (quizá el mismo día). O incluso, sencillamente apagar el ordenador una hora antes (con el ahorro energético que supone).

Se podría caer en la tentación de procesar las imágenes con 30 repeticiones en lugar de con 20, pues la diferencia de tiempo asciende a algo más de 23 segundos por caso. Con eso se conseguirían unos 77 minutos (diez más que antes). Pero, en contraposición, mientras que así procesar los algoritmos paralelizados costarían unos 107*40*5 = 21400 segundos (357 minutos, aprox. 6 horas), llevar a cabo el mismo procesado con esas 20 repeticiones implicaría unos 71*40*5=14200 segundos (237 minutos, aprox. 4 horas). Son dos largas horas de diferencia y apenas se consiguen diez minutos más de ahorro, por lo que no merece la pena.

Como se ha visto, paralelizar esos 200 casos empleando el algoritmo de 20 repeticiones en lugar de hacerlo con el de 30 supone una ventaja que puede ser decisiva en múltiples casos. Por tanto, a mayor número de casos a procesar, tanto mayor será la diferencia de tiempo que se produce entre procesar de uno u otro modo. En este sentido, la conclusión práctica que extrae este estudio paramétrico es que, **siempre que la precisión y los experimentos lo permitan, realizar el procesado con 20 repeticiones por caso aportará el máximo ahorro de tiempo en el cálculo.**

5.5.2. Sinergias en el paralelizado

Se va a llevar a cabo un estudio, empleando la serie de cinco repeticiones[62], con el objeto de indagar si puede existir un efecto sinérgico en la ejecución de los bucles, en continuo dentro del código, en comparación a los tiempos que exhiben cuando la ejecución se realiza individual y aisladamente en cada uno de ellos.

La Tabla 5.4 recoge los resultados obtenidos con el algoritmo modificado para que proporcione cuarenta tomas de muestra por cada bucle aislado, esto es, sin ejecutar el script en su completitud (cabe recordar que los tiempos que arrojaron los bucles cuando fueron ejecutados en continuo dentro del código alfa están debidamente expuestos en la Tabla 5.3).

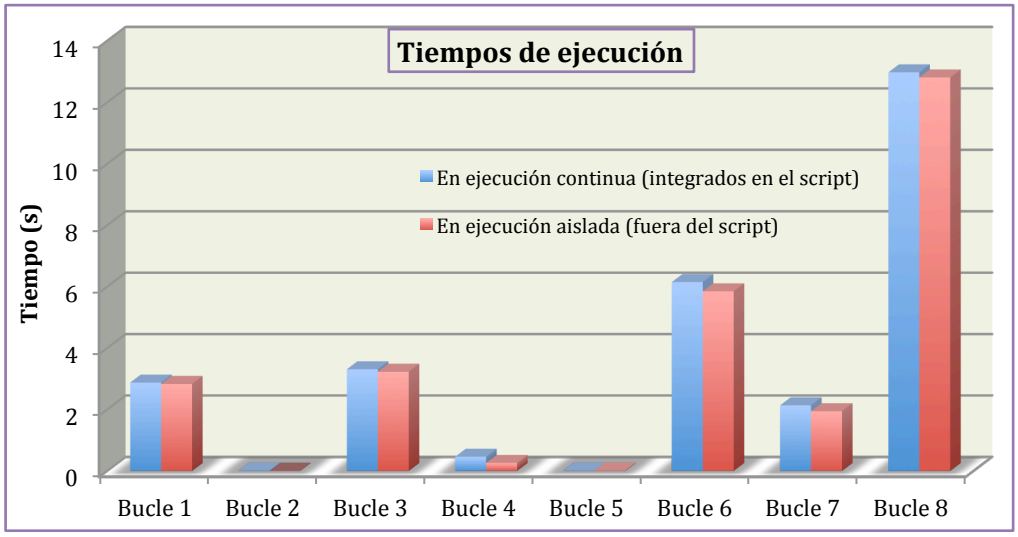
---

[62] Ya que es la más corta y con la cual se ha trabajado para el desarrollo del código alfa. La tendencia que se encuentre en esta serie se reproducirá por el resto de ellas. Además, estadísticamente el resultado será robusto, puesto que el tiempo medido de cada bucle estará constituido a partir de 40 muestras de tiempo, como siempre. Por tanto las conclusiones serán robustas.

Paralelizado					
	Tiempo en continuo (s)	Desv. St.	Tiempo en aislado (s)	Desv. St.	Porcentaje variación
Bucle 1	2,879	0,006	2,832	0,026	1,617
Bucle 2	0,009	0,000	0,003	0,0002	69,543
Bucle 3	3,31	0,036	3,228	0,02	2,468
Bucle 4	0,461	0,004	0,259	0,04	43,782
Bucle 5	0,016	0,001	0,014	0,006	14,539
Bucle 6	6,149	0,068	5,848	0,18	4,905
Bucle 7	2,135	0,010	1,945	0,16	8,912
Bucle 8	12,98	0,011	12,813	0,269	1,293
Total	**27,94**	**0,135**	**26,942**	**0,701**	**3,575**

Tabla 5.4. Resultados de los tiempos de ejecución para los ocho bucles en continuo y en aislamiento. En rojo se resaltan los bucles no paralelizados.

Con el fin de poder visualizar los datos de una forma más clara, se adjuntan en la Figura 5.15 tanto los tiempos como las mejoras que reportan cada tipo de ejecución.



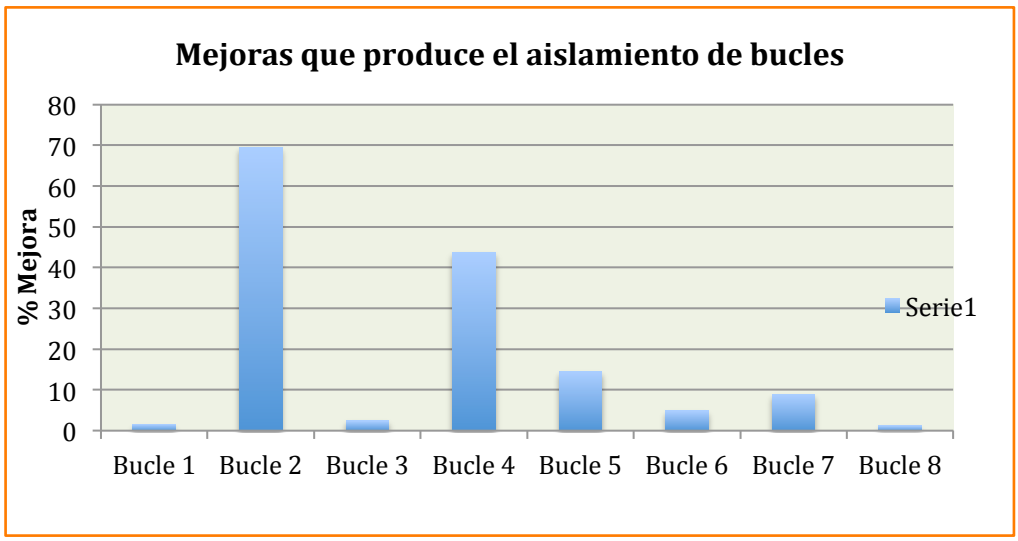


Figura. 5.15.a) Gráfica de tiempos por bucle en continuo y en aislado. B) Mejoras producidas por el aislamiento.

Se pueden ver varias cosas (todo son mejoras):

• Dentro de las mejoras, las más acusadas las hallamos en los bucles 2, 4 y 5, respectivamente. En el 2 y en el 5 las magnitudes son muy pequeñas, como puede apreciarse. Sin embargo, la robustez de haber tomado 40 muestras sostienen que esos porcentajes de mejora son reales. El hecho de que sean los bucles no paralelizados los que destaquen en mejoría con respecto al resto no es casualidad y, probablemente, vaya de la mano con el hecho de que a la CPU le cuesta menos procesar una y otra vez el mismo bucle sin necesidad de cooperar con la GPU, pues se producen migraciones de ejecución de líneas constantemente y en las dos direcciones[63]. Por eso, cuando ordenamos a un mismo bucle no paralelizado que se ejecute durante 40 vueltas no hay ninguna variable *gpuArray* y la CPU trabaja por su cuenta sin contar con la GPU.

• Para el resto de los bucles, que son paralelizados, en todos ellos se producen mejorías que varían entre el 1 y el 9 %. Es decir, son visibles pero no exageradas. Esto quiere decir que también cuando se produce una alternativa continua entre ejecuciones CPU-GPU, a los bucles les "sienta" mejor ser ejecutados individualmente. Podría explicarse en base a que se trabaja constantemente con las mismas variables (sean *gpuArray* y gestionadas por la GPU o *single*/*double* y gestionada por la CPU), así como también se emplean las mismas funciones y se realizan exactamente las mismas operaciones. Si se tiene en cuenta que los primeros valores tomados en las series de 40 tomas de muestra son siempre notablemente más elevados, la respuesta pueda encontrarse en que, de alguna manera, la CPU, la GPU (o ambas) crean algún tipo de enlace, elemento o memoria virtual, de un tiempo de uso muy corto (se crearían instantáneamente y su uso sería efímero) que les permite agilizar la ejecución.

La conclusión podría ser que, una hipotética ejecución del código, por adición de tiempo de cálculo individual de cada bucle, daría lugar a un tiempo total del algoritmo por caso de algo más de un 3 % de ahorro con respecto a la misma ejecución en continuo, lo cual no está mal. Ahora bien, cabe decir dos cosas: Primero, que esa mejora no es aplicable en términos prácticos, pues los algoritmos se ejecutan secuencialmente y en continuo. Segundo, el porcentaje de mejora para el caso paralelizado se diluye al comprobar que se obtiene uno muy similar para el caso sin paralelizar (cosa que se ha hecho aunque no se haya mostrado).

De todos modos, este estudio podría ser interesante si se tratasen de analizar en mayor profundidad mecanismos de transferencia de datos CPU-GPU, creación de posibles elementos de corto tiempo de vida, que sirven a los dispositivos como apoyo auxiliar, o demás aspectos, ya más puramente informáticos y de hardware, con el fin de vincular el efecto a la causa teórica exacta y adquirir más conocimiento sobre cómo operan a nivel íntimo tanto la CPU como la GPU.

---

[63] Recordar la programación heterogénea, que hacía alusión a que las líneas se ejecutaban indistintamente por la CPU o por la GPU, según fuese la clase de las variables asignadas y según sea la función empleada (que podría ser apta para paralelizado o no).

5.5.3. Coste computacional

Pese a que en el apartado 4.2 y 4.4 ya fueron expuestas las gráficas asociadas a los distintos recursos que se verían afectados por las ejecuciones, tanto del código base, como del código alfa, se dejó pendiente el planteamiento de una comparativa final que dilucidara y aportara pruebas cuantitativas acerca de en qué medida la paralelización en GPU mejora el código ejecutado en la CPU convencional.

En este sentido, ya ha quedado claro el beneficio que reporta la paralelización en términos de tiempo de cálculo (que a la postre era fijado como el parámetro crítico a optimizar al introducir los objetivos al inicio del documento). Ahora bien, el aspecto asociado al coste juega también su papel, pues frecuentemente los ordenadores equipados con la *NVidia GeForce GTX 960* en el grupo de Técnicas Ópticas (que hay dos) se encuentran muy demandados para múltiples aplicaciones que implican a un buen elenco de investigadores y alumnos. Por tanto, no solo ahorrar tiempo es un factor relevante, inducir que un determinado proceso (como es el caso del paralelizado que nos ocupa por ejemplo) pueda verse descongestionado a nivel de recursos de cálculo, puede suponer una gran ventaja. Por ejemplo, se podrían estar corriendo dos aplicaciones simultáneamente, una de diseño gráfico (que consume bastantes recursos) y la otra un procesado de imágenes. Esto siempre que en el procesado se apliquen las herramientas de paralelización aquí desarrolladas (y las que podrían venir gracias a trabajos futuros) y siendo cauteloso con los niveles de requerimientos que exige la segunda aplicación, contrastando con los ahorros que ofrece el paralelizado.

Sin más dilación y, una vez llevados a cabo los ensayos con MatLab[64] , las Figuras 5.16 y 5,17 recogen gráficamente lo que la Tabla 5.5 lo hace numéricamente. Se representan las señales promediadas, tanto en el código alfa como en el base, para la *RAM*, la *CPU* y la *GPU*. Constituye una aproximación a la medida de la cantidad de recursos que se emplean en la ejecución de los algoritmos. Por su parte, en la misma figura se incluye un extracto de los datos atribuidos exclusivamente al Bucle 6, que es el más relevante y el que más mejora ha reportado en la paralelización.

	Sin paralelizar			Paralelizado		
	RAM (MB)	**CPU (% Uso)**	**GPU (% Uso)**	**RAM (MB)**	**CPU (% Uso)**	**GPU (% Uso)**
	8615,824	29,300	12,039	8391,789	21,300	12,748
Desv. St.	3715,1563	17,079	3,775	3533,576	8,422	11,635

BUCLE 6			
	RAM (MB)	**CPU (% Uso)**	**GPU (% Uso)**
Paralelizado	11242,604	26,497	26,094
Sin paralelizar	11624,011	49,161	11,524

Tabla 5.5. RAM, CPU y GPU para: a) Promedios de señal (completas). b) Bucle 6.

[64] Se ha llevado a cabo con la serie de 20 repeticiones. La que ha demostrado conseguir la máxima mejora en el paralelizado.

Se desprenden las siguientes lecturas:

• En cuanto a la RAM, se observa un menor uso generalizado en el caso del algoritmo paralelizado. También en lo relativo al bucle 6. Esto es algo que se esperaba y se justificó en el apartado 4.4.

• En cuanto a la CPU, aquí la disminución de carga en la ejecución del código alfa con respecto al base es muy palpable, mejorando 8 puntos porcentuales. También fue justificada.

• En lo referente a la GPU, de forma generalizada aparentemente no hay un cambio demasiado visible, pero si se analiza la desviación estándar se entiende que la señal paralelizada tiene mucha más variabilidad, lo que hace que el promedio esté muy repartido, como muestra la gráfica de la Figura 5.16. En ese sentido, es clara la crecida de uso cuando se ejecuta el bucle 6 que es, como se ha reiterado en varias ocasiones, con mucho, el más pesado.

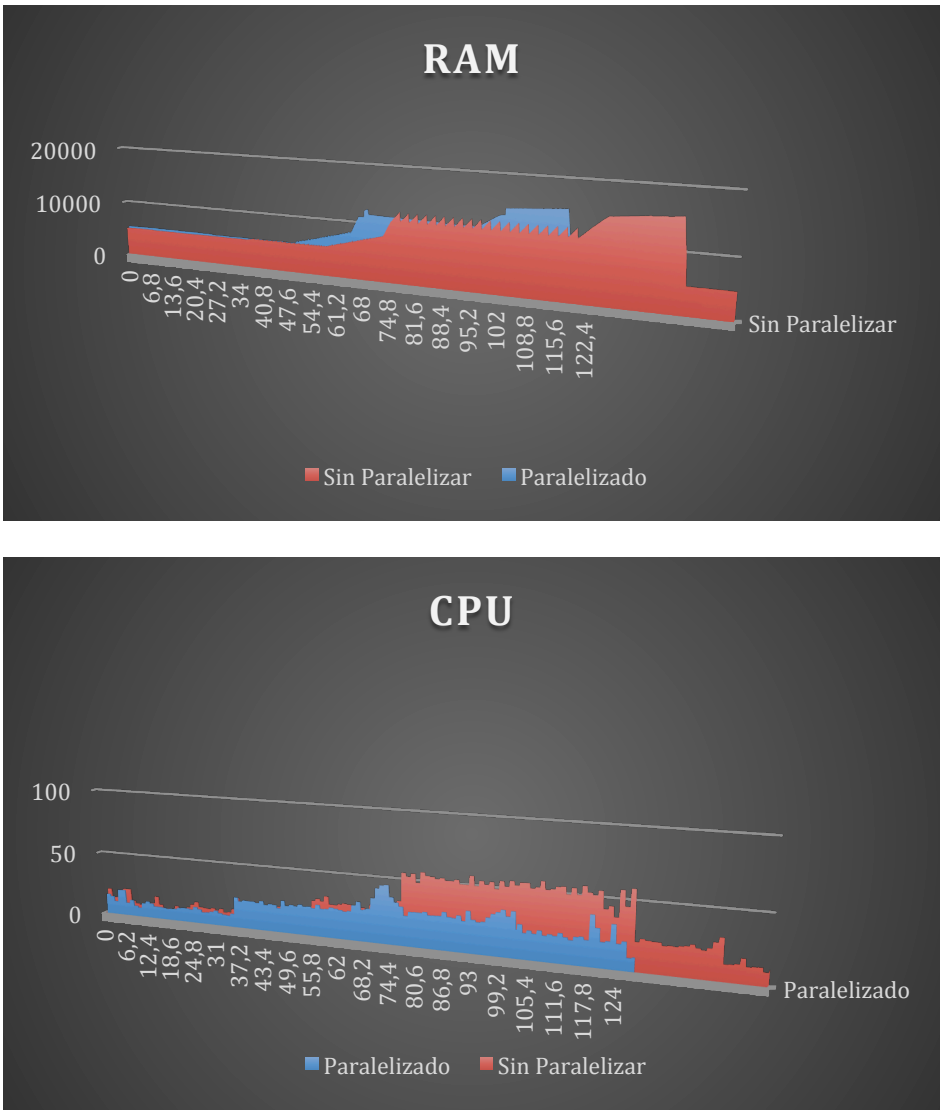


Figura. 5.16. Gráficas comparativas frente al tiempo de: a) RAM, b) CPU.

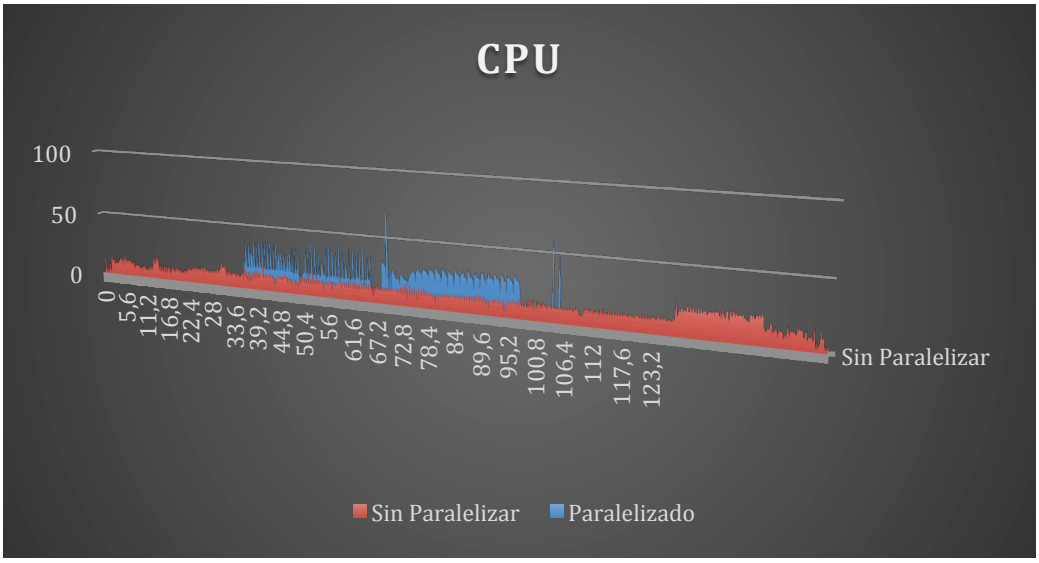
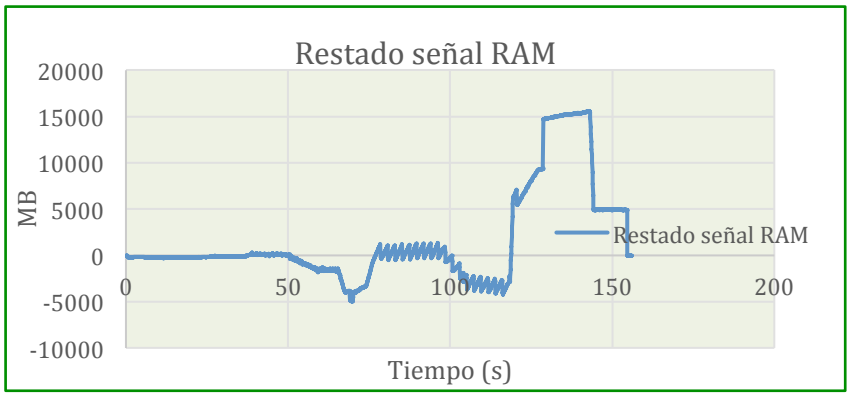


Figura. 5.17. Gráfica comparativas frente al tiempo de la GPU.

Para poner punto y final a este apartado de análisis se va a presentar el restado de señales que se ha obtenido entre el código alfa y el base. De esa forma se pueden afianzar y consolidar los análisis que se han ido haciendo acerca de los distintos elementos que conforman los recursos del ordenador o, de otro modo, el coste computacional. Se recogen en la Figura 5.18.



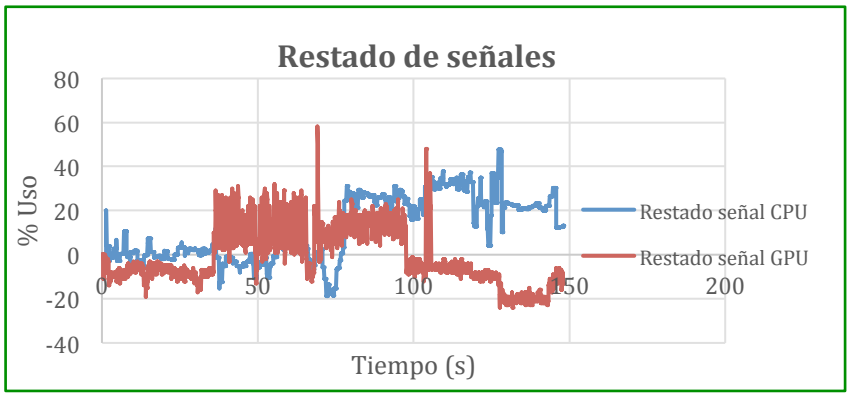


Figura. 5.18. Gráfica mostrando el restado de señal: a) RAM NO_Paral. – RAM_Paral. b) CPU NO_Paral. – CPU_Paral. c) GPU_Paral – GPU NO_Paral.

Se puede apreciar que la señal de RAM es ampliamente positiva en favor del código alfa, pues la diferencia de las señales arroja mucha mayor área positiva (mayor consumo para el

código base) que negativa (menor consumo para el código base). Existe un tramo intermedio, coincidente con la etapa "dentada" (por tanto asociada al Bucle 6) en la que el restado es nulo, lo que quiere decir que ambos códigos solicitan la misma cantidad de RAM en ese periodo.

Por su parte, la Figura 5.18 muestra también que la señal restada de la CPU es muy positiva a favor del código alfa, especialmente en el último tercio de la ejecución, que se corresponde con los Bucles 6 en adelante. En cuanto a la GPU, el restado se ha invertido a sabiendas de que es esperable que sea mucho mayor la que consume el código alfa que el base. Y eso es lo que se ha plasmado, ha salido positiva, lo que denota que, efectivamente, es el código paralelizado el que trabaja más. Parcialmente, las diferencias estriban en que la GPU trabaja mucho más en la zona central de la ejecución mientras que, tanto al principio como hacia el final del código, trabaja más la GPU en el código base.

De nuevo estos análisis numéricos (que en realidad ya no tenían nada de nuevo) corroboran lo que se ha estado analizando en el apartado 4.4. **Los recursos que consume el ordenador durante el paralelizado se ven notablemente disminuidos.**

5.5.4. Comparación final y elección del paralelizado

Cuando se planteó un boceto de índice para este trabajo, se pretendía llegar a este punto con la incertidumbre acerca de cuál de las dos vías de paralelización podría destaparse como la herramienta más adecuada aplicada al código base original con el que habitualmente el grupo de Técnicas Ópticas procesa las imágenes que adquiere mediante la técnica LEI. En ese sentido y, pese a que de acuerdo con la jerarquía de niveles impuesta, todo apuntaba a que la asociada con la GPU tenía que dar obligatoriamente un paso adelante, se distaba mucho de pensar cuán mala implementación iba a suponer la vía CPU.

En este contexto, este punto pretendía ser el marco de un análisis final en el que se pudieran comparar frente a frente cada una de las metodologías, ya no solo con respecto a la mejora que supusieran cada uno con respecto al código base original sino que, más allá de eso, las verdaderas fortalezas y carencias debían salir a relucir en este sub-apartado. Desgraciadamente no ha ocurrido como se planteó y, alcanzadas estas líneas y, después de todo lo que se ha dicho, aportar cualquier prueba adicional que demuestre lo muy por encima que está el uso de la paralelización vía tarjeta gráfica con respecto a la paralelización vía núcleos de la CPU, es dar vueltas sobre una cuestión totalmente esclarecida.

En ese sentido, simplemente hacer notar que, con una diferencia muy marcada, el método de paralelizado que ha demostrado ser capaz de practicar reducciones, no solo en el tiempo de cálculo (factor diferencial) sino también en los recursos, ha sido la paralelización de nivel 2. Así las cosas, se recomienda implementar esta herramienta en los algoritmos que procesen imágenes y realicen operaciones similares a las del código base con el que se ha trabajado a lo largo de este proyecto (que puede verse en el *Anexo B*).

6. CONCLUSIONES

6.1. Conclusiones y análisis de los objetivos conseguidos

El marco en el que se situaban los objetivos propuestos, al inicio de este documento, era el de tratar de aplicar las herramientas de paralelizado, existentes y accesibles en la actualidad, al optimizado de un código de procesado de imágenes, basado en la técnica óptica LEI, que el grupo del departamento habitualmente emplea cuando necesita llevar a cabo estudios sobre los procesos de combustión, especialmente en chorro Diésel. En ese sentido, cabe hacer notar que, sí se ha conseguido abarcar el objetivo marcado, puesto que los análisis que se han ido exponiendo a lo largo del Capítulo 5 han reflejado como, paralelizar el código base haciendo uso del *Parallel Computing Toolbox* de *MatLab*, con ayuda de una GPU dotada con arquitectura CUDA, han arrojado mejoras en la reducción del tiempo global de ejecución claramente por encima del 20 % para las cuatro primeras series, esto es, cuando se ejecuta el algoritmo empleando de entre cinco a veinte repeticiones por caso procesado. En el caso de emplear un número mayor de repeticiones, se ha visto que el resultado (que también mejora) se ve desafortunadamente deslucido, lo que podría ser explicado en base a que el buffer se colapsa y produce un lapso de espera.

