OPTIMIZACIÓN DE MODELOS DE IA GENERATIVA EN HARDWARE DE ÚLTIMA GENERACIÓN

OPTIMIZING GENERATIVE AI MODELS ON NEXT-GENERATION HARDWARE

TRABAJO FIN DE GRADO
CURSO 2024-2025

AUTOR
OLGA POSADA IGLESIAS
NOTA: 7.9

DIRECTOR
CARLOS GARCÍA SÁNCHEZ

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

OPTIMIZACIÓN DE MODELOS DE IA GENERATIVA EN HARDWARE DE ÚLTIMA GENERACIÓN

OPTIMIZING GENERATIVE AI MODELS ON NEXT-GENERATION HARDWARE

TRABAJO DE FIN DE GRADO EN INGENIERÍA DE COMPUTADORES

AUTOR
OLGA POSADA IGLESIAS

DIRECTOR
CARLOS GARCÍA SÁNCHEZ

CONVOCATORIA: SEPTIEMBRE 2025

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

31 DE AGOSTO DE 2025

DEDICATORIA

A Lía.

AGRADECIMIENTOS

A todos aquellos que han estado ahí estos años, en especial a Esther, Rivera, Paula, Mario, Asur, Cris, David, Maya, Raquel, Óscar, Neli, Celia, Clau, Bea y Enri.

RESUMEN

OPTIMIZACIÓN DE MODELOS DE IA GENERATIVA EN HARDWARE DE ÚLTIMA GENERACIÓN

El objetivo de este Trabajo de Fin de Grado es analizar y mejorar la eficiencia y el rendimiento de modelos de inteligencia artificial generativa en hardware de última generación. En esta investigación identificaremos los principales cuellos de botella computacionales relacionados con la ejecución de modelos generativos y sugeriremos métodos para optimizarlos y aumentar su eficiencia. Para esto, se han analizado distintas configuraciones de hardware y software, utilizando diversas estrategias de disminución de latencia, gestión de recursos eficaz y ajuste de parámetros. Los resultados obtenidos permiten formular recomendaciones prácticas para la implementación de modelos de IA generativa en escenarios reales, y así poder tener un mejor equilibrio entre rendimiento, consumo energético y sostenibilidad.

Palabras clave

Inteligencia Artificial, modelos generativos, optimización, hardware, rendimiento, aceleradores, latencia, eficiencia.

ABSTRACT

OPTIMIZING GENERATIVE AI MODELS ON NEXT-GENERATION HARDWARE

The objective of this thesis is to analyze and improve the efficiency and performance of generative artificial intelligence models on next-generation hardware. In this research, we will identify the main computational bottlenecks related to the execution of generative models and suggest methods to optimize them and increase their efficiency. To this end, we have analyzed different hardware and software configurations, using various strategies to reduce latency, manage resources effectively, and adjust parameters. The results obtained allow us to formulate practical recommendations for the implementation of generative AI models in real-world scenarios, thereby achieving a better balance between performance, energy consumption, and sustainability.

Keywords

Artificial Intelligence, generative models, optimization, hardware, performance, accelerators, latency, efficiency.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 - Introducción

1.1 Motivación

La Inteligencia Artificial generativa es una de las ramas más innovadoras de la inteligencia artificial, tal y como se afirma en el artículo "Generative Artificial Intelligence: A Systematic Review and Applications" [1]. A diferencia de la inteligencia artificial clásica, que se centra en modelos de clasificación o de predicción, la inteligencia artificial generativa es capaz de generar material original en múltiples formatos (texto, imágenes, vídeo, audio, código, etc.) ante una orden o instrucción del usuario. Dentro de la inteligencia artificial generativa destacan los modelos de lenguaje de gran escala, conocidos como LLMs (Large Language Models).

Los LLMs son modelos basados en aprendizaje auto-supervisado, es decir, el modelo se entrena con datos de entrada (texto no etiquetado), de esta manera el modelo aprende a predecir los siguientes tokens. Lo que diferencia los LLMs de otros modelos basados en este tipo de aprendizaje es que los LLMs se entrenan con volúmenes inmensos de datos textuales, consiguiendo aprender representaciones distribuidas del lenguaje. Para aprender estas también ha sido necesaria la arquitectura llamada transformador, que es un modelo de redes neuronales basado en mecanismo de atención que procesan secuencias en paralelo, permitiendo capturar dependencias de largo alcance en secuencias textuales. De esta manera consiguen generar texto coherente y contextualizado.

Los LLMs han revolucionado la inteligencia artificial, expandiéndose desde entornos de investigación hasta aplicaciones cotidianas tan comúnmente conocidas como ChatGPT. Tal y como indica Anand Gokul en su artículo "LLMs and AI: Understanding its reach and impact" [2], los LLMs están transformando la forma en que nos comunicamos y comprendemos el lenguaje, con un alcance que afecta tanto al ámbito profesional como al personal.

Actualmente la discusión presente sobre la sostenibilidad y el impacto a largo plazo de la inteligencia artificial (en adelante acotada como IA) se centra principalmente en la fase de entrenamiento de los modelos, debido a que conlleva una

excesiva demanda de recursos energéticos, de fuentes hídricas, así como de recursos materiales. Emplear un modelo de gran escala como es el GPT-3, a título de ejemplo, precisó cifras superiores a 1.200 MW/h y emitió aproximadamente 550 toneladas de dióxido de carbono. Por lo demás, consumió más de 700.000 litros de agua para el acondicionamiento térmico. No obstante, estudios recientes señalan que el impacto ambiental de la fase de inferencia es más delicado, ya que, a diferencia de la fase de entrenamiento, la inferencia es continua y a gran escala, siendo un 90% del gasto total del ciclo de vida de los LLMs [3]. Esto resalta la importancia actual de la optimización de esta fase de inferencia en términos de consumo y sostenibilidad, punto clave en nuestro estudio.

En esta línea argumental, Jegham et al. [3], How hungry is IA? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference, representa uno de los primeros procedimientos sistematizados para cuantificar la huella ambiental de la inferencia en modelos de lenguaje de vanguardia. El estudio de investigación abarca treinta modelos de OpenAI, Anthropic, Meta, DeepSeek. Entre los resultados más destacables, se resalta que O3 y DeepSeek-R1 son los más intensos, excediendo los 33 Wh por consulta larga, lo que equivale a 70 veces el consumo de GPT-4.1 nano, uno de los modelos más eficientes. En oposición, Claude-3.7. Sonnet se corona como el modelo más sostenible, ya que logra el equilibrio entre capacidad y sostenibilidad con el medioambiente.

Una característica fundamental que refleja el estudio es el rol central de la infraestructura de despliegue. Por ejemplo, GPT-4º mini, aunque es un modelo de menor escala, resulta menos eficiente que GPT-4º al efectuarse en un hardware con menos avances tecnológicos (A100 vs. H100/H200). Del mismo modo, los modelos de DeepSeek revelan huellas hídricas sobredimensionadas en relación con sus centros de datos en China, con mayores valores de PUE y sistemas de refrigeración menos ecológicos.

Los resultados muestran que la sostenibilidad de la IA, no solo depende del diseño del algoritmo, sino también de variables externas, como, por ejemplo, el tipo de hardware, la eficiencia de los centros de datos y las fuentes energéticas, entre otros factores implicados. Asimismo, se destaca la importancia de implementar estándares para certificar la transparencia y la regulación de la huella ambiental por cada consulta,

fomentar el uso de energías renovables y promover tecnología de refrigeración sostenible con el planeta.

1.2 Objetivos

El presente trabajo tiene como finalidad explorar y evaluar la optimización de la inferencia de modelos de inteligencia artificial generativa en hardware de última generación, prestando especial atención a su eficiencia en entornos edge y a su sostenibilidad computacional. A partir de la motivación expuesta, se plantean los siguientes objetivos:

1.2.1 Objetivo general

Analizar y validar un pipeline de optimización para modelos de IA generativa en dominios de texto e imagen, empleando frameworks abiertos y runtimes especializados, con el fin de mejorar el rendimiento y la eficiencia en hardware Intel de nueva generación.

1.2.2 Objetivos específicos

- Implementar un entorno experimental basado en Python y Ubuntu que integre las principales librerías y frameworks del ecosistema actual: PyTorch, Hugging Face Transformers, ONNX y OpenVINO.
- Seleccionar y ejecutar modelos representativos de lenguaje (DistilGPT-2 y Tiny-GPT-2) e imagen (Stable Diffusion 1.5 y Stable Diffusion Turbo), comparando su comportamiento en distintos escenarios de optimización.
- Evaluar el impacto de técnicas de cuantización (FP32 vs. INT8) sobre la latencia, el throughput, el consumo de memoria y el uso de CPU.
- Analizar la portabilidad de modelos mediante la exportación a ONNX y su ejecución en ONNX Runtime, contrastando resultados frente a PyTorch.
- Medir las ganancias obtenidas con OpenVINO Runtime en hardware Intel, destacando su relevancia para el despliegue eficiente en dispositivos edge.

- Reflexionar sobre la sostenibilidad de la IA generativa, considerando el impacto energético y material de la fase de inferencia y la importancia de infraestructuras eficientes.

1.3 Plan de trabajo

El desarrollo de este trabajo se estructuró en una serie de fases encadenadas que permitieron avanzar desde la revisión teórica hacia la experimentación práctica y el análisis de resultados. La planificación contempló tanto la dimensión conceptual como la implementación técnica, con el fin de asegurar la coherencia entre motivaciones, objetivos y resultados obtenidos.

El proyecto fue planificado contemplando tanto la implementación técnica como su dimensión conceptual y estructurado en las siguientes fases:

1) Revisión del estado del arte y definición del ecosistema de software
   Se realizó una revisión bibliográfica sobre el ecosistema actual que rodea a la IA generativa, para así identificar las herramientas a emplear.
2) Configuración del entorno experimental
   Se implementó un entorno de pruebas que permitiese el acceso a las librerías y frameworks necesarios para la correcta ejecución de la fase de experimentación.
3) Selección y preparación de modelos representativos
   Se seleccionaron los distintos modelos de lenguaje y de imagen a utilizar y se planificó su optimización.
4) Ejecución de experimentos y escenarios de optimización
   Se llevaron a cabo ejecuciones en múltiples escenarios, primero sentando una base de referencia, y luego según se iba optimizando y afinando el modelo.
5) Análisis comparativo de resultados
   Se procesaron los datos obtenidos mediante las herramientas de análisis planificadas y se visualizaron para así poder generar comparativas entre modelos y entornos.
6) Discusión y conclusiones

Por último, se discutieron los resultados en relación a la literatura revisada y destacando las mejoras de rendimiento alcanzadas y posibles líneas de trabajo futuras.

Capítulo 2 - Estado de la cuestión

Como resultado del gran consumo energético de los modelos de lenguaje a gran escala, se ha despertado un gran interés en la optimización de estos modelos en hardware avanzado. La eficiencia energética pasa a ocupar un lugar céntrico del debate por el incremento de las emisiones de carbono causado por la adopción de los LLMs [4]. Se proponen numerosas técnicas entre las cuales destacan dos que pueden reducir el consumo en cuestión hasta un 45% sin necesidad de repercutir negativamente sobre la eficacia operativa. Estas técnicas, la inferencia local y la cuantización, ponen un foco sobre la relevancia de la optimización para conseguir una Inteligencia Artificial más sostenible. El estudio empírico realizado por Chen et al. en 2024 acerca del consumo de energía de LLMs en diferentes plataformas GPU establece que la eficiencia energética durante la inferencia se ve notablemente afectada por la escala del modelo y la arquitectura de hardware [5]. Según estos estudios, el rendimiento y sostenibilidad deben ir de la mano a la hora de programar IA generativa y esto sirve de motivación para esta investigación en el escenario de hardware edge.

En este capítulo discutiremos la tecnología y las diferentes técnicas de optimización que existen actualmente.

2.1 Modelos generativos

2.1.1 Modelos generativos para texto

Vaswani et al. [6] introduce la arquitectura llamada transformador, que supone un momento crucial en el procesamiento del lenguaje natural al erradicar cualquier necesidad de recurrencias o convoluciones por apoyar el aprendizaje sobre mecanismos de atención. A partir de este giro en la metodología se establecen los fundamentos necesarios para los grandes modelos de lenguaje.

A partir de aquí, se presenta GPT-2 [7], que tiene la capacidad de generar a partir de datos no supervisados textos coherentes y contextualizados. Su importancia consiste en probar que los modelos pueden conseguir capacidades multitarea sin ajustes específicos entrenando sobre corpus extensos. Más tarde se introduce GPT-3 [8], que

reduce la obligación de entrenamiento adicional y da resultados en redacción, traducción y razonamiento. Este modelo que afianzó la idea de few-shot learning contaba con 175.000 millones de parámetros.

En fechas más recientes [9] se desarrolla LLaMA. Se trata de una familia de modelos eficientes y ligeros que compensa la predilección por la opacidad y centralización de recursos en escasas organizaciones y democratiza el acceso a LLMs de alto rendimiento. Los avances en el entorno textual muestran progreso hacia una arquitectura más flexible, eficiente y escalable.

2.1.2 Modelos generativos para imagen

Para el campo de la visión a través de computador, las GANs (Generative Adversarial Networks) supusieron un cambio incorporando un marco competitivo entre generador y discriminador [10]. Este cambio facilita la generación de imágenes sintéticas con un alto nivel de realismo. Otras innovaciones como StyleGAN [11] sirvieron para pulir la generación de caras y estilos visuales con niveles de control inéditos.

Sin embargo, fue necesario el nacimiento de los modelos de difusión por las restricciones de la diversidad de muestras y la estabilidad de entrenamiento. Se demuestra que, en cuanto a síntesis, estas arquitecturas eran superiores a las GANs, dando lugar a una teoría distinta [12]. Uno de los avances destacados es Stable Diffusion que instaura la aplicación de espacios docentes con el objetivo de conseguir generación en entornos abiertos, accesible y eficiente [13]. De manera simultánea, Imagen conseguía que el escalado en parámetros y datos alcanzase resultados fotorrealistas partiendo de descripciones textuales.

El progreso en esta área deja ver una evolución partiendo desde arquitecturas adversariales hacia procesos de difusión probabilística, consiguiendo aumentar el control y la fidelidad en la síntesis de imágenes.

2.1.3 Modelos generativos para audio

También se puede observar un crecimiento evidente en la síntesis de audio. WaveNet [14] es un modelo autoregresivo que puede generar raw audio (audio en crudo), consigue voces humanas naturales y realistas. El gran coste computacional de

esta IA con calidad excelente propició la búsqueda de otras con más eficiencia como WaveGlow [15] y sistemas basados en espectrogramas como Tacotron [16]. Estos incrementaron la calidad de la expresividad y entonación de las voces.

El último límite queda ejemplificado por el modelo VALL-E [17]. Se trata de un modelo de Microsoft que se basa en aprendizaje tipo language model y técnicas de codificación neuronal para así poder generar voces a partir de pequeñas muestras de tan solo unos segundos. Esta estrategia ofrece oportunidades en accesibilidad y personalización mientras que levanta debates éticos relativos a la clonación de voz.

Resumiendo, el curso que sigue este ámbito manifiesta una evolución desde modelos autorregresivos a arquitecturas híbridas y neuronales más escalables que hacen posible la generación de música y voz con una gran fidelidad.

2.1.4 Modelos multimodales

El rumbo actual de la IA generativa está basado en integrar modalidades múltiples en un solo marco de aprendizaje. El modelo Flamingo [18] y otros similares fusionan el procesamiento de texto y de imagen para llevar a cabo razonamientos visuales con escasos ejemplos. A partir de este punto de vista se ponen los cimientos para arquitecturas más robustas.

El informe técnico de GPT-4 [19] significa un logro al integrar capacidades multimodales de texto e imagen y mejorar la interacción en aplicaciones prácticas. Al mismo tiempo, Gemini [20] expande el espectro incorporando texto, imagen, audio y video para asentar un marco unificado de generación y comprensión de información multimodal. Estas innovaciones pronostican un tiempo futuro donde los modelos generativos actúen como sistemas cognitivos generalistas y puedan operar en entornos ricos y complejos.

2.2 Necesidad de optimización en Hardware Edge

El desarrollo rápido de la inteligencia artificial generativa ha traído consigo un incremento del consumo de recursos computacionales. Los modelos de gran escala, tales como los modelos de difusión en visión o los transformadores aplicados en lenguaje natural, necesitan de GPUs de alta gama o clústeres especializados para poder realizar

su entrenamiento e inferencia. Ahora bien, con la necesidad de llevar estas capacidades a dispositivos tales como IoT, móviles o nodos Intel NUC, que son más compactos, surge la complicación de necesitar una optimización en hardware Edge [21].

La **latencia de red** (en inferencia con recursos cloud), el **coste**, la **seguridad**, la **privacidad** y la **eficiencia energética** motivan la realización de esta transición y la **necesidad de inferencia en el nodo edge**. Con la ejecución de los modelos en Edge se consigue reducir la latencia ya que se evita la necesidad de enviar de manera constante datos a la nube. Como subraya Candanedo et al. [22], Edge computing mitiga la dependencia de redes externas, proporcionando respuestas en tiempo real, necesaria en aplicaciones tales como el reconocimiento facial o en monitorizaciones biométricas.

Además, con los despliegues locales se consigue un menor coste operativo, debido a no ser tan necesario el uso de infraestructuras centralizadas en servidores. Seguidamente, el procesamiento de la información en el origen mejora la privacidad de los datos, siendo esto de alta importancia en áreas como la salud o la seguridad [23]. Por último, en un factor tan determinante como es el de la eficiencia energética, los dispositivos Edge disponen de limitaciones de memoria, batería y capacidad de cómputo, obligando así a realizar técnicas de reducción de consumo sin lastrar el rendimiento [24].

Aunque una gran parte de los estudios se centran en el análisis de la eficiencia energética de los LLMs en infraestructuras de alto rendimiento, sus datos no son aplicables a dispositivos Edge. Por ejemplo, Chen et al. [5] demuestran que la elección de la plataforma GPU y la escala del modelo son factores decisivos para la eficiencia energética durante la inferencia. Sin embargo, los dispositivos Edge, tales como los Intel NUC, tienen limitaciones en memoria, en su capacidad de cómputo y en la disipación térmica, obligando a aplicar técnicas de optimización específicas. Teniendo esto en cuenta, el reto no solo radica en seleccionar el hardware adecuado, sino en la adaptación de modelos y algoritmos para aprovechar al máximo los recursos que se ven más limitados que los dispositivos localizados en entornos de centro de datos.

Diversas investigaciones han tratado diferentes soluciones de optimización. Martín [25] resalta el diseño de nodos inalámbricos que combinan técnicas de IA y ciberseguridad en el Edge de IoT, consiguiendo con esto un balance entre rendimiento, consumo y latencia. Estrategias como el model pruning, la cuantización o la inferencia distribuida permite la adaptación de redes neuronales profundas a entornos con recursos limitados [26]. Asimismo, Intriago [27] señala que la Industria 4.0 se beneficia del despliegue de Edge en la minimización de costes de transmisión y en conseguir una latencia baja en procesos industriales críticos.

El desafío no reside únicamente en trasladar modelos a dispositivos compactos, sino en la reconfiguración de las arquitecturas y los algoritmos para un mejor aprovechamiento del hardware heterogéneo disponible. El Edge computing debe ser visto como un componente clave en el continuum de la computación, donde la proximidad a los datos se ve reflejada en eficiencia, autonomía y sostenibilidad, esto es lo que resalta Robles Enciso [28].

En conclusión, la necesidad de optimización en hardware Edge refleja una tensión entre las limitaciones físicas de los dispositivos compactos y las crecientes exigencias de la IA generativa. La resolución de este reto conlleva un esfuerzo coordinado en los campos de la investigación de arquitecturas eficientes, técnicas de compresión y marcos de ejecución ligeros, para asegurar que todos los beneficios de la IA se puedan llevar a entornos de alta privacidad, bajo consumo y mínima latencia.

2.3 Técnicas de optimización

2.3.1 Optimización a nivel de modelo

La implementación de modelos de inteligencia artificial en dispositivos móviles, IoT o hardware edge ha presentado una nueva problemática: la implementación de sistemas de IA eficientes y sostenibles [29]. Debido a que la IA tiene un coste computacional elevado, se debe buscar la manera de implementarla en estos entornos que cuentan con recursos limitados sin comprometer significativamente la precisión.

A continuación, se van a exponer tres técnicas de optimización de latencia, consumo energético y uso de memoria: la cuantización, el pruning y la distilación de conocimiento.

2.3.1.1 Cuantización

La cuantización es una técnica que optimiza el uso de memoria. Este método convierte los pesos y activaciones de un modelo de IA, que normalmente son representados en precisión flotante de 32 bits (FP32), a formatos menos pesados como INT8, INT4 o incluso binarios. Así, no solo se reduce el uso de memoria, sino que además se acelera la inferencia, especialmente en procesadores que soportan instrucciones optimizadas para enteros [30].

Existen dos enfoques en la cuantización, Post-Training Quantization (PTQ) y Quantization-Aware Training (QAT). En PQT se aplica la reducción de precisión una vez entrenado el modelo y en QAT se entrena el modelo teniendo en cuenta las restricciones de cuantización, lo que mejora la robustez frente a pérdidas de precisión [31]. Algunas herramientas que aplican la cuantización son Intel Neural Compressor, ONNX Runtime y OpenVINO.

Por ejemplo, Soni et al. [32] aplicaron cuantización en redes neuronales para IoT, logrando reducciones del 70% en memoria y aceleración de hasta 3× en inferencia sin pérdidas significativas de precisión, demostrando la efectividad de aplicar la cuantización.

Además, el trabajo de Manoj et al. [33] resalta su aplicación en sistemas inalámbricos, donde la baja latencia es prioritaria.

2.3.1.2 Poda

La poda, o pruning, es una técnica que busca reducir el tamaño y coste computacional de los modelos. Este procedimiento se basa en eliminar parámetros o elementos redundantes o que no contribuyen mucho.

Existen dos tipos de pruning, estructurado y no estructurado. En el primero se eliminan canales, filtros o capas completas, manteniendo la uniformidad de la arquitectura. Esto es útil en hardware paralelo como GPUs. Y, en el pruning no

estructurado se suprimen conexiones individuales dispersas [34], proporcionando una mayor flexibilidad, aunque esto conlleva la necesidad de un soporte más especializado.

Investigaciones de Marvellous et al. [35] en wearables con baterías limitadas demostraron que un pruning agresivo puede reducir hasta un 60% el consumo energético, con degradaciones mínimas en exactitud. Por ello, el pruning se ha convertido en una técnica fundamental para alargar la vida útil de dispositivos portátiles.

Un experimento de Harris [36] exploró la combinación del pruning con otras técnicas en drones de vigilancia autónomos, consiguiendo mejoras en tiempo real con modelos de visión artificial que antes eran demasiado pesados para hardware compacto, mostrando que el pruning no solo economiza memoria, sino que también habilita aplicaciones que necesitan una alta capacidad de respuesta.

2.3.1.3 Knowledge Distillation

La última técnica de optimización a nivel de modelo que se va a explorar es la destilación de conocimiento. Este procedimiento permite entrenar un modelo compacto (student) para imitar el comportamiento de un modelo más grande (teacher). Fue propuesto inicialmente por Hinton et al. [37], y está ganando relevancia en la optimización de Large Language Models (LLMs) y arquitecturas de visión.

Uno de los ejemplos prácticos de destilación es DistilBERT [38], este modelo reduce en un 40% los parámetros de BERT manteniendo más del 95% de su rendimiento original. Esta técnica es útil en entornos que buscan un balance entre eficiencia y precisión, como en asistentes virtuales móviles o sistemas de traducción en tiempo real.

Noura et al. [39] destacan que la destilación puede combinarse con pruning y cuantización para lograr modelos aún más ligeros. Por ejemplo, en la detección de objetos en dispositivos IoT, se ha mostrado que los modelos comprimidos mediante destilación logran un rendimiento parecido al del modelo maestro, con reducciones importantes en energía y memoria. Además, Aach et al. [40] plantean su uso en edge AI sobre sistemas HPC, ya que la destilación es útil en la transferencia de capacidades complejas a nodos locales con hardware limitado.

2.3.2 Optimización en entrenamiento

Cuando hablamos de optimizar un LLM en entrenamiento, nos referimos a aplicar diferentes técnicas que permitan que el modelo se entrene y de esta manera aprenda, de un modo más eficiente, reduciendo tiempo, coste y consumo sin perder calidad en los resultados.

En el artículo Large Scale Distributed Deep Networks [41], se presenta la técnica de optimización en entrenamiento mediante paralelismo de datos aplicado a aprendizaje profundo. El paralelismo de datos en entrenamiento de redes neuronales profundas consiste en dividir los datos de entrenamiento en varios nodos para que cada uno de ellos lo procese, y finalmente sincronizar los gradientes. Centrándonos en modelos de lenguaje de gran escala, en el artículo Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism [42] consiguen aplicar este paralelismo en entrenamiento a LLMs. Introducen el concepto de paralelismo a nivel de modelo, que consiste en dividir capas y operaciones de atención propias de los LLMs entre múltiples GPUs. Llegan a demostrar que esta técnica es fundamental para entrenar modelos con miles de millones de parámetros.

Por otro lado, con motivo de reducir coste en memoria y cómputo, Micikevicius et al. [43] propone la técnica mixed precision training, es decir, entrenar redes neuronales con representación de menor precisión como FP16 o bfloat16, demostrando que reduce el uso de memoria y acelera el entrenamiento sin pérdida significativa de calidad. Siguiendo en la línea de reducción de coste en memoria, Chen et al. [44] introduce la técnica de gradient checkpointing, que consiste en recalcular gradientes bajo demanda, en vez de almacenarlos todos. Asimismo, