Optimización del Problema de Selección de *Flags* con los Compiladores `Clang` y `GCC`

Alberto Valderas-González[0009−0003−3280−8936], Juan Carlos de la Torre[0000−0002−3662−6422], y Bernabé Dorronsoro[0000−0003−0481−790X]

Escuela Superior de Ingeniería, Universidad de Cádiz, España
alberto.valderasgonzalez@alum.uca.es
{juan.detorre, bernabe.dorronsoro}@uca.es

Resumen. La optimización automática de compiladores constituye una estrategia clave para mejorar el rendimiento del software más allá de los niveles de optimización predefinidos. En este trabajo se estudia el problema de selección de *flags* de compilación para los compiladores `GCC` y `Clang`, empleando un Algoritmo Genético Celular. La metodología propuesta representa cada configuración como un vector binario que indica el uso, o no, de opciones de optimización, utilizando como función objetivo la minimización del tiempo de ejecución del programa resultante. Los experimentos se realizan sobre un programa que unifica diferentes características, construido a partir de una selección de *kernels* de *PolyBench*, considerando dos plataformas *hardware* distintas: un dispositivo de bajo consumo *Latte Panda Alpha* y un servidor de alto rendimiento. Los resultados muestran que las configuraciones obtenidas mediante el algoritmo evolutivo superan de forma significativa a los niveles estándar de optimización, especialmente en `GCC`, donde se alcanzan mejoras superiores al 70% respecto al nivel de optimización más agresivo proporcionado por el compilador. En `Clang` las mejoras son más moderadas, cercanas al 15%, lo que sugiere la necesidad de incorporar herramientas adicionales del ecosistema `LLVM`. Estos resultados evidencian que las configuraciones genéricas de los compiladores no siempre explotan plenamente el potencial de cada combinación *software-hardware*.

Palabras clave: Optimización de código · Tiempo de ejecución · Algoritmo genético celular · GCC · Clang.

1 Introducción

El rendimiento del software es un objetivo fundamental en el desarrollo de sistemas eficientes en diferentes ámbitos, como los sistemas embebidos, las aplicaciones científicas y las infraestructuras de computación de alto rendimiento. En este contexto, los compiladores modernos como `GCC` y `Clang`, este último basado en la infraestructura `LLVM` [15], desempeñan un papel esencial. Estas herramientas incorporan mecanismos de optimización diseñados para transformar el código fuente con el fin de mejorar diversas métricas de eficiencia, entre

ellas el tiempo de ejecución, el uso de memoria y el consumo energético [3]. Estas optimizaciones se aplican mediante una serie de transformaciones que, cuando se combinan adecuadamente, generan código ejecutable más eficiente sin alterar la semántica original del programa.

Sin embargo, la efectividad de estas transformaciones es muy sensible a la naturaleza del software, la arquitectura objetivo y la secuencia específica de optimizaciones aplicada. Aunque tanto `GCC` como `Clang` ofrecen niveles de optimización predefinidos, como `-O1`, `-O2` u `-O3`, estas configuraciones genéricas no garantizan alcanzar el máximo rendimiento en todos los escenarios [4, 17], e incluso pueden provocar una degradación del rendimiento en casos concretos [12].

Las optimizaciones orientadas al rendimiento pueden producirse en múltiples niveles, incluidas las transformaciones de código intermedio, el análisis de flujo de datos, el desenrollado de bucles, la vectorización y la planificación de instrucciones [16]. Esta complejidad implica que, a pesar de su madurez, la interacción entre múltiples opciones del compilador y las arquitecturas del procesador puede a menudo producir comportamientos impredecibles. Aunque la infraestructura `LLVM` es ampliamente utilizada en investigación por su modularidad y flexibilidad, `GCC` sigue siendo una alternativa habitual debido a la estabilidad proporcionada por décadas de actualizaciones.

La optimización de código continúa siendo un área de investigación de gran relevancia debido a su fuerte impacto en el rendimiento del software y a la heterogeneidad de los entornos *hardware* y software. Además, la búsqueda de configuraciones óptimas es un problema extremadamente complejo, ya que el espacio de búsqueda suele ser demasiado grande para ser abordado mediante enfoques exhaustivos.

La literatura presenta varias metodologías para abordar esta complejidad [6, 11]. Tradicionalmente, el ajuste heurístico manual se basaba en la experiencia del desarrollador y en observaciones empíricas. Sin embargo, a medida que los compiladores modernos incorporaron cientos de opciones de optimización, cuyo comportamiento suele variar en cada versión, el ajuste manual dejó de ser viable debido a su falta de escalabilidad y reproducibilidad.

Esto dio lugar a la aparición de algoritmos de *autotuning*, que exploran sistemáticamente el gran espacio de optimización proporcionado por los compiladores modernos [13]. Los algoritmos de búsqueda avanzados, como las metaheurísticas, han demostrado ser eficaces para identificar configuraciones de alta calidad en un tiempo razonable, equilibrando exploración y eficiencia. Herramientas como *OpenTuner* [2] o *TVM* [9] ejemplifican la exitosa integración de estas técnicas.

De forma paralela, los enfoques basados en aprendizaje automático también han ganado relevancia, utilizando aprendizaje supervisado o por refuerzo para predecir configuraciones efectivas a partir de características del código [7] o de representaciones intermedias [8]. Entre los ejemplos más destacados se encuentra el *framework MLGO* [19] de *Google*, que optimiza decisiones específicas como la inserción de funciones *inline* o la asignación de registros. Además, el *tuning* específico para una arquitectura busca aprovechar las particularidades del *hard-*

ware de destino, aunque esto suele limitar la portabilidad de las optimizaciones resultantes [10].

Para abordar estos desafíos, los algoritmos evolutivos se han convertido en herramientas muy utilizadas para la optimización de compiladores [12]. Su capacidad para navegar por grandes espacios combinatorios les permite encontrar soluciones que superan significativamente en calidad a las configuraciones estándar en un tiempo relativamente reducido. Habitualmente, debido a su modularidad y flexibilidad, la literatura tiende a favorecer el uso de la infraestructura `LLVM` frente a otros compiladores como `GCC`. Por este motivo, este trabajo propone un estudio de las capacidades de optimización de ambos compiladores mediante una metodología automatizada para determinar el conjunto de optimizaciones, más comúnmente llamadas *flags* de compilación, más adecuado para un programa dado. Esta metodología se basará en un trabajo previo de Aragón-Jurado et al. [4], aunque incorporando algunos cambios que la adaptan a los objetivos del estudio.

La principal contribución de este trabajo consiste en la comparación detallada de las capacidades de optimización de código de dos de los compiladores más utilizados en la literatura: `GCC` y `Clang`. Se realizarán compilaciones utilizando opciones predefinidas de optimización, así como de secuencias obtenidas de forma automática que, posiblemente, mejoren a aquellas opciones predeterminadas.

La estructura de este trabajo es la siguiente: en la Sección 2 se discutirá sobre el problema a resolver, los procesos de compilación que se llevarán a cabo, el algoritmo utilizado y lo que significa la incertidumbre en la medición de tiempos. Tras esto, en la Sección 3 se explicará la configuración experimental diseñada para este trabajo. En la Sección 4 se mostrarán los resultados, siendo por tanto discutidos en la Sección 5. Finalmente, las conclusiones obtenidas, además de posibles líneas de trabajo futuro, serán comentadas en la Sección 6.

2 Metodología

En esta sección se describe la metodología seguida para abordar la optimización del tiempo de ejecución mediante la selección de optimizaciones de compilación. En primer lugar, se formaliza el problema de selección de *flags*, tras lo que se detalla el proceso de compilación empleado con `GCC` y `Clang`, y se presenta el algoritmo genético celular utilizado para explorar el espacio de soluciones. Finalmente, se expone el procedimiento adoptado para reducir la incertidumbre asociada a la medición de los tiempos de ejecución.

2.1 Problema de Selección de *Flags*

Para este trabajo se empleará el Problema de Selección de *Flags* [12] (*FSP*). Esta representación, bastante común en la optimización de código mediante opciones del compilador, describe una secuencia de optimización como un vector de valores booleanos, donde cada valor indica la inclusión o exclusión de una determinada optimización en el proceso de compilación. El objetivo consiste, por tanto, en

obtener el conjunto que minimice una determinada función de *fitness*, que en este caso será el tiempo de ejecución.

Debido a la gran magnitud y complejidad del problema, en el que el espacio de búsqueda del *FSP* crece exponencialmente con el número de opciones de optimización disponibles, resulta inviable aplicar métodos exhaustivos para resolver el problema. En la Sección 2.3 se introduce el algoritmo genético utilizado en este trabajo.

2.2 Proceso de compilación

Los compiladores `GCC` y `Clang` fueron seleccionados para realizar estos experimentos principalmente debido a su amplio uso en la literatura. Sin embargo, estas herramientas presentan diferencias arquitectónicas significativas, lo que obliga a emplear un enfoque distinto para cada una de ellas.

Ambos compiladores proporcionan sus propias opciones de optimización y, por tanto, es necesario seleccionar aquellas que se incluirán en el conjunto T de *flags* de optimización disponibles. Para ello, la selección de opciones utilizables siguió los siguientes criterios:

- Selección: los *flags* deben contar con documentación clara y deben funcionar tanto de forma individual como en combinación de otros al compilar el software.
- Rechazo: se excluyen los *flags* parametrizados, es decir, aquellos que requieren valores numéricos específicos o argumentos externos.

El proceso de compilación para `GCC` puede realizarse con un único comando:

```
1 gcc -O3 <optimization flags> *.c -o exec -lm
```

Como puede observarse, se utiliza la opción predeterminada `-O3`, ya que el compilador requiere aplicar al menos un nivel predeterminado de optimización para que los *flags* de optimización individuales surtan efecto. De entre todos los *flags* de optimización disponibles se decidió utilizar en este trabajo `-O3`. El prefijo `-fno`, que puede concatenarse a todos los *flags* de optimización de este compilador, permite crear opciones auxiliares que desactivan explícitamente optimizaciones específicas habilitadas por defecto. Su uso duplica de manera efectiva el espacio de búsqueda, pero permite al algoritmo desactivar optimizaciones introducidas por `-O3`.

En el caso de `Clang` [14] se utilizan aquellas opciones contenidas en la invocación directa del compilador, evitando el uso del *framework* de optimización `opt` de `LLVM`. La razón es que su uso implicaría un problema de optimización diferente y más complejo, puesto que requiere no sólo especificar las transformaciones seleccionadas (problema de selección de *flags*) sino también, el orden en el que se deben aplicar las transformaciones (problema de orden de fases) [4].

Con respecto al comando de compilación a utilizar con `Clang`, la modularidad y flexibilidad de `LLVM` permite varias opciones que conducen a resultados diferentes. Las posibles opciones se analizan en la Sección 4.1.

2.3 Algoritmo Genético Celular

Se propone utilizar un Algoritmo Genético Celular [1] (cGA) para resolver el problema considerado. Se trata de una variante de algoritmo genético que implementa una población descentralizada, lo que le hace especialmente eficaz para mantener una alta diversidad poblacional durante períodos prolongados. Por ello, se considera un gran enfoque para abordar problemas combinatorios de esta naturaleza [4], superando a menudo a métodos evolutivos más convencionales.

En un cGA, los individuos se disponen en una malla toroidal, donde la interacción entre individuos se restringe a un pequeño vecindario local. Esta competencia localizada impide que la información de buenas soluciones se disemine rápidamente por toda la población, ralentizando así la convergencia y permitiendo al algoritmo mejorar su capacidad de exploración, sin mermar su potencial de explotación de las regiones más prometedoras.

Cada uno de los individuos, o cromosomas, del algoritmo será compuesto por una secuencia de valores booleanos que representan la inclusión o exclusión de cada una de las posibles *flag* de optimización. Por tanto, la secuencia final contendrá aquellas opciones cuyo valor correspondiente en el cromosoma así lo indique. La Fig. 1 muestra la estructura interna de un posible cromosoma y la secuencia resultante de éste.

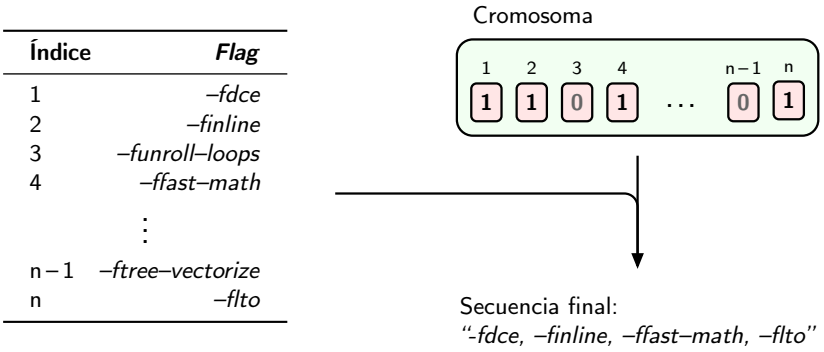


Fig. 1. Estructura interna de un cromosoma genérico de tamaño n junto con su secuencia de optimización producida.

2.4 Incertidumbre en la medición de tiempos

La medición del tiempo de ejecución es un tema ampliamente estudiado que puede verse afectado por diversos factores, entre ellos los procesos en segundo plano, la planificación del sistema operativo o incluso picos de tensión en la fuente de alimentación del sistema. Por ello, es necesario utilizar metodologías que minimicen esta incertidumbre temporal [5].

"Wcase15" [4] es un método que consiste en (1) tomar 15 mediciones independientes del tiempo de ejecución del programa, (2) descartar las diez mejores medidas (i.e., los 10 menores tiempos), y (3) calcular posteriormente un intervalo de confianza para el tiempo de ejecución mediante un proceso de *bootstrapping* utilizando las cinco mediciones restantes. Esta metodología ha demostrado una gran efectividad en la reducción de la incertidumbre en el tiempo de ejecución del software [4] y, por consiguiente, será la empleada para la medición del tiempo de ejecución.

3 Configuración experimental

Para este estudio se utilizarán las versiones 11.3 y 9.0.1 de `GCC` y `Clang`, respectivamente. Todas las ejecuciones se realizarán además en dos plataformas distintas: un dispositivo embebido de bajo consumo, y un servidor de altas prestaciones:

- Latte Panda Alpha: un dispositivo de alto rendimiento y bajo consumo equipado con un procesador Intel Core m3-8100Y de doble núcleo, con una frecuencia máxima de 3,4 GHz y 8 GB de RAM. El sistema operativo utilizado es Ubuntu 22.04.2 LTS.
- HPE Proliant XL220n: Nodo del clúster de supercomputación de la Universidad de Cádiz [20]. Está equipado con dos CPU Intel Xeon Platinum 8358, con 32 núcleos cada una (frecuencia de 2,6 GHz) y 256 GB de RAM. El sistema operativo es Rocky 8.6.

El programa utilizado para la optimización está compuesto por una selección de la conocida colección de *benchmarks PolyBench* [18] (algunos programas fueron descartados debido a sus tiempos de ejecución excesivos). Estos *kernels* se ejecutaron secuencialmente, simulando un software unificado completo, haciendo uso de una amplia variedad de operaciones:

- *2mm*
- *3mm*
- *atax*
- *bicg*
- *cholesky*
- *doitgen*
- *gemm*
- *gemver*
- *gesummv*
- *mvt*
- *symm*
- *syr2k*
- *syrk*
- *trisolv*
- *trmm*

Por último, los parámetros de configuración del cGA utilizado fueron establecidos experimentalmente, y pueden observarse en la Tabla 1. La población está formada por 100 individuos inicializados aleatoriamente y organizados en una malla toroidal de 10×10, donde cada individuo está compuesto por un cromosoma de distinta longitud, dependiendo de si la compilación se realiza con `GCC` o con `Clang`.

4 Resultados

Esta sección detalla el estudio preliminar realizado para optimizar el proceso de compilación de `Clang` y presenta los resultados obtenidos al aplicar el cGA para resolver el problema de *FSP* para tanto `GCC` como `Clang`.

Tabla 1. Configuración de parámetros para el cGA.

Algoritmo Genético Celular	
Longitud de cromosoma	212 (`GCC`) / 37 (`Clang`)
Tamaño de la población	100
Estructura de la población	10×10
Inicialización de individuos	Aleatoria (binaria)
Vecindad	L5
Selección de progenitores	Torneo binario
Cruce	Cruce de un solo punto
Probabilidad de cruce	$\rho_c = 1.0$
Mutación	*Bit-flip* aleatorio
Probabilidad de mutación	$\rho_0 = 0.05$
Reemplazo	Reemplazar si mejora
Ejecuciones independientes	30

4.1 Estudio preliminar para la compilación con `Clang`

Para determinar la configuración más efectiva para el proceso de compilación con `Clang`, se probaron tres secuencias de comandos diferentes. Todas las configuraciones se diseñaron siguiendo el mismo principio: deben generar representaciones intermedias (*IR*) del código para permitir utilizar algún *framework* de `LLVM` y así hacer compatible este estudio con otros de la literatura que pudieran mejorar aún más la calidad del proceso de optimización. Esto se consigue mediante el uso del *flag* `-emit-llvm` al inicio de la secuencia. Para todas las secuencias, los niveles de optimización predeterminados `-O0`, `-O1`, `-O2` y `-O3` serán utilizados para compilar el *benchmark*, y posteriormente se compararán los resultados obtenidos. Las secuencias estudiadas son:

Secuencia de compilación 1 Para esta primera secuencia se utiliza el *flag* `-disable-llvm-passes` al generar el *IR*, con el fin de forzar la desactivación de cualquier optimización adicional dentro del flujo de trabajo que no dependa completamente de `Clang` en sí. El segundo paso fusiona estas representaciones intermedias en un único archivo mediante `llvm-link`, mientras que el paso final completa las fases restantes de compilación para producir el ejecutable, utilizando `-O0` para garantizar que no se realicen optimizaciones adicionales no deseadas.

```
clang -O<n> -emit-llvm -Xclang -disable-llvm-passes -c *.c
llvm-link *.bc -o file.bc
clang -O0 -Xclang -disable-llvm-passes file.bc -o exec -lm
```

Secuencia de compilación 2 Esta secuencia sigue la misma lógica que la anterior, pero elimina el *flag* `-disable-llvm-passes`. Esto se hace para permitir que las optimizaciones puedan actuar de manera completa dentro del flujo de compilación de `Clang` en caso de que esto sea necesario.

```
clang -O<n> -emit-llvm -c *.c
llvm-link *.bc -o file.bc
clang -O0 file.bc -o exec -lm
```

Compilation Sequence 3 Finalmente, esta opción es idéntica a la anterior secuencia, con la excepción del uso de `-O3` en el comando final en lugar de `-O0`, para comprobar si esta diferencia afecta al resultado final de la compilación.

```
clang -O<n> -emit-llvm -c *.c
llvm-link *.bc -o file.bc
clang -O3 file.bc -o exec -lm
```

Para cada una de las secuencias descritas, así como para los niveles de optimización predeterminados, el *benchmark* fue compilado y su tiempo de ejecución tomado 1000 veces distintas en el clúster de supercomputación. Los resultados pueden observarse en la Fig. 2.

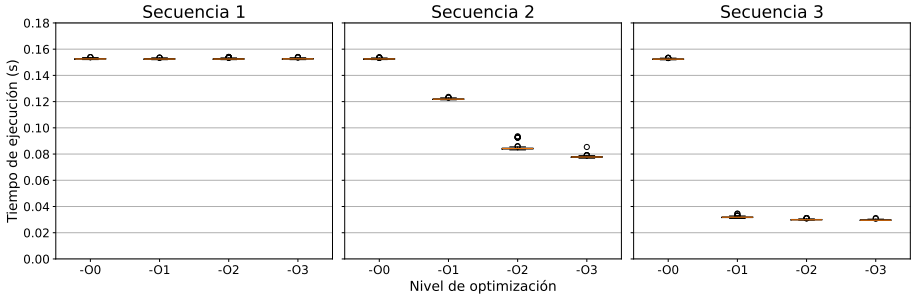


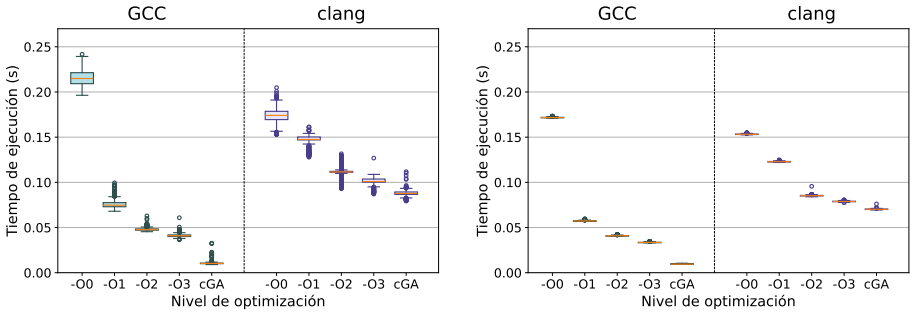
Fig. 2. Distribución de los tiempos de ejecución de 1000 mediciones independientes para cada secuencia de optimización estudiada y niveles de optimización predeterminados.

Como se puede ver, cada una de las secuencias de compilación actúan de una forma distinta, siendo la única que actúa de la forma esperada (mejoras incrementales según el nivel de optimización) la secuencia 2. Por ello, esa será la utilizada en el estudio final entre ambos compiladores. Esta secuencia será por tanto modificada para incluir *flags* de optimización de la siguiente manera:

```
clang -O3 <optimization flags> -emit-llvm -c *.c
llvm-link *.bc -o file.bc
clang -O0 file.bc -o exec -lm
```

4.2 Comparación de tiempos de ejecución

La Fig. 3 ilustra la comparación entre los tiempos de ejecución de las mejores soluciones encontradas por el cGA frente a los niveles de optimización predeterminados de `GCC` y `Clang` para ambas arquitecturas. Los valores presentados corresponden a 1000 mediciones independientes de cada compilación.



(a) LattePanda Alpha (b) HPE Proliant XL220n

Fig. 3. Comparación del tiempo de ejecución de la mejor solución encontrada por el algoritmo genético celular para ambos compiladores y arquitecturas frente a las opciones de optimización predeterminadas (1000 mediciones independientes).

Por una parte, a la izquierda de las Figuras 3a y 3b se pueden observar los resultados para `GCC`. La mejora obtenida respecto a la opción de optimización predeterminada más rápida, `-O3`, es considerable tanto para la *Latte Panda Alpha* como para el servidor HPE Proliant XL220n, alcanzando tiempos de ejecución inferiores a una centésima de segundo en ambas plataformas. Las compilaciones con `GCC` también muestran un salto masivo en el tiempo de ejecución entre los *flags* `-O0` y `-O1`.

Por otro lado, la parte derecha de las figuras presentan los resultados obtenidos con `Clang`. Las ejecuciones de las soluciones optimizadas tardaron aproximadamente la mitad de tiempo que las ejecuciones con el *flag* base `-O0`, logrando una mejora cercana al 15% respecto a la opción más agresiva `-O3`. Por otra parte, la distribución de los tiempos a lo largo de los niveles de optimización se muestra repartida.

La Tabla 2 muestra los porcentajes de mejora del tiempo de ejecución del *benchmark* frente a las *flags* de optimización predeterminadas más comunes: `-O0` y `-O3`. Como se puede apreciar, las mejoras más importantes se obtuvieron con `GCC`, y el nivel de mejora es similar para ambas arquitecturas.

Tabla 2. Mejora de rendimiento de las mejores soluciones encontradas por el cGA en comparación con los *flags* de optimización `-O0` y `-O3`.

Compilador	**Arquitectura**	`-O0`	`-O3`
`GCC`	Latte Panda Alpha	95,1%	74,4%
	HPE Proliant XL220n	94,4%	71,3%
`Clang`	Latte Panda Alpha	49,4%	13,3%
	HPE Proliant XL220n	54,6%	11,4%

5 Discusión de los resultados

Los resultados presentados en la Sección 4 permiten extraer varias conclusiones interesantes sobre los experimentos realizados.

En primer lugar, se observa que el cGA es capaz de identificar configuraciones de optimización que reducen significativamente el tiempo de ejecución del programa. Estas mejoras son especialmente notables en el caso de `GCC`, donde las reducciones del tiempo de ejecución superan el 70% en comparación con `-O3`, el nivel de optimización más efectivo.

En el caso de `Clang`, sin embargo, las mejoras obtenidas son más modestas. Esta discrepancia puede atribuirse a varios factores, siendo el más destacado las fases específicas del proceso de compilación en las que operan estas herramientas, ya que se exigió que su secuencia de compilación utilizara únicamente la invocación de `Clang` durante la optimización. Por el contrario, `GCC` trata toda la compilación como un proceso único, permitiendo así optimizar el código en un espectro mucho más amplio.

Otras observaciones adicionales sobre los resultados obtenidos son las diferencias en el tiempo de ejecución entre compiladores al optimizar el código únicamente mediante las opciones de optimización predeterminados como `-O0`, `-O1`, `-O2` y `-O3`. Se puede observar que, para `-O0`, `Clang` consigue una ejecución más rápida que `GCC`, lo que sugiere que, incluso cuando se indica explícitamente que no se optimice el código, los compiladores siempre realizan algunos ajustes menores que consideran adecuados. Sin embargo, a partir de `-O1`, `GCC` es capaz de superar fácilmente la compilación de `Clang`.

6 Conclusiones y trabajo futuro

Este trabajo propone el uso de un Algoritmo Genético Celular (cGA) para resolver el problema de selección de *flags* de optimización en programas informáticos, utilizando los compiladores de `GCC` y `Clang`. Se seleccionó como caso de estudio un programa compuesto por una selección de *kernels* del conocido *benchmark Polybench*, y se hicieron experimentos en dos plataformas distintas, una de bajo consumo y otra de alto rendimiento.

Tras la fase experimental, se observaron algunas aumentos en rendimiento, alcanzando mejoras de hasta el 70% para `GCC` en comparación con el nivel estándar

`-O3`, mientras que para `Clang` la mejora fue menor, situándose aproximadamente en un 15%.

Además, se concluye que `Clang` tiende a aplicar un mayor número de mejoras al código base incluso cuando no se especifican explícitamente optimizaciones, lo que sugiere un diseño interno orientado hacia la eficiencia por defecto. En cambio, `GCC` demuestra una capacidad superior para aprovechar los recursos del sistema tan pronto como se activan incluso optimizaciones mínimas. Esto pone de manifiesto la clara necesidad de utilizar herramientas adicionales de `LLVM`, como `opt`, junto con `Clang` para reducir la brecha de eficiencia entre ambos compiladores. Sin embargo, esto complica la dificultad del problema de optimización.

No obstante, en ambos casos, las configuraciones predeterminadas resultan insuficientes para alcanzar el máximo potencial de la combinación *hardware-software*. Es necesario recurrir a combinaciones de transformaciones más específicas y avanzadas para explotar plenamente las capacidades de la arquitectura subyacente.

Como trabajo futuro sería interesante incorporar herramientas adicionales de `LLVM`, especialmente `opt`, en la fase de compilación para estudiar si `Clang` puede reducir la brecha observada frente a `GCC` cuando se utiliza de forma más completa su infraestructura de optimización. Se podría explorar además una optimización multiobjetivo, considerando no solo tiempo de ejecución, sino también otras métricas como consumo energético o tamaño del binario.

Agradecimientos. Esta publicación ha sido financiada por el Fondo Europeo de Desarrollo Regional (FEDER) y la Consejería de Universidad, Investigación e Innovación de la Junta de Andalucía mediante los proyectos GRASS (FEDER-UCA-2024-A2-23) y gCODE (PIDI_2024_01307).

Declaración de intereses. Los autores declaran no tener conflicto de intereses.

Referencias

1. Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms, Operations Research/Computer Science Interfaces, vol. 42. Springer, Berlin, Germany (2008)
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: Opentuner: An extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 1–12 (2014). https://doi.org/10.1145/2628071.2628092
3. Aragón-Jurado, J.M., de la Torre, J.C., Ruiz, P., Dorronsoro, B.: Automatic software tailoring for green internet of things. Internet of Things p. 101521 (2025)
4. 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 (2023)
5. Aragón Jurado, J.M., Torre Macías, J.C.d.l., Talbi, E.G., Dorronsoro Díaz, B., et al.: A study on the influence of runtime uncertainty in the optimization of software programs (2021)

6. Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A survey on compiler autotuning using machine learning. ACM Computing Surveys (CSUR) **51**(5), 1–42 (2018)
7. Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J., Silvano, C.: Cobayn: Compiler autotuning framework using bayesian networks. ACM Trans. Archit. Code Optim. **13**(2) (Jun 2016). https://doi.org/10.1145/2928270, https://doi.org/10.1145/2928270
8. Cavazos, J., Moss, J.E.B.: Inducing heuristics to decide whether to schedule. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. p. 183–194. PLDI '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/996841.996864
9. 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)
10. Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., Wu, C.: Evaluating iterative optimization across 1000 datasets. SIGPLAN Not. **45**(6), 448–459 (Jun 2010). https://doi.org/10.1145/1809028.1806647, https://doi.org/10.1145/1809028.1806647
11. Dorronsoro, B., Aragón-Jurado, J.M., Jareño, J., de la Torre, J.C., Ruiz, P.: A survey on automatic source code transformation for green software generation. In: Abraham, M.A. (ed.) Encyclopedia of Sustainable Technologies (Second Edition), pp. 765–779. Elsevier (2024)
12. Garciarena, U., Santana, R.: Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion. pp. 1159–1166 (2016)
13. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to GPU codes. In: 2012 innovative parallel computing (InPar). pp. 1–10. Ieee (2012)
14. LLVM: Clang: a C language family frontend for LLVM. https://clang.llvm.org/
15. LLVM: The LLVM Compiler Infrastructure. https://llvm.org/
16. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
17. Olvera Espinosa, C.: Evolving Trends in Software Performance and Energy Consumption: An Analysis of Compilers and Interpreters over Time. Master's thesis, Aalto University (2024)
18. Pouchet, L.N.: PolyBench/C – Homepage of Louis-Noël Pouchet. http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/
19. Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., Li, X.D.: Mlgo: a machine learning guided compiler optimizations framework (2021), https://arxiv.org/pdf/2101.04808
20. Universidad de Cádiz: Supercomputación en la UCA. https://supercomputacion.uca.es/ (2023)