En contrapartida, se ha podido comprobar que la paralelización mediante el uso del *Parallel Pool* de *MatLab* – que permite al procesador gestionar los recursos de modo que el cálculo sea paralelizado repartiendo la carga entre todos los núcleos o workers disponibles – no es útil aplicada al algoritmo de procesado de imágenes objeto de análisis. Los resultados son tan malos como inesperados pues, si bien este tipo de paralelización (al que se ha hecho llamar como de nivel 1) está indicada para algoritmos no tan complejos y en circunstancias en las que no se está aprovechando todo el nivel de recursos que ofrece cada núcleo, no ha permitido terminar de plantear el análisis paramétrico para estudiar cómo afectaría el aumento de las repeticiones. En ese sentido, con este método de paralelización el estudio terminó en las 15 repeticiones y arrojando resultados que confiere, a una hipotética comparación con otro método (cualquiera que éste sea) un muy escaso interés.

En lo que se refiere a la paralelización por vía GPU, haciendo uso de la gráfica NVidia, el estudio de sensibilidad al número de repeticiones sí que ha sido, no solo posible, sino exitoso. En su marco fue donde se obtuvieron los primeros resultados de mejora que impulsaron aun más el desarrollo definitivo del código alfa, lo que, a su vez, incentivó volver a realizarlos y consolidar así las mejorías que se empezaron a apuntar. Pese a que hubiese sido muy productivo un hipotético crecimiento de la mejora del paralelizado al aumentar el número de repeticiones por caso, desafortunadamente ésta únicamente demuestra ser arrastrada, esto es, se ha demostrado que no existe una significatividad estadística en la mejora buscada en ese sentido.

En otro orden de cosas, la comparación entre el código base original y el código alfa desarrollado vía GPU, ha probado que pueden obtenerse diferencias de tiempos de ejecución globales que, para grandes cantidades de casos a procesar en una única tanda, suponen una cantidad de tiempo (en términos absolutos) creciente geométricamente, gracias a que los

bucles paralelizados (especialmente el que se ha ordenado como el sexto) son internos, situándose dentro de otros más externos, responsables de hacer variar el caso procesado (pasar de uno al siguiente) así como de fechas (extraer casos acumulados de un fichero asociado a días de ensayos completos).

Finalmente y, en relación a parte de lo que ya se ha ido concluyendo, no ha sido necesaria la última prueba verificadora que arrojara cuál de las dos vías de paralelización implementadas (los niveles 1 y 2) se destapaba como idónea. Esta comparación se ha evitado finalmente en la memoria, dado que una mejoraba ostensiblemente los tiempos de ejecución del código base (vía GPU) y, la otra (vía CPU) ha arrojado unos resultados tan nefastos que ni siquiera permite que el código pueda trabajar con más de 15 repeticiones por caso procesado, lo que, por otro lado, es más que habitual en los experimentos y ensayos que lleva a cabo el equipo de Técnicas Ópticas.

Para cerrar este último capítulo, destinado a las conclusiones obtenidas en el presente TFG, merece la pena resaltar el buen resultado que ha arrojado también el código alfa en relación al coste computacional. Al inicio del proyecto se fijó con rotundidad que sería el tiempo de cálculo el parámetro discriminante. A la postre, ha sido decisivo para validar el potencial que exhibe la herramienta de paralelización, basada en la GPU y en su arquitectura CUDA, aplicada al caso particular del algoritmo con el que se ha trabajado, pero en lo referente a los recursos, los resultados obtenidos refuerzan todavía más este extremo. Se ha podido advertir como, efectuar las correspondientes migraciones de código a la GPU que supone la acción de paralelizar, se ha ganado más capacidad de RAM, más memoria física y mayor disponibilidad de uso de la CPU en comparación a cuando el código se ejecuta al natural. Esta, que en principio no era la cuestión diana, puede ofrecer la posibilidad de ejecutar dos aplicaciones potentes simultáneamente, que antes la computadora no podía en el tiempo que lleva el procesado (se ha visto que el nivel de recursos que exigía el algoritmo ya en su versión original era importante).

6.2. Perspectivas y trabajos futuros

Fuera ya de lo que es el marco que atañe a esta memoria, merece la pena hacer un último apunte en relación a las posibilidades que lleva implícitas la vía del paralelizado haciendo uso de la tarjeta gráfica. Como se ha ido viendo a lo largo del trabajo, de los tres posibles niveles de profundidad/complejidad con los que es posible abordar el problema del optimizado del algoritmo de procesado de imágenes, aquí únicamente se han explorado los dos primeros. De los cuales, uno de ellos (nivel 2) ha dado resultados productivos. Sin perjuicio de ello, lo que más alienta es que, si se prosigue el estudio por esta vía y se trata de profundizar todavía más en las herramientas de paralelizado que ofrece CUDA, puede abrirse un horizonte lleno de posibilidades en lo que a optimizaciones de algoritmos se refiere.

Se ha visto que los hilos son los que, de forma individual e independiente, ejecutan el Kernel en el que se implementan las líneas de código que interesa acelerar (comúnmente constituyentes de cuello de botella en ejecución secuencial por parte de la CPU). El nivel de programación que se requiere para llevar a cabo la implementación del nivel 3, exige un poco de especialización al usuario pues, si bien *MatLab* es una herramienta informática ampliamente utilizada en el ámbito de la ingeniería y la ciencia básica por su lenguaje de alto nivel, programar en C es un poco más duro y, sobretodo, específico. Si en el contexto de un proyecto algo más ambicioso se quisiera explorar esta vía, los resultados que han arrojado la que en este TFG sí se ha explotado son, no solo útiles, sino indicadores de que una mayor mejoría sería perfectamente adquirible. Es importante reflexionar sobre el hecho de que se ha conseguido mejorar, cerca de un 25 %, un código complejo, pesado y de muchas líneas, empleando una herramienta como *MatLab,* que no confiere al usuario la oportunidad de controlar y dar instrucciones de forma individual a los miles de hilos de que dispone cada núcleo de la GPU. Por esa razón, se insta encarecidamente a explorar esa vía, cuando el presupuesto, el tiempo y la disponibilidad del personal adecuado así lo permita.

Reseñar simplemente una cosa más y es que, los éxitos que se han conseguido con este algoritmo que procesa imágenes, sería muy razonable pensar que pudieran ser aprovechados y compartidos por otro tipo de algoritmos que lleven a cabo procesados y labores similares pues, al fin y al cabo, cualquier ámbito en el que se requiera captación de imagen y sonido, o de cualquier otro tipo de señal que procesar, podría verse altamente beneficiado. Podrían ser potenciales candidatos a algoritmo paralelizable, por ejemplo, los que se basan en seguimiento visual de objetos en movimiento.

7. BIBLIOGRAFÍA

[1] J. E. Dec, "*A conceptual model of DI Diesel Combustion based on Laser-Sheet Imaging*", SAE paper 970873, SAE Trans, 106 (1997), pp. 1319 – 1348.

[2] A. Bergek; C. Berggren, "*The Impact of environmental policy instruments on innovation: A review of Energy and Automotive industry studies*", Ecological Economics, 106 (2014) pp. 112 - 123.

[3] F. Payri and J. Desantes, "*Motores de Combustión Interna Alternativos*", 2011.

[4] A. P. Chust, Estudio del Proceso de Inyección Diesel mediante visualización y procesado de imágenes, Valencia: Universidad Politécnica de Valencia. 2001.

[5] Pinotti, M., Desarrollo y puesta a punto de un sistema óptico para el encendido de chorros diésel mediante plasma inducido por láser. 2013

[6] Turns, S.R. An Introduction to Combustion: Concepts and Applications. **Second Edition**, McGraw-Hill, New York, 2000.

[7] García, A. (2011). "*Estudio de los efectos de la post-inyección sobre el proceso de combustión y la formación de hollín en motores Diésel"*. Barcelona: Editorial Reverté S.A.

[8] Kamimoto, T., Kobayashi, H. "*Combustion processes in diesel engines*". **Progress in Energy and Combustion** Science.Vol. 17, Issue 2, pp. 163-189, 1991.

[9] V. D. LLopis, " *Estudio de nuevas estrategias para el control de la combustión en modos parcialmente premezclados en Motores de Encendido por Compresión*", Tesis doctoral (2013). Departamento de Máquinas y Motores Térmicos. Universitat Politècnica de València.

[10] Kokjohn, S.J. (2012) Reactivity controlled compresion ignition.(RCCI) combustion. Doctoral Thesis University of Wisconsin 2012.

[11] Starck, L., Lecointe, B., Forti, L., Jeuland, N. Impact of fuel characteristics on HCCI combustion: Performances and emissions. **Fuel**, Vol. 89, Issue 10, pp. 3069-3077, 2010.

[12] L. D. Z. Pemberthy, *Caracterización de los Procesos de Inyección – Combustión Diésel mediante Visualización y Procesado Digital de Imágenes*. Tesis Doctoral, Valencia: Universidad Politécnica de Valencia, 2010.

[13] Planck, M., (1948) "The Theory of Heat Radiation", P. Blakiston's Son & Co.

[14] Hottel H.C. y Broughton F.P."*Determination of true temperature and total radiation from luminous gas flames*" Industrial and Engineering Chemistry, vol. 4, pp. 166-175,1932

[15] C. Monin, *Caracterización del Proceso de Formación de Hollín en una Llama de Difusión Diésel de Baja Temperatura*. Tesis Doctoral, Valencia: Universidad Politécnica de Valencia, 2009.

[16] Siebers D.L. and Pickett L.M. "*Injection pressure and orifice diameter effects on soot in DI*

diesel fuel jets" Thiesel 2002 Conference on thermo- and fluid Dynamic Processes in Diesel Engines

[17] Musculus M.P.B. and Pickett L.M. "*Diagnostics considerations for optical laser-extinction measurements of soot in high-pressure transient combustion environments*" Combustion and Flame 141, pp. 371-391 (2005)

[18] J. V. Pastor; J. M. García-Oliver; R. Novella; T. Xuan; "Soot Quantification of Single – Hole Diesel Sprays by Means of Extinction Imaging", SAE International (2015)

[19] C. Chicano, *Cálculo paralelo con CUDA y Matlab en el seguimiento visual de objetos*. Trabajo de Fin de Grado, Escuela Técnica Superior de Ingeniería. Universidad de Sevilla, 2015.

[20] J. Straus, *Introduction to Parallel/GPU computing using MATLAB. Virtual Simulation Lab seminar series*. Norwegian University of Science and Technology, 2017.

[21] F. Rodríguez, *Programación Matlab en Paralelo Sobre Clúster Computacional: Evaluación de Prestaciones*. Proyecto de Fin de Carrera, Cartagena: Universidad Politécnica de Cartagena, 2010.

[23] M. Ujaldón, *Programación de GPUs con CUDA*. Seminario impartido en la Universidad de Chile. Profesor Titular de la Universidad de Málaga, 2016.

[24] Harald Brunnhofer. Mathematician Ph. D. of Virginia Polytechnic Institute and State University. MatLab tutorials: https://es.mathworks.com/videos/parallel-computing-tutorial-product-landscape-1-of-9-91563.html

[25] L. García, *Cómo acelerar aplicaciones de MATLAB*, Application Engineer. MatLab tutorials: https://es.mathworks.com/videos/speeding-up-matlab-applications-99194.html

8. PRESUPUESTO

8.1. Introducción

En este apartado se recogen todos los costes asociados a la ejecución de este trabajo de investigación. Para ello se tendrán en cuenta tanto los costes asociados a mano de obra como los asociados a los materiales.

El coste referido a la mano de obra agrupa los costes debidos al trabajo del personal que ha intervenido en la realización de este trabajo. Los datos son obtenidos a partir de los costes presupuestados según el instituto CMT-Motores Térmicos.

Cuando se habla del coste de materiales solo se tiene en cuenta la amortización de los mismos, que supone el coste asociado al tiempo que cada material ha estado inmovilizado debido a su requerimiento para este trabajo, no solo al tiempo de uso.

Finalmente se establece un presupuesto total, que tienen en cuenta los costes indirectos, el beneficio industrial y el IVA.

8.2. Coste de mano de obra

Primeramente, se presentan los costes asociados a mano de obra del proyecto. Para el cálculo de los diferentes salarios se consideran 52 semanas al año, con una disponibilidad de 6 semanas de vacaciones, y siendo la duración de la semana laboral de 40 horas. Se muestra a continuación en desglose de cálculo salarial para cada uno de los puestos de trabajo implicados.

Ingeniero Industrial – Catedrático	
Horas/año [horas]	1840
Salario bruto [€]	112792
Precio de hora trabajada [€/hora]	61.3
Ingeniero Industrial – Doctorando del departamento	
Horas/año [horas]	1840
Salario bruto [€]	71208
Precio de hora trabajada [€/hora]	38.7
Ingeniero Técnico Industrial - Técnico	
Horas/año [horas]	1840
Salario bruto [€]	50784
Precio de hora trabajada [€/hora]	27.6
Ingeniero Técnico Industrial - Becario	
Horas/año [horas]	1840
Salario bruto [€]	27600
Precio de hora trabajada [€/hora]	15

En base a estos costes se pueden calcular los costes asociados a la mano de obra, que son repartidos para cada tarea principal.

MANO DE OBRA - PROCESADO DE DATOS			
Operario	**Coste Unitario [€/horas]**	**Cantidad [horas]**	**Total [€]**
Doctorando	38.7	100	3870
Becario	15	200	3000
Coste total Mano de Obra – Procesado de datos			6870

MANO DE OBRA - ANALISIS DE DATOS			
Operario	**Coste Unitario [€/horas]**	**Cantidad [horas]**	**Total [€]**
Catedrático	61.3	20	1226
Doctorando	38.7	50	1935
Becario	15	100	1500
Coste total Mano de Obra – Análisis de datos			4661

Se procede entonces a reunir los costes en un presupuesto total de mano de obra:

PRESUPUESTO - MANO DE OBRA	
Concepto	**Total [€]**
Procesado de datos	6870
Análisis de datos	4661
TOTAL[€]	**11531**

8.3. Coste de material

Seguidamente, se muestran los costes asociados a las instalaciones de las cuales se han obtenidos los datos analizados. Se recogen los costes de la instalación motor maqueta, los equipos auxiliares, los equipos electrónicos e informáticos.

COSTE MATERIAL			
Elemento	**Coste Unitario [€/ud]**	**Cantidad[ud]**	**Total [€]**
Ordenador + NVidia GeForce GTX 960	1000	1	1000
Matlab	2000	1	2000
Coste total Material			**3000**

Se procede entonces a calcular el precio de la amortización de los costes materiales asociados al proyecto.

COSTE AMORTIZACIÓN MATERIAL				
Elemento	**Coste [€]**	**Vida útil [años]**	**Periodo amortizado [horas]**	**Importe [€]**
Ordenador + NVidia GeForce GTX 960	1000	5	1440	32,88
Matlab	2000	1	1440	331,03
Coste total Amortizado	**363,91**			

8.4. Presupuesto total

A partir de todos los presupuestos parciales obtenidos se puede obtener el coste total de este trabajo:

PRESUPUESTO TOTAL	
Concepto	**Coste [€]**
Mano de obra	11531
Costes indirectos (10%)	1153,1
Costes beneficio industrial (5%)	576,55
Subtotal	**13260,65**
IVA (21%)	16045,39
Materiales	363,91
Coste Total	**16409,3**

El coste total del proyecto asciende a **DIECISEIS MIL CUATROCIENTOS NUEVE CON TRES CÉNTIMOS**

ANEXOS

ANEXO A. Código Base

```
clc
clear all
close all
warning off

% addpath(genpath('D:\Pablo\LIP\MATLAB Functions'));%D:\MatlabCMT\CMTToolbox

%% INPUTS
mainpath='D:\Ejemplos_Omar\2process';
date=dir(mainpath);
pixmm = 11.25;
NumBI = 4;% images number without injection of each repetition
FramesLess=0;
width=13*pixmm;
FlameControl=1.6;
LEDcontrol=1800;                                          %%%%%%%%% Pablo cambia 1800
por 1450 o 950
diffSat= 60;% the limitation of the saturated value of the difference between total
intenstiy and flame intensity
minimum_noise=20;
Image_plot_repetition= 'off';
Image_plot_avg= 'on';
Save_Reps= 'off';
% xinjector = 157;%114 LD123 LT117 NO5050_157
% yinjector =16; %NO5050_16


 for x=3:length(date)
      date_path=strcat(mainpath, '\', date(x).name);
      cd(strcat(date_path,'\Photron'))
      cases=dir;

     for t=3:3

%          %ERRORS
%          if x==4&&(t==10||t==12||t==13||t==19)||x==5&&t==3       %7449 error
%              error=1;                                                %%Added in line
196
%          elseif x==3&&(t==6||t==7||t==17)                        %32 erorr   No se sabe
t==14 18 19 20 22
%              error=2;
%          elseif x==3&&(t==14||t==22)%||t==18||t==19||t==20)
%              continue
%          else
%              error=0;
%          end
error=0;

        cases_path=strcat(date_path,'\Photron\',cases(t).name);
        cd(cases_path);
        display('Now Processing:'); display(cases(t).name, date(x).name);
        results=strcat(cases_path, '\ResultsTia');
        results_rep=strcat(cases_path, '\ResultsTia\ResultsXrep');

        %% Processed Controls
        % Check if any repetition has alredy been processed or if the full
        % case has. Useful when lounching various cases in a row

        plot_ctrl_avg=dir(strcat(results,'\Avg_*.txt'));
        processed_control_avg=dir(strcat(results,'\KL_LEI_Avg.mat'));
        processed_control_conditions=dir(strcat(results,'\Case_Conditions_*.mat'));
        processed_control_rep=dir(strcat(results_rep,'\KL_LEI_XRep.mat'));

        if strcmp(Image_plot_avg,'on')
            if ~isempty(processed_control_avg)&&~isempty(plot_ctrl_avg)
                continue
            end
        else
            if ~isempty(processed_control_avg)
                continue
            end
        end

        %% Processing
```

```
        if ~isempty(processed_control_conditions)
            load(strcat(results,'\',processed_control_conditions.name));
            display('Processing over Repetition already done.');
        else

            ind=strfind(cases_path, '\');
            conditions = cases_path(ind(end)+1:end);
            ind_cond=strfind(conditions, '_');
            Technique=conditions(1:ind_cond(1)-1);
            Fuel=conditions(ind_cond(1)+1:ind_cond(2)-1);
            Engine_Condition=conditions(ind_cond(2)+1:ind_cond(3)-1);
            Inj_Pressure=conditions(ind_cond(end)+1:end);

            Ents = getarchivos('ConfigCamara.ent');
            if ~isempty(Ents)
                input = leeropciones(Ents{1});
                display(ParentFolder)
                display('Warning: Camera setting file already exists!')

            else
                ConfigCamera('Photron','mraw');
                Ents = getarchivos('ConfigCamara.ent');
                input = leeropciones(Ents{1});
            end

            RepNum = input.NRep;% repetation number
            NumPerRep = input.NumPerRep; %-FramesLess;   % images number of each
                                                    repetation need to be processed
            Framerate = input.FrameRate;
            xsize = input.ancho;
            ysize = input.alto;
            if error==2
                RepNum=29;
            end

            mkdir(results);
            mkdir(results_rep);

            %% Injector Nozzle Position Detection

            inycheck=dir('Inypos.mat');
            if ~isempty(inycheck)
                load('Inypos.mat');
                display(ParentFolder)
                display('Warning: Injector Position data already exists!')
            else
                bright_ref=0;

                for k=1:RepNum
                    [Images,Times]=GetMovie(0,k);

                    for kk=1:NumPerRep
                        bright=mean(mean(Images(:,:,kk)));
                        if bright>bright_ref
                            bright_ref=bright;
                            ref_pos=kk;
                            ref_rep=k;
                        end
                    end
                end
                [Images,Times]=GetMovie(0,ref_rep);
                imshow(Images(:,:,ref_pos), [0 max(max(Images(:,:,ref_pos)))]);
                set(gcf,'Name','Select the Injector Positoin on the Image');
                inyposXY=round(ginput(1));
                close gcf
                clear Images Times
                save('Inypos.mat', 'inyposXY');
            end
            xinjector=inyposXY(1);
            yinjector=inyposXY(2);

            %% Background Light Reference and Noise Level Reference Determination.
Autodetection of LED ON and LED OFF images
```

BUCLE 1

```
%% Combustion Detector added to same loop //

for n=1:RepNum
    data =GetMovie(0,n,'Photron','mraw',NumPerRep,Framerate);
    m=1;
    g=1;
    for c=1:NumBI
        if max(max(data(yinjector:yinjector+200,:,c)))>LEDcontrol
            Iback{n,m}=data(:,:,c);
            m=m+1;
        else
            Inoise{n,g}=data(:,:,c);
            g=g+1;
        end
    end
    if size(Iback,2)~=size(Inoise,2)
        if size(Iback,2)<size(Inoise,2)
            for d=NumBI+1:NumPerRep
                if max(max(data(yinjector:yinjector+200,:,d)))>LEDcontrol
                    Iback{n,m}=data(:,:,d);
                    break
                end
            end
        else
             for d=NumBI+1:NumPerRep
                if max(max(data(yinjector:yinjector+200,:,d)))<LEDcontrol
                    Inoise{n,size(Inoise(n,:),2)}=data(:,:,d);
                    break
                end
             end
        end
    end

    %% Combustion Detector

    Inoisesum_rep=zeros(ysize,xsize);
    for kk=1:length(Inoise(n,:))
        Inoisesum_rep=Inoisesum_rep+double(Inoise{n,kk});
    end
    Inoiseavg_rep=Inoisesum_rep./length(Inoise(n,:));
    InoiseRef_rep=max(max(Inoiseavg_rep(ysize/2:end,:)));
    if InoiseRef_rep==0
        InoiseRef_rep=minimum_noise;
    end
    for j=NumBI:NumPerRep
        if max(max(data(yinjector:yinjector+200,:,j)))<LEDcontrol
            if max(max(data(ysize/2:end,:,j)))>InoiseRef_rep*FlameControl;
                FireRepCount(n)=1;
                break
            end
        end
    end
end

    if ~exist('FireRepCount')
    cd(results)
    fid=fopen('flag.txt', 'wt');
    fprintf(fid, 'No Combustion Repetition Found in this Case');
    fclose(fid);
    continue
end

if error==1
    FireRepCount(30)=0; %%Error in data acquisition
end

CombustionCount=sum(FireRepCount)/RepNum*100;
Rep2Process=find(FireRepCount==1);
Ibacksum = zeros(ysize,xsize);
Iback_count= 0;
```

BUCLE 2

```
for i=1:size(Iback,2)*size(Iback,1)
    if ~isempty(Iback{i})
        Ibacksum=Ibacksum+double(Iback{i}(:,:));

          %sum1=sum1+double(Iback1{i}(:,:));
```

```
        Iback_count=Iback_count+1;
            end
        end
        Ibackavg=Ibacksum./Iback_count;
        Inoisesum = zeros(ysize,xsize);
        Inoise_count= 0;
       for i=1:size(Inoise,2)*size(Inoise,1)
            if ~isempty(Inoise{i})
                Inoisesum=Inoisesum+double(Inoise{i}(:,:));

                  %sum1=sum1+double(Iback1{i}(:,:));

                Inoise_count=Inoise_count+1;
            end
       end

        Inoiseavg=Inoisesum./Inoise_count;
        InoiseRef=max(max(Inoiseavg(ysize/2:end,:)));

        %% LEI Image Processing
        % Actual Image Processing for LEI technique

        LEDcount=zeros(RepNum,NumPerRep);
        BLKcount=zeros(RepNum,NumPerRep);
```

BUCLE 3

```
        for r=1:length(Rep2Process)
            n=Rep2Process(r);
            data = GetMovie(0,n,'Photron','mraw',NumPerRep,Framerate);

            for Nimg=1:NumPerRep
                if max(max(data(yinjector:yinjector+100,:,Nimg)))>LEDcontrol; %check
                                                    if there is LED in the "f" Image
                    LEDcount(n,Nimg)=1;
                    %                       BLKcount(n,Nimg)=0;
                    TimeLED(n,Nimg)=1000000/Framerate*Nimg;
                    %             if Nimg>NumBI
                    Itotal{n,Nimg}=data(:,:,Nimg);
                    %             end
                else
                    %                       LEDcount(n,Nimg)=0;
                    BLKcount(n,Nimg)=1;
                    TimeBLK(n,Nimg)=1000000/Framerate*Nimg;
                    %             if Nimg>NumBI
                    Iflame{n,Nimg}=data(:,:,Nimg);
                    %             end
                end
            end
        end

        %creat a mask

        mask = zeros(ysize,xsize);
        mask = Ibackavg>0.03*max(max(Ibackavg));
        IbackavgFin = Ibackavg.*mask;

        %% Time interpolation Applied on Images
```

BUCLE 4

```
        for r=1:length(Rep2Process)

            i=Rep2Process(r);
            c=1;
            d=1;
            for j=NumBI+1:NumPerRep
                if LEDcount(i,j)==0
                    if ~isempty(find(LEDcount(i,j+1:end)==1,1,'first'))
                        Itotalint{i,j}(:,:)=(Itotal{i,j-1}+Itotal{i,j+1})./2;
                        TimeLEDint(i,j)=(TimeLED(i,j-1)+TimeLED(i,j+1))/2;
                    end
                else if BLKcount(i,j)==0
                        if ~isempty(find(BLKcount(i,j+1:end)==1,1,'first'))
                            Iflameint{i,j}(:,:)=(Iflame{i,j-1}+Iflame{i,j+1})./2;
                               TimeBLKint(i,j)=(TimeBLK(i,j-1)+TimeBLK(i,j+1))/2;
                        end
                    end
                end
            end
        end
```

```
            ItotalFINAL{length(Rep2Process),NumPerRep-NumBI}=0;
            IflameFINAL{length(Rep2Process),NumPerRep-NumBI}=0;
            TimeLEDfinal=zeros(length(Rep2Process),NumPerRep-NumBI);
            TimeBLKfinal=zeros(length(Rep2Process),NumPerRep-NumBI);
```

BUCLE 5

```
            for r=1:length(Rep2Process)

                i=Rep2Process(r);
                %     c=1;
                %     d=1;
                %     e=1;
                %     f=1;
                for j=NumBI+1:NumPerRep
                    if LEDcount(i,j)==1
                        ItotalFINAL(i,j)=Itotal(i,j);
                        TimeLEDfinal(i,j)=TimeLED(i,j);
                        %             c=c+1;
                        if j<=length(Iflameint(:,:))
                            IflameFINAL(i,j)=Iflameint(i,j);
                            TimeBLKfinal(i,j)=TimeBLKint(i,j);
                            %                 d=d+1;
                        else
                            IflameFINAL(i,j)=Iflame(i,j-1);
                            TimeBLKfinal(i,j)=TimeBLK(i,j-1);
                        end
                    else
                        IflameFINAL(i,j)=Iflame(i,j);
                        TimeBLKfinal(i,j)=TimeBLK(i,j);
                        if j<=length(Itotalint)
                            ItotalFINAL(i,j)=Itotalint(i,j);
                            TimeLEDfinal(i,j)=TimeLEDint(i,j);
                        else
                            ItotalFINAL(i,j)=Itotal(i,j-1);
                            TimeLEDfinal(i,j)=TimeLED(i,j-1);
                        end
                    end
                end
            end

            cd(results);
            savename=strcat('Case_Conditions_', date(x).name, '_', conditions,'.mat');

        end
            %% KL calculation and Saving - PER REPETITION
            %if isempty(processed_control_rep)&&strcmp(Save_Reps,'on')

                cd(results_rep);
```

BUCLE 6

```
                for r=1:length(Rep2Process)
                    i=Rep2Process(r);
                    for j=NumBI+1:NumPerRep
                        %             Flamepos=find(TimeBLK(i,:)==TimeBLKord(i,j));
                        %             mask2=zeros(ysize,xsize);
                        %             mask2=Iflame1{i,Flamepos}>60;

                        %             diff=(double(Itotalint{i,j})*mask)-
(double(Iflame1{i,Flamepos}*mask2));
                        a=double(ItotalFINAL{i,j});
                        b=double(IflameFINAL{i,j});
%                             c=double(b);
                        diff=double(a-b);
                        KL_rep{j}=log(complex((IbackavgFin./diff)));
                        KL_Axis_rep{j}=KL_rep{j}(1:70*pixmm,xinjector);
                        KLsat_rep{j}=log((IbackavgFin./diffSat).*mask); %saturated KL
value
                        KLsat_Axis_rep{j}=KLsat_rep{j}(1:70*pixmm,xinjector);


                    end
                    save(strcat('KL_LEI_Rep_',num2str(i),'.mat'),'KL_rep', 'KLsat_rep',
'KL_Axis_rep', 'KLsat_Axis_rep', '-v7.3');
                    clearvars KL_rep KLsat_rep KL_Axis_rep KLsat_Axis_rep;
                end
```

```
       %% KL calculation and Saving - AVG

            cd(results)

BUCLE 7

            for j= NumBI+1:NumPerRep;
                sum1 = zeros(ysize,xsize);
                sum2 = zeros(ysize,xsize);
                for r = 1:length(Rep2Process);
                    i=Rep2Process(r);
                    sum1 = sum1+double(ItotalFINAL{i,j}(:,:));
                    sum2 = sum2+double(IflameFINAL{i,j}(:,:));
                end
                Itotalsum{j} = sum1;
                Itotalavg{j} = mask.*double(Itotalsum{j})./length(Rep2Process);%average
                                            total illumination with LED and flame

                Iflamesum{j} = sum2;
                Iflameavg{j} = double(Iflamesum{j})./length(Rep2Process);%average flame
                                                                          illumination
                mask2 = zeros(ysize,xsize);
                mask2 = Iflameavg{j}>60;
                Iflameavg{j} = Iflameavg{j}.*mask2;

                %calculate the KL of soot
                diff = double(Itotalavg{j})-double(Iflameavg{j});
                KL_avg{j}=log(complex(IbackavgFin)./diff);
                KLsat_avg{j}=log((IbackavgFin./diffSat).*mask);%saturated KL value
                Time(j)= 1000000/Framerate*j;%unit [us]
            end


            %% Plots for Average Results

            if strcmp(Image_plot_avg,'on')
                plot_ctrl_avg=dir(strcat(results,'\Avg_*.txt'));
                if ~isempty(plot_ctrl_avg)
%                     continue
                else
                    cd(results);

BUCLE 8

                    for j= NumBI+1:NumPerRep

                        %PLOTS
                        h1=figure(1);
                        KLrot{j}=imrotate(KL_avg{j},90);
                        imshow(KLrot{j}(((xsize-xinjector)-round(12*pixmm)):((xsize-
xinjector)+round(12*pixmm)),yinjector:round(70*pixmm)),[0,3.5]);
                        axis ([0 (70*pixmm-yinjector)  0 24*pixmm]);
                        axis on;
                        set(gca,'Xtick',[0:pixmm*10:(70*pixmm-yinjector)],'Ytick',
[0:pixmm*4: 24*pixmm],...
                            'YTickLabel',{'12' '8' '4' '0' '4' '8'
'12'},'XTickLabel',{'0' '10' '20' '30' '40' '50' '60' '70' '80'},...
                            'fontsize',22,'FontWeight','bold');
                        xlabel('Spray axis[mm]','Fontsize',22,'FontWeight','bold');
                        ylabel(' [mm]','Fontsize',22,'FontWeight','bold');
                        colormap(jet);
                        colorbar('location','EastOutside','Fontsize',14);

text(15,30,strcat(num2str(round(Time(j))),'μs'),'color','k','BackgroundColor','w','Fonts
ize',22,'Fontweight','b');
                        set(h1,'PaperPositionMode','auto');

name=strcat(results,'\',conditions,'_AVG_',num2str(round(Time(j))),'us.png');
                        print(h1,'-dpng',name);
                        close;
                        fclose all


h2=plot(KL_avg{j}(1:70*pixmm,xinjector),'Color',[0,1,0],'LineWidth',1.25);
                                                hold on
                                                h3 =

plot(KLsat_avg{j}(1:70*pixmm,xinjector),'Color',[1,0,0],'LineWidth',1.25);
                                                axis ([0 80*pixmm 0 6]);
```

```
set(gca,'Xtick',[0:10*pixmm:80*pixmm],'Ytick',[0:1:5],...
                                                    'XTickLabel',{'0' '10' '20' '30'
'40' '50' '60' '70' '80'},...
                                                    'YTickLabel',{'0' '1' '2' '3' '4'
'5' },'fontsize',16,'FontWeight','bold');
                                                xlabel('Spray axis
[mm]','Fontsize',20,'FontWeight','bold'); ylabel('KL [-
]','Fontsize',20,'FontWeight','bold');

title(strcat(num2str(round(Time(j))),'μs'),'Fontweight','b');

name=strcat(results,'\',conditions,'AVG_axis_',num2str(round(Time(j))),'us.png');

legend([h2,h3,],'KL','KLsat','Fontsize',12,'location','NorthWest','boxoff');
                                                saveas(h2,name,'png');
                                                close
                    end

                    fid=fopen('Avg_Fully_Plotted.txt', 'wt')
                    fprintf(fid, 'Average Results Fully Plotted');
                    fclose(fid);
                end
            end

            clearvars -except x t mainpath date pixmm NumBI FrameLess width FlameControl
LEDcontrol diffSat minimum_noise...
                Image_plot_repetition Image_plot_avg date_path cases
            fclose all;
            end
     end
    display(strcat('Case_',cases(t).name,'_processed'));

display('All Done');
```

