Modelos de lenguaje sostenibles mediante optimización evolutiva del motor de inferencia

Javier Jareño[1][0000−0002−1735−0520], José Miguel Aragón-Jurado[2][0000−0002−9835−8793], Juan Carlos de la Torre[1][0000−0002−3662−6422], Patricia Ruiz[1][0000−0001−5891−0646], and Bernabé Dorronsoro[1][0000−0003−0481−790X]

[1] Escuela Superior de Ingeniería, Universidad de Cádiz, Cádiz, España
{javier.jareno, juan.detorre, patricia.ruiz, bernabe.dorronsoro}@uca.es
[2] Graduate School of Information, Production and Systems, Waseda University, Kitakyushu, Japón
josemi.aragon@aoni.waseda.jp

Resumen Los modelos de lenguaje de gran tamaño (*Large Language Models*, LLM) se han convertido en herramientas habituales en educación, investigación, programación y generación de contenido, pero su despliegue masivo incrementa de forma notable el coste energético asociado a la inferencia. Este trabajo aborda este problema desde la pila de software, proponiendo una metodología automática para optimizar motores de inferencia de LLM mediante transformaciones de compilador. La búsqueda se formula como una instancia del *Green Software Code Optimization Problem* (gSCOP) y se resuelve mediante un algoritmo genético celular con soluciones de longitud variable, un operador de cruce basado en la subsecuencia común más larga y una mutación contextual guiada por densidad. La propuesta se evalúa sobre `llama.cpp` en un Mac Mini con Apple Silicon M4, empleando modelos Llama 3.2 de 1 y 3 mil millones de parámetros cuantizados a 4 bits. Los resultados muestran reducciones del 12.16% en energía y del 15.96% en tiempo frente a la versión sin optimizar para el modelo de 1B, así como mejoras energéticas del 5.3% frente a `-O0` y del 2.7% frente a `-O3` para el modelo de 3B. Además, la secuencia optimizada para `Q4_0` generaliza a `Q8_0`, manteniendo ahorros cercanos al 12%. Estos resultados confirman que las optimizaciones conscientes del hardware pueden reducir el consumo energético sin modificar el modelo neuronal.

Palabras clave Large Language Models · IA sostenible · Eficiencia energética · Optimización de compiladores · Llama 3.2 · Cuantización · Algoritmos genéticos

1 Introducción

Los modelos de lenguaje de gran tamaño (*Large Language Models*, LLM) se han consolidado como una de las tecnologías más influyentes de la inteligencia artificial reciente, con aplicaciones en educación, investigación, programación,

asistencia documental y generación de contenido [16]. Sin embargo, su adopción masiva plantea un reto fundamental: el elevado coste computacional y energético de la inferencia. A diferencia del entrenamiento, que se realiza de forma puntual, la inferencia se ejecuta de manera continua y a gran escala. De hecho, se ha estimado que la generación de texto en modelos LLM puede requerir varios julios por token, por lo que millones de consultas diarias pueden traducirse en un consumo acumulado muy elevado [21]. Mejorar la eficiencia energética de esta fase es, por tanto, una cuestión central para avanzar hacia una inteligencia artificial más sostenible.

Una parte importante de esta eficiencia depende del motor de inferencia y de su adaptación al hardware objetivo. En este contexto, `llama.cpp` [11] se ha convertido en una plataforma de referencia para ejecutar LLM en dispositivos heterogéneos, incluyendo CPU, GPU y arquitecturas de bajo consumo. Su implementación en C/C++ y su soporte para distintos niveles de cuantización lo convierten en un candidato especialmente adecuado para estudiar optimizaciones a nivel de software sin modificar el modelo neuronal.

Además de las optimizaciones propias del motor, la infraestructura de compilación influye de forma directa en el rendimiento final del programa. LLVM [13] permite aplicar transformaciones de código, denominadas *passes*, sobre una representación intermedia del software. No obstante, niveles genéricos como `-O1`, `-O2` o `-O3` no están diseñados para una aplicación y una arquitectura concretas, ni optimizan explícitamente el consumo energético. Además, reducir el tiempo de ejecución no implica necesariamente reducir la energía consumida [4].

Este trabajo aborda la optimización energética de la inferencia de LLM desde la pila de software. Para ello, se formula la selección y ordenación de transformaciones LLVM como una instancia del *Green Software Code Optimization Problem* (gSCOP) [5], resuelta sobre `llama.cpp` mediante un algoritmo genético celular [2]. La propuesta busca secuencias de *passes* adaptadas a una pareja software–hardware concreta, con el objetivo de reducir el consumo energético sin alterar la arquitectura del modelo, sus pesos ni su cuantización.

Las principales contribuciones son: (i) una metodología automática para optimizar energéticamente motores de inferencia de LLM mediante LLVM; (ii) un análisis del efecto de los niveles genéricos de optimización sobre tiempo y energía en `llama.cpp`; (iii) una representación de longitud variable para gSCOP con operadores evolutivos específicos para secuencias de *passes*; y (iv) una evaluación con modelos Llama 3.2 de 1 y 3 mil millones de parámetros, incluyendo el estudio de generalización entre niveles de cuantización [10].

2 Estado del arte

La eficiencia energética de los LLM se ha abordado principalmente desde tres niveles complementarios: el modelo, la arquitectura de red y la plataforma de ejecución. Las técnicas de compresión reducen el tamaño o la complejidad del modelo para disminuir el coste de inferencia [9]. Entre ellas destacan la cuantización, que reduce la precisión numérica de pesos y activaciones; la poda, que

elimina parámetros o estructuras poco relevantes; y la destilación, que entrena un modelo más pequeño para reproducir el comportamiento de otro mayor. Aunque estas técnicas son efectivas, suelen modificar el modelo original y pueden requerir ajuste fino o reentrenamiento para recuperar precisión.

Una segunda línea consiste en diseñar arquitecturas intrínsecamente más eficientes. Los transformadores eficientes reducen el coste de la atención mediante patrones dispersos, aproximaciones de bajo rango, compartición de parámetros o activación parcial del modelo [19]. Estas estrategias pueden disminuir el número de operaciones por token y mejorar el rendimiento por vatio, pero dependen de decisiones tomadas durante el diseño o entrenamiento del modelo. Por ello, no siempre son aplicables a modelos ya desplegados ni a motores de inferencia existentes.

La tercera línea se centra en el hardware y en el software de ejecución. Los aceleradores especializados explotan paralelismo, baja precisión y dispersión para mejorar la eficiencia de las operaciones dominantes en transformadores [12]. En paralelo, compiladores y *frameworks* de despliegue como TVM y TensorRT optimizan el grafo computacional mediante fusión de operadores, selección de *kernels* y reorganización de memoria [8,15]. Sin embargo, estas herramientas suelen operar a un nivel más alto y delegan la generación final de código máquina en compiladores de propósito general.

La optimización de bajo nivel mediante compiladores plantea un problema diferente. La literatura sobre *phase ordering* ha demostrado que el orden de aplicación de las transformaciones puede afectar significativamente al rendimiento del software [7]. Asimismo, trabajos previos han formulado la búsqueda de secuencias de *passes* como un problema de optimización para mejorar tiempo de ejecución o consumo energético [6,5]. No obstante, la inferencia autoregresiva de LLM presenta características particulares, como dependencia del ancho de banda de memoria, reutilización intensiva de estructuras cuantizadas y flujos de control menos regulares que los de muchas cargas clásicas de computación de altas prestaciones [3].

En consecuencia, existe una brecha clara: la mayoría de propuestas reducen el coste de inferencia modificando el modelo, diseñando hardware específico o aplicando optimizaciones a nivel de grafo, mientras que apenas se ha explorado la optimización energética directa del código del motor de inferencia. Este trabajo se sitúa en ese espacio y propone una metodología complementaria a la compresión y al diseño arquitectónico. Al actuar sobre la representación intermedia LLVM de `llama.cpp`, permite obtener versiones del motor adaptadas al hardware objetivo, reduciendo energía sin modificar el LLM ni requerir reentrenamiento.

3 Metodología

El objetivo de este trabajo es reducir el consumo energético de un motor de inferencia de LLM sin modificar el modelo neuronal, sus pesos ni su cuantización. Para ello, se aborda la optimización del código de `llama.cpp` como un problema

de selección y ordenación de transformaciones de compilación. En particular, se resuelve el gSCOP [5], que consiste en encontrar una secuencia de transformaciones de código capaz de generar una versión semánticamente equivalente del programa con menor consumo energético.

Sea P el programa original y sea $T = t_1, \ldots, t_k$ el conjunto de transformaciones LLVM disponibles. Una solución del problema se define como una secuencia ordenada $\mathbf{S} = [s_1, \ldots, s_N]$, donde cada $s_i \in T$. Aplicar dicha secuencia sobre P produce una nueva versión P' del programa. El problema se formula como:

$$\min_{\mathbf{S}} F(P, \mathbf{S}) = \text{Energy}(P'). \tag{1}$$

donde P' es el resultado de aplicar las transformaciones de $\mathbf{S}$ en el orden especificado. La repetición de transformaciones está permitida, ya que aplicar un mismo *pass* en diferentes puntos de la secuencia puede producir efectos distintos sobre la representación intermedia del programa [20]. De este modo, el espacio de búsqueda no solo depende de qué transformaciones se seleccionan, sino también de su orden y frecuencia.

La metodología propuesta actúa sobre la representación intermedia de LLVM. Primero, el proyecto `llama.cpp` se compila utilizando `clang` con las opciones `-disable-llvm-passes` y `-disable-llvm-optzns`, evitando así la aplicación de optimizaciones no controladas. A continuación, se emplea Whole Program LLVM (WLLVM) [17] para extraer un único fichero de representación intermedia LLVM, denotado como P_{IR}, que contiene el código completo del motor de inferencia. Esta extracción se realiza una sola vez al inicio del proceso. Posteriormente, cada solución candidata $\mathbf{S}$ se aplica sobre P_{IR} mediante `opt`, generando una versión transformada que se recompila con `clang` para obtener el ejecutable final. La calidad de la solución se evalúa midiendo la energía consumida durante la inferencia con el ejecutable resultante.

La búsqueda de secuencias de transformaciones se realiza mediante un algoritmo genético celular (*cellular Genetic Algorithm*, cGA) [2,1]. A diferencia de un algoritmo genético panmíctico, el cGA distribuye la población sobre una malla toroidal, de forma que cada individuo solo interactúa con sus vecinos. Esta estructura espacial favorece la diversidad y reduce el riesgo de convergencia prematura, un aspecto especialmente relevante en un problema combinatorio de gran dimensión como gSCOP.

Cada individuo codifica una secuencia de *passes* LLVM de longitud variable. Esta decisión evita fijar a priori el número de transformaciones, permitiendo que el algoritmo descubra soluciones compactas o extensas según las necesidades del problema. La recombinación se realiza mediante un nuevo operador basado en la subsecuencia común más larga (*Longest Common Subsequence*, LCS), que preserva los fragmentos compartidos por dos soluciones y recombina aleatoriamente los segmentos restantes. Así, se explotan patrones de transformaciones prometedores sin impedir la generación de secuencias con nuevas longitudes.

La mutación se guía mediante un nuevo operador contextual por densidad. Este operador puede eliminar, insertar o modificar genes con probabilidades p_r, p_a y p_m, respectivamente. Para las operaciones de inserción y modificación se

utiliza una matriz de transición $\mathcal{M}$, donde $\mathcal{M}_{i,j}$ representa la frecuencia con la que el *pass* t_i aparece seguido de t_j en soluciones de alta calidad. Dado un contexto formado por un predecesor p y un sucesor s, la probabilidad de seleccionar un nuevo *pass* se obtiene mediante:

$$\mathbf{v} = \mathcal{M}_{p,\cdot} + (\mathcal{M}_{\cdot,s})^{\mathsf{T}} + \mathbf{1}. \tag{2}$$

De esta forma, se favorecen transformaciones que encajan bien con el contexto local de la secuencia, en lugar de introducir cambios completamente aleatorios. Si una secuencia candidata provoca un error de compilación, viola una restricción interna de LLVM o produce un fallo durante la ejecución de `llama-bench`, se le asigna una aptitud penalizada. Además, las transformaciones consideradas se restringen a *passes* estándar de LLVM que preservan la semántica del programa.

La experimentación se llevó a cabo con LLVM 16.0.6 utilizando el gestor de *passes* heredado. El cGA se ejecutó 30 veces de forma independiente para cada experimento. La población estuvo formada por 100 individuos, con selección por torneo binario en vecindario $L5$, recombinación LCS con probabilidad $p_c = 0.99$ y mutación contextual con $p_r = 1.5/n$, $p_a = 1/n$ y $p_m = 3/n$, siendo n la longitud de la secuencia. La longitud de los cromosomas se mantuvo dentro del intervalo $[50, 200]$. El criterio de parada fue alcanzar 10 000 evaluaciones o no observar mejora durante las últimas 10 generaciones. La política de reemplazo fue elitista, permitiendo reemplazar un individuo si la nueva solución era mejor o se encontraba dentro de un margen del 1%.

Como plataforma hardware se utilizó un Mac Mini con Apple Silicon M4, equipado con una CPU de 10 núcleos, formada por 4 núcleos de rendimiento y 6 de eficiencia, una GPU integrada de 10 núcleos y 16 GB de memoria RAM. El sistema operativo empleado fue macOS Sequoia 15.1. El consumo energético se midió mediante `powermetrics` [14], accediendo a los sensores hardware del SoC (*System-on-Chip*, SoC) con un intervalo de muestreo de 100 ms. La energía total se obtuvo integrando numéricamente las muestras de potencia de los dominios CPU y GPU durante el tiempo de ejecución. El equipo se mantuvo conectado a la red eléctrica mediante regulador de tensión para minimizar fluctuaciones externas.

Los modelos evaluados pertenecen a la familia Llama 3.2 [10]. En concreto, se utilizaron Llama 3.2 1B Instruct cuantizado a 4 bits (`Q4_0`), Llama 3.2 3B Instruct cuantizado a 4 bits y Llama 3.2 1B Instruct cuantizado a 8 bits (`Q8_0`). Los dos modelos `Q4_0` se emplearon para optimizar el motor de inferencia en dos escalas distintas, mientras que el modelo `Q8_0` permitió estudiar si la secuencia optimizada para `Q4_0` generaliza a otro nivel de cuantización.

La evaluación se realizó con `llama-bench`, configurado con `-p 128 -n 128`, de forma que cada ejecución procesa 128 tokens de entrada y genera 128 tokens de salida. Esta configuración representa una carga de inferencia completa, al incluir tanto procesamiento de *prompt* como generación autoregresiva, manteniendo al mismo tiempo un coste asumible para el proceso evolutivo. Para reducir el efecto del ruido experimental, cada solución candidata se ejecutó cinco veces y se tomó como valor de aptitud el peor consumo observado, siguiendo

la estrategia *wcase* empleada en gSCOP [5]. Posteriormente, la mejor solución obtenida en las 30 ejecuciones independientes se evaluó otras 30 veces y se comparó con las versiones compiladas mediante los niveles estándar `-O0`, `-O1`, `-O2`, `-O3`, `-Os` y `-Oz`, incluyendo también configuraciones con `-flto` cuando corresponde. La significancia estadística de las comparaciones se analizó mediante el test no paramétrico Scott-Knott ESD con un nivel de confianza del 95% [18]. Finalmente, para reducir contaminación entre ejecuciones, se controló el estado de memoria virtual y caché del sistema, reiniciando el entorno experimental entre generaciones del algoritmo.

4 Resultados

Esta sección presenta los resultados obtenidos al aplicar la metodología propuesta sobre `llama.cpp`. En primer lugar, se analiza el impacto del número de tokens sobre el tiempo y la energía de inferencia, con el objetivo de seleccionar una carga de trabajo adecuada para la evaluación. Posteriormente, se comparan las versiones optimizadas mediante cGA con los niveles estándar de optimización de LLVM. Finalmente, se estudia la capacidad de generalización de la secuencia optimizada entre distintos niveles de cuantización.

El análisis preliminar del número de tokens, del que la Fig. 1 muestra la fase de generación por ser la más costosa, confirma que la generación autorregresiva es considerablemente más costosa que el procesamiento del *prompt*. Para el mayor tamaño evaluado, la generación requirió aproximadamente un 605.57% más de energía y un 748.08% más de tiempo que el procesamiento de tokens. Además, el efecto de los niveles genéricos de optimización de LLVM fue más visible conforme aumentaba el tamaño de la entrada. En particular, para procesamiento de tokens, `-O1` redujo el tiempo hasta un 10.56%, mientras que `-O3` redujo la energía hasta un 9.61%. En generación, sin embargo, las mejoras fueron más moderadas, con reducciones del 1.78% en tiempo y del 2.45% en energía. A partir de estos resultados, se seleccionó la configuración `-p 128 -n 128`, que combina procesamiento de *prompt* y generación de respuesta manteniendo un coste asumible para el algoritmo evolutivo.

La Fig. 2 muestra los resultados para Llama 3.2 1B cuantizado a 4 bits. La solución obtenida mediante cGA redujo el consumo energético total del SoC en un 12.16% respecto a la versión sin optimizar, y en un 1.96% respecto a `-O3`. En tiempo de ejecución, la mejora frente a `-O0` fue del 15.96%, alcanzando un rendimiento comparable al de `-O3`. Al incorporar optimización en tiempo de enlace mediante `-flto`, tanto la solución cGA como `-O3` alcanzaron los mejores rangos estadísticos, situándose alrededor de 40 J de consumo total.