Anexo B. Ejecución del código base

```
                150              % BUCLE 1
< 0.01        1 151              tic;
< 0.01        1 152              for n=1:5
  3.13        5 153                  data =GetMovie(0,n,'Photron','mraw',NumPerRep,Framerate);
< 0.01        5 154                  m=1;
< 0.01        5 155                  g=1;
< 0.01        5 156                  for c=1:NumBI
< 0.01       20 157                      if max(max(data(yinjector:yinjector+200,:,c)))>LEDcontrol
< 0.01       10 158                          Iback{n,m}=data(:,:,c);
< 0.01       10 159                          m=m+1;
< 0.01       10 160                      else
< 0.01       10 161                          Inoise{n,g}=data(:,:,c);
< 0.01       10 162                          g=g+1;
< 0.01       20 163                      end
< 0.01       20 164                  end
< 0.01        5 165                  if size(Iback,2)~=size(Inoise,2)
```

```
                223              % BUCLE 2
< 0.01        1 224               tic;
< 0.01        1 225              for i=1:size(Iback,2)*size(Iback,1)
< 0.01       10 226                  if ~isempty(Iback{i})
< 0.01       10 227                      Ibacksum=Ibacksum+double(Iback{i}(:,:)); %sum1=sum1+double(Iback1{i}(:,:));
< 0.01       10 228                      Iback_count=Iback_count+1;
< 0.01       10 229                  end
< 0.01       10 230              end
< 0.01        1 231              Ibackavg=Ibacksum./Iback_count;
< 0.01        1 232              Inoisesum = zeros(ysize,xsize);
< 0.01        1 233              Inoise_count= 0;
< 0.01        1 234             for i=1:size(Inoise,2)*size(Inoise,1)
< 0.01       10 235                  if ~isempty(Inoise{i})
< 0.01       10 236                      Inoisesum=Inoisesum+double(Inoise{i}(:,:)); %sum1=sum1+double(Iback1{i}(:,:));
< 0.01       10 237                      Inoise_count=Inoise_count+1;
< 0.01       10 238                  end
< 0.01       10 239             end
```

```
                250          % BUCLE 3
< 0.01        1 251          tic;
< 0.01        1 252           for r=1:length(Rep2Process)
< 0.01        5 253               n=Rep2Process(r);
  3.15        5 254               data = GetMovie(0,n,'Photron','mraw',NumPerRep,Framerate);
                255
< 0.01        5 256               for Nimg=1:NumPerRep
  0.11     1250 257                   if max(max(data(yinjector:yinjector+100,:,Nimg)))>LEDcontrol; %check if there is LED in the "f" Image
< 0.01      625 258                       LEDcount(n,Nimg)=1;
                259                       %                   BLKcount(n,Nimg)=0;
< 0.01      625 260                       TimeLED(n,Nimg)=1000000/Framerate*Nimg;
                261                       %             if Nimg>NumBI
  0.17      625 262                       Itotal{n,Nimg}=data(:,:,Nimg);
                263                       %             end
< 0.01      625 264                   else
                265                       %                   LEDcount(n,Nimg)=0;
< 0.01      625 266                       BLKcount(n,Nimg)=1;
< 0.01      625 267                       TimeBLK(n,Nimg)=1000000/Framerate*Nimg;
                268                       %             if Nimg>NumBI
  0.16      625 269                       Iflame{n,Nimg}=data(:,:,Nimg);
                270                       %             end
< 0.01     1250 271                   end
< 0.01     1250 272               end
< 0.01        5 273           end
```

```
                287             % BUCLE 4
< 0.01        1  288             tic;
< 0.01        1  289             for r=1:length(Rep2Process)
                290
< 0.01        5  291                 i=Rep2Process(r);
< 0.01        5  292                 c=1;
< 0.01        5  293                 d=1;
< 0.01        5  294                 for j=NumBI+1:NumPerRep
< 0.01     1230  295                     if LEDcount(i,j)==0
< 0.01      615  296                         if ~isempty(find(LEDcount(i,j+1:end)==1,1,'first'))
  0.27      612  297                             Itotalint(i,j)(:,:)=(Itotal(i,j-1)+Itotal(i,j+1))./2;
< 0.01      612  298                             TimeLEDint(i,j)=(TimeLED(i,j-1)+TimeLED(i,j+1))/2;
< 0.01      612  299                         end
< 0.01      615  300                     else if BLKcount(i,j)==0
< 0.01      615  301                             if ~isempty(find(BLKcount(i,j+1:end)==1,1,'first'))
  0.27      613  302                                 Iflameint(i,j)(:,:)=(Iflame(i,j-1)+Iflame(i,j+1))./2;
< 0.01      613  303                                    TimeBLKint(i,j)=(TimeBLK(i,j-1)+TimeBLK(i,j+1))/2;
< 0.01      613  304                             end
< 0.01      615  305                         end
< 0.01     1230  306                     end
< 0.01     1230  307                 end
< 0.01        5  308             end
```

```
                317             % BUCLE 5
< 0.01        1  318             tic;
< 0.01        1  319             for r=1:length(Rep2Process)
                320
< 0.01        5  321                 i=Rep2Process(r);
                322                 %     c=1;
                323                 %     d=1;
                324                 %     e=1;
                325                 %     f=1;
< 0.01        5  326                 for j=NumBI+1:NumPerRep
< 0.01     1230  327                     if LEDcount(i,j)==1
< 0.01      615  328                         ItotalFINAL(i,j)=Itotal(i,j);
< 0.01      615  329                         TimeLEDfinal(i,j)=TimeLED(i,j);
                330                         %             c=c+1;
< 0.01      615  331                         if j<=length(Iflameint(:,:))
< 0.01      613  332                             IflameFINAL(i,j)=Iflameint(i,j);
< 0.01      613  333                             TimeBLKfinal(i,j)=TimeBLKint(i,j);
                334                             %                 d=d+1;
< 0.01        2  335                         else
< 0.01        2  336                             IflameFINAL(i,j)=Iflame(i,j-1);
< 0.01        2  337                             TimeBLKfinal(i,j)=TimeBLK(i,j-1);
< 0.01      615  338                         end
< 0.01      615  339                     else
< 0.01      615  340                         IflameFINAL(i,j)=Iflame(i,j);
< 0.01      615  341                         TimeBLKfinal(i,j)=TimeBLK(i,j);
< 0.01      615  342                         if j<=length(Itotalint)
< 0.01      612  343                             ItotalFINAL(i,j)=Itotalint(i,j);
< 0.01      612  344                             TimeLEDfinal(i,j)=TimeLEDint(i,j);
< 0.01        3  345                         else
< 0.01        3  346                             ItotalFINAL(i,j)=Itotal(i,j-1);
< 0.01        3  347                             TimeLEDfinal(i,j)=TimeLED(i,j-1);
< 0.01      615  348                         end
< 0.01     1230  349                     end
```

```
                364             % BUCLE 6
< 0.01        1  365             tic;
                366             % // Este for estoy consiguiendo mejorarlo un 40 %
< 0.01        1  367             for r=1:length(Rep2Process)
< 0.01        5  368                 i=Rep2Process(r);
< 0.01        5  369                 for j=NumBI+1:NumPerRep
                370                     %             Flamepos=find(TimeBLK(i,:)==TimeBLKord(i,j));
                371                     %             mask2=zeros(ysize,xsize);
                372                     %             mask2=Iflame1(i,Flamepos)>60;
                373
                374                     %             diff=(double(Itotalint(i,j))*mask)-(double(Iflame1(i,Flamepos)*mask2));
  0.21     1230  375                     a=double(ItotalFINAL(i,j));
  0.25     1230  376                     b=double(IflameFINAL(i,j));
                377 %                   c=double(b);
  0.75     1230  378                     diff=double(a-b);
  4.99     1230  379                     KL_rep(j)=log(complex((IbackavgFin./diff)));
  0.19     1230  380                     KL_Axis_rep(j)=KL_rep(j)(1:70*pixmm,xinjector);
  3.56     1230  381                     KLsat_rep(j)=log((IbackavgFin./diffSat).*mask); %saturated KL value
  0.16     1230  382                     KLsat_Axis_rep(j)=KLsat_rep(j)(1:70*pixmm,xinjector);
                383
                384
< 0.01     1230  385                 end
                386 %                 save(strcat('KL_LEI_Rep_',num2str(i),'.mat'),'KL_rep', 'KLsat_rep', 'KL_Axis_rep', 'KLsat_Axis_rep', '-v7.3');
  0.57        5  387                 clearvars KL_rep KLsat_rep KL_Axis_rep KLsat_Axis_rep;
< 0.01        5  388             end
```

```
                399            % BUCLE 7
< 0.01        1 400             tic;
                401             % // Consigo mejorar el for entre un 18 y 27 % aprox.
< 0.01        1 402             for j= NumBI+1:NumPerRep;
< 0.01      246 403                 sum1 = zeros(ysize,xsize);
< 0.01      246 404                 sum2 = zeros(ysize,xsize);
< 0.01      246 405                 for r = 1:length(Rep2Process);
< 0.01     1230 406                     i=Rep2Process(r);
  0.52     1230 407                     sum1 = sum1+double(ItotalFINAL(i,j)(:,:));
  0.50     1230 408                     sum2 = sum2+double(IflameFINAL(i,j)(:,:));
< 0.01     1230 409                 end
< 0.01      246 410                 Itotalsum(j) = sum1;
  0.18      246 411                 Itotalavg(j) = mask.*double(Itotalsum(j))./length(Rep2Process);%average total illumination with LED and flame
< 0.01      246 412                 Iflamesum(j) = sum2;
  0.17      246 413                 Iflameavg(j) = double(Iflamesum(j))./length(Rep2Process);%average flame illumination
  0.04      246 414                 mask2 = zeros(ysize,xsize);
  0.04      246 415                 mask2 = Iflameavg(j)>60;
  0.14      246 416                 Iflameavg(j) = Iflameavg(j).*mask2;
                417
                418                 %calculate the KL of soot
  0.10      246 419                 diff = double(Itotalavg(j))-double(Iflameavg(j));
  0.91      246 420                 KL_avg(j)=log(complex(IbackavgFin)./diff);
  0.70      246 421                 KLsat_avg(j)=log((IbackavgFin./diffSat).*mask);%saturated KL value
< 0.01      246 422                 Time(j)= 1000000/Framerate*j;%unit [us]
< 0.01      246 423             end
```

```
                526            % BUCLE 8
< 0.01        1 527                 tic;
< 0.01        1 528                 for j= NumBI+1:NumPerRep
                529
                530                     %PLOTS
  0.19      246 531                     h1=figure(1);
  1.75      246 532                     KLrot(j)=imrotate(KL_avg(j),90);
 13.38      246 533                     imshow(KLrot(j)(((xsize-xinjector)-round(12*pixmm)):((xsize-xinjector)+round(12*pixmm)),yinjector:round(70*pixmm)
                534 %                     axis ([0 (70*pixmm-yinjector)  0 24*pixmm]);
                535 %                     axis on;
                536 %                     set(gca,'Xtick',[0:pixmm*10:(70*pixmm-yinjector)],'Ytick', [0:pixmm*4: 24*pixmm],...
                537 %                         'YTickLabel',{'12' '8' '4' '0' '4' '8' '12'},'XTickLabel',{'0' '10' '20' '30' '40' '50' '60' '70' '80'},...
                538 %                         'fontsize',22,'FontWeight','bold');
                539 %                     xlabel('Spray axis[mm]','Fontsize',22,'FontWeight','bold');
                540 %                     ylabel(' [mm]','Fontsize',22,'FontWeight','bold');
                541 %                     colormap(jet);
                542 %                     colorbar('location','EastOutside','Fontsize',14);
                543 %                     text(15,30,strcat(num2str(round(Time(j))),'µs'),'color','k','BackgroundColor','w','Fontsize',22,'Fontweight','b
                544 %                     set(h1,'PaperPositionMode','auto');
                545 %                     name=strcat(results,'\',conditions,'_AVG_',num2str(round(Time(j))),'us.png');
                546 %                     print(h1,'-dpng',name);
                547 %                     close;
                548 %                     fclose all
                549
```