El desglose de potencia de CPU y GPU, mostrado en la Fig. 3, revela que las mejoras no se deben únicamente a una reducción del tiempo de ejecución. Mientras que `-O3` tiende a reducir principalmente la potencia de CPU, la solución encontrada por el cGA disminuye de forma más clara la potencia media de GPU, con una reducción cercana al 10% respecto a `-O3`. Este resultado es especialmente relevante en Apple Silicon, donde CPU y GPU comparten memoria unificada y

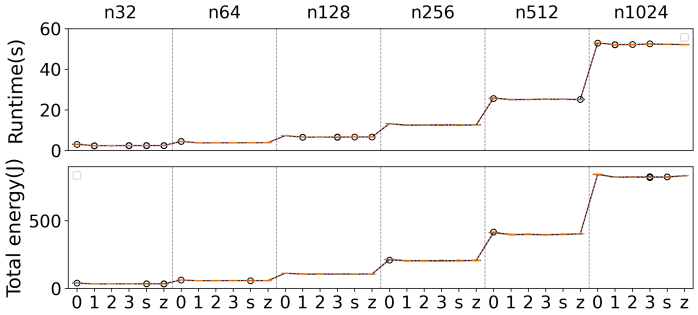
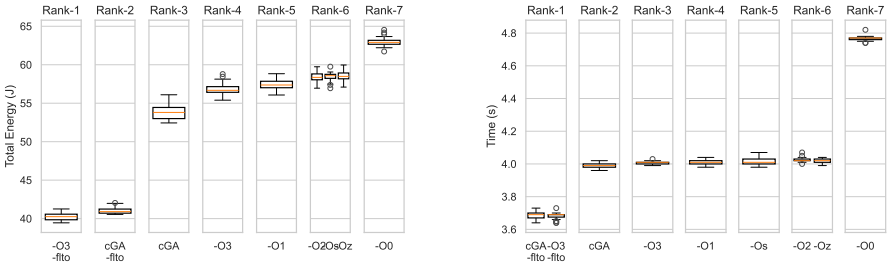


Fig. 1. Tiempo de ejecución y consumo energético de Llama 3.2 1B `Q4_0` en `llama.cpp` para distintos tamaños de generación de tokens, considerando los niveles genéricos de optimización de LLVM.



(a) Consumo energético total del SoC.

(b) Tiempo de ejecución.

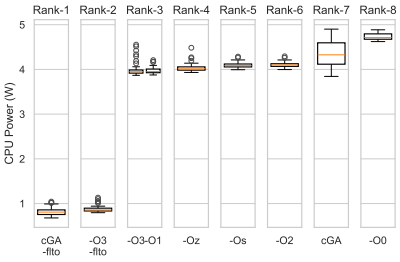
Fig. 2. Eficiencia energética y rendimiento temporal de Llama 3.2 1B `Q4_0` para la versión optimizada mediante cGA y los niveles genéricos de optimización del compilador. Las cajas representan los percentiles 25–75 y los bigotes $1.5 \cdot \text{IQR}$.

la eficiencia del flujo de datos puede influir de forma directa en el consumo global del SoC.

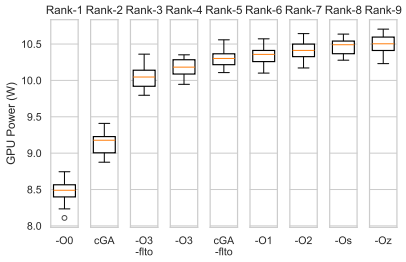
Para evaluar la escalabilidad de la propuesta, el proceso de optimización se repitió con Llama 3.2 3B cuantizado a 4 bits. Como se observa en la Fig. 4, la solución cGA obtuvo una reducción energética del 5.3% frente a `-O0` y del 2.7% frente a `-O3`, alcanzando además un rango estadístico superior al de los niveles genéricos de optimización. En tiempo de ejecución, la mejora frente a `-O0` fue también del 5.3%, manteniendo un rendimiento competitivo respecto a `-O3`.

El análisis de potencia de la Fig. 5 confirma de nuevo que la reducción energética está asociada, en gran medida, a una menor potencia media de GPU. Este comportamiento sugiere que la búsqueda evolutiva identifica transformaciones que no solo aceleran la ejecución, sino que también modifican el patrón de utilización de los recursos del SoC.

La capacidad de generalización entre niveles de cuantización se evaluó aplicando la secuencia optimizada para Llama 3.2 1B `Q4_0` sobre Llama 3.2 1B `Q8_0`,

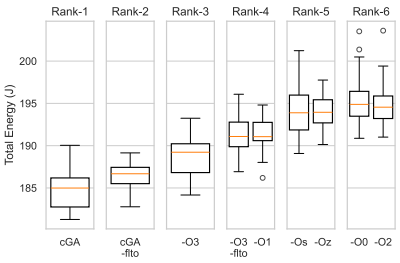


(a) Potencia media de CPU.

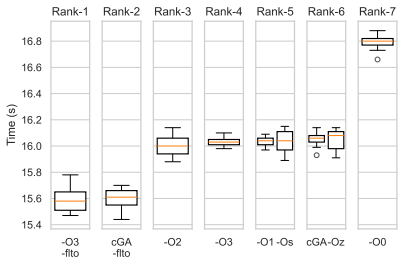


(b) Potencia media de GPU.

Fig. 3. Potencia media de CPU y GPU de Llama 3.2 1B `Q4_0` para la versión optimizada mediante cGA y los niveles genéricos de optimización del compilador. Las cajas representan los percentiles 25–75 y los bigotes $1.5 \cdot \text{IQR}$.



(a) Consumo energético total del SoC.



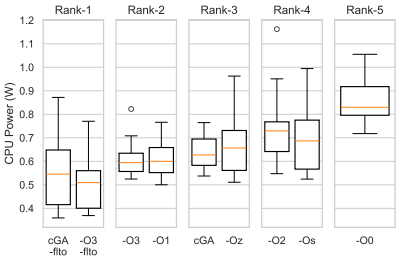
(b) Tiempo de ejecución.

Fig. 4. Eficiencia energética y rendimiento temporal de Llama 3.2 3B `Q4_0` para la versión optimizada mediante cGA y los niveles genéricos de optimización del compilador. Las cajas representan los percentiles 25–75 y los bigotes $1.5 \cdot \text{IQR}$.

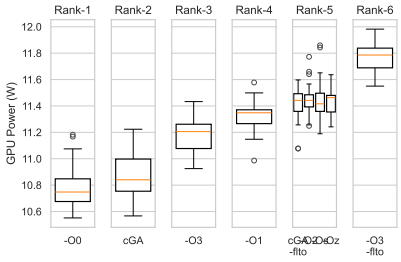
sin modificar el motor de inferencia. Los resultados de la Fig. 6 muestran que la solución cGA alcanzó una mediana de aproximadamente 64.24 J, lo que supone una reducción del 11.77% frente a `-O0` y del 6.03% frente a `-O3`. Además, el tiempo de ejecución se redujo un 12.27% respecto a `-O0`, obteniendo en este caso una ventaja estadísticamente significativa frente a `-O3`. Estos resultados indican que la optimización aprendida para un modelo cuantizado a 4 bits conserva su utilidad en una versión de mayor precisión.

5 Discusión

Los resultados obtenidos muestran que los niveles estándar de optimización de compilador no constituyen necesariamente el límite superior de eficiencia energética para la inferencia de LLM. Aunque `-O3` es una configuración agresiva y ampliamente utilizada, su objetivo principal es mejorar el rendimiento medio

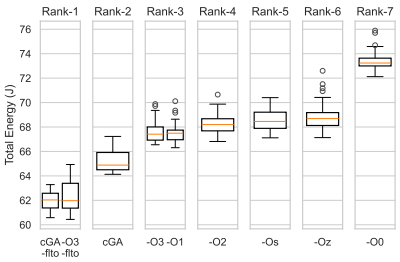


(a) Potencia media de CPU.

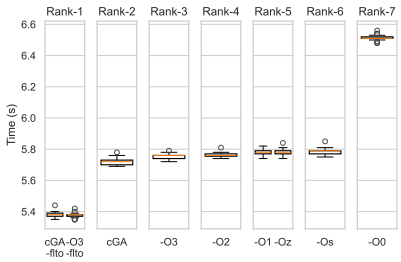


(b) Potencia media de GPU.

Fig. 5. Potencia media de CPU y GPU de Llama 3.2 3B `Q4_0` para la versión optimizada mediante cGA y los niveles genéricos de optimización del compilador. Las cajas representan los percentiles 25–75 y los bigotes $1.5 \cdot \mathrm{IQR}$.



(a) Consumo energético total del SoC.



(b) Tiempo de ejecución.

Fig. 6. Eficiencia energética y rendimiento temporal de Llama 3.2 1B `Q8_0` al aplicar la secuencia optimizada para Llama 3.2 1B `Q4_0`. Las cajas representan los percentiles 25–75 y los bigotes $1.5 \cdot \mathrm{IQR}$.

de un amplio conjunto de programas, no minimizar energía para una pareja software–hardware concreta. En cambio, la metodología propuesta adapta la secuencia de *passes* al motor de inferencia, al modelo y a la arquitectura objetivo, permitiendo encontrar configuraciones más adecuadas para `llama.cpp` sobre Apple Silicon.

Un aspecto relevante es que la reducción energética no depende únicamente de disminuir el tiempo de ejecución. En varios casos, la solución cGA alcanza tiempos comparables a `-O3`, pero con menor consumo total. El análisis de potencia indica que parte de esta mejora procede de una utilización más eficiente de la GPU integrada, especialmente en los modelos cuantizados. Este resultado refuerza la necesidad de optimizar energía de forma explícita, en lugar de asumir que una ejecución más rápida será siempre más eficiente.

La propuesta también resulta complementaria a las técnicas habituales de eficiencia en LLM. Métodos como cuantización, poda, destilación o arquitecturas

eficientes actúan sobre el modelo y pueden requerir reentrenamiento, ajuste fino o cambios en la precisión. Por el contrario, este trabajo modifica únicamente el motor de inferencia, manteniendo intactos la arquitectura, los pesos y el nivel de cuantización del LLM. Esta característica permite reducir el consumo sin introducir riesgos directos sobre la calidad de las respuestas generadas.

El análisis de las secuencias encontradas sugiere que la inferencia autoregresiva de LLM requiere estrategias distintas a las optimizaciones genéricas de compilador. La secuencia evolucionada prioriza transformaciones relacionadas con el flujo de control y el acceso a memoria, evitando una aplicación excesiva de transformaciones escalares generales. Además, la mejora observada al aplicar la secuencia optimizada para `Q4_0` sobre `Q8_0` indica que la optimización captura propiedades del motor de inferencia y de la plataforma, y no solo un comportamiento accidental de una configuración concreta. No obstante, las secuencias resultantes deben interpretarse como soluciones especializadas para cada pareja software–hardware, no como secuencias universales.

Finalmente, el coste energético inicial de la búsqueda debe considerarse dentro del balance ambiental de la propuesta. Aunque la optimización evolutiva requiere compilar y evaluar miles de variantes, dicho coste se amortiza cuando la versión optimizada se despliega a escala. En la experimentación realizada, el coste acumulado fue de aproximadamente 17.41 MJ (4.84 kWh), amortizable en menos de 2.9 millones de consultas de unos 300 tokens. Por tanto, en escenarios con un elevado número de inferencias o con despliegues distribuidos, la optimización automática del software de inferencia constituye una vía práctica para reducir el consumo acumulado de los LLM.

6 Conclusiones y trabajo futuro

Este trabajo ha presentado una metodología automática para reducir el consumo energético de la inferencia de modelos de lenguaje de gran tamaño mediante optimizaciones de compilador. Para ello, se ha formulado la selección y ordenación de transformaciones LLVM como una instancia del gSCOP, resuelta mediante un algoritmo genético celular con representación de longitud variable y operadores específicos para secuencias de *passes*. A diferencia de las aproximaciones basadas en modificar el modelo, la propuesta actúa directamente sobre el motor de inferencia y mantiene inalterados la arquitectura, los pesos y la cuantización del LLM.

La metodología se ha evaluado sobre `llama.cpp` en una plataforma Apple Silicon M4, utilizando modelos Llama 3.2 de 1 y 3 mil millones de parámetros cuantizados a 4 bits. Los resultados muestran que las secuencias encontradas por el cGA reducen tanto el consumo energético como el tiempo de ejecución frente a la versión sin optimizar, manteniendo además un comportamiento competitivo frente a los niveles estándar de LLVM. En el modelo de 1B parámetros se alcanzaron reducciones del 12.16% en energía y del 15.96% en tiempo frente a `-O0`; en el modelo de 3B parámetros, la mejora energética fue del 5.3% frente a `-O0` y del 2.7% frente a `-O3`. Estos resultados confirman que las optimizaciones genéri-

cas de compilador no representan necesariamente el límite superior de eficiencia energética para la inferencia de LLM.

Además, la secuencia optimizada para Llama 3.2 1B `Q4_0` generalizó de forma efectiva a la variante `Q8_0`, con una reducción energética del 11.77% frente a `-O0` y del 6.03% frente a `-O3`. Este resultado sugiere que la propuesta captura propiedades relevantes del motor de inferencia y de la plataforma hardware, aunque las secuencias obtenidas siguen siendo específicas de cada configuración. El análisis de los *passes* seleccionados muestra, además, una estrategia distinta a `-O3`, más orientada a simplificar el flujo de control y mejorar los accesos a memoria que a aplicar de forma intensiva transformaciones escalares genéricas.

Como trabajo futuro, se plantea extender la metodología a modelos de mayor tamaño y a plataformas hardware más heterogéneas, incluyendo GPUs discretas y entornos de altas prestaciones. También resulta de interés estudiar formulaciones multiobjetivo que optimicen simultáneamente energía, latencia y uso de memoria, así como mecanismos de transferencia que permitan reutilizar conocimiento entre modelos, cuantizaciones y arquitecturas hardware.

Agradecimientos Esta publicación forma parte del proyecto eMob (PID2022-137858 OB-I00), financiado por MICIU/AEI/10.13039/501100011033 y por FEDER, "Una manera de hacer Europa", así como de los proyectos GRASS (FEDER-UCA-2024-A2-23), financiado por el Programa Operativo FEDER Andalucía 2021-2027 y la Consejería de Universidad, Investigación e Innovación de la Junta de Andalucía, y gCODE (PIDI_2024_01307), financiado por los Fondos Europeos de Desarrollo Regional (FEDER) y la Consejería de Universidad, Investigación e Innovación de la Junta de Andalucía. Javier Jareño agradece al Ministerio de Ciencia, Innovación y Universidades el apoyo recibido mediante la ayuda FPU24/04054. J.M. Aragón-Jurado agradece al Ministerio de Ciencia, Innovación y Universidades el apoyo recibido mediante la ayuda FPU21/02026. Juan Carlos de la Torre agradece a la Universidad de Cádiz el apoyo recibido mediante el proyecto "Técnicas avanzadas de Inteligencia Artificial aplicadas al Tratamiento de Imágenes en entornos reales" (I+i 2024/03).

Referencias

1. Alba, E., Dorronsoro, B.: The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. IEEE Transactions on Evolutionary Computation **9**(2), 126–142 (2005)
2. Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms. Springer Publishing Company, 1st edn. (2008)
3. Aminabadi, R.Y., Rajbhandari, S., Awan, A.A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al.: Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15. IEEE (2022)
4. Aragón-Jurado, J.M., Bangash, A.A., Dorronsoro, B., Ali, K., Hindle, A., Ruiz, P.: Does faster mean greener? runtime and energy trade-offs in ios applications with compiler optimizations. Sustainable Computing: Informatics and Systems p. 101166 (2025)

5. Aragón-Jurado, J.M., de la Torre, J.C., Ruiz, P., Dorronsoro, B.: Automatic software tailoring for Green Internet of Things. Internet of Things **30**, 101521 (2025). https://doi.org/10.1016/j.iot.2025.101521
6. Aragón-Jurado, J.M., de la Torre, J.C., Ruiz, P., Galindo, P.L., Zomaya, A.Y., Dorronsoro, B.: Automatic Software Tailoring for Optimal Performance. IEEE Transactions on Sustainable Computing **9**(3), 464–481 (2024). https://doi.org/10.1109/TSUSC.2023.3330671
7. Ashouri, A., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J., Bignoli, A., Palermo, G., Silvano, C.: MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Transactions on Architecture and Code Optimization **14**(29) (2017)
8. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al.: {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 578–594 (2018)
9. Chitty-Venkata, K.T., Mittal, S., Emani, M., Vishwanath, V., Somani, A.K.: A survey of techniques for optimizing transformer inference. Journal of Systems Architecture **144**, 102990 (2023)
10. Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al.: The Llama 3 Herd of Models. arXiv preprint 2407.21783 (2024)
11. Ggerganov, I.: llama.cpp: LLM inference in C/C++ (2024), https://github.com/ggerganov/llama.cpp, [Accessed Dec. 10, 2024]
12. Kachris, C.: A survey on hardware accelerators for large language models. Applied Sciences **15**(2), 586 (2025)
13. LLVM: The LLVM Compiler Infrastructure. https://llvm.org/, [Accessed Jul. 18, 2025]
14. Mozilla: powermetrics – Firefox Source Docs documentation. https://firefox-source-docs.mozilla.org/performance/powermetrics.html, [Accessed Jul. 18, 2025]
15. NVIDIA Corporation: NVIDIA TensorRT. https://developer.nvidia.com/tensorrt (2024), [Accessed: December 15, 2025]
16. Raiaan, M.A.K., Mukta, M.S.H., Fatema, K., Fahad, N.M., Sakib, S., Mim, M.M.J., Ahmad, J., Ali, M.E., Azam, S.: A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. IEEE Access (2024)
17. Ravitch, T.: Whole Program LLVM. https://github.com/travitch/whole-program-llvm (2024), [Accessed Dec. 11, 2024]
18. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. IEEE Transactions on Software Engineering **43**(1), 1–18 (2016)
19. Tay, Y., Dehghani, M., Bahri, D., Metzler, D.: Efficient transformers: A survey. ACM Comput. Surv. **55**(6) (Dec 2022). https://doi.org/10.1145/3530811, https://doi.org/10.1145/3530811
20. de la Torre, J.C., Ruiz, P., Dorronsoro, B., Galindo, P.L.: Analyzing the influence of LLVM code optimization passes on software performance. In: International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems. pp. 272–283. Springer (2018)
21. Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., Deng, Y.: WildChat: 1M ChatGPT Interaction Logs in the Wild. In: The Twelfth International Conference on Learning Representations (2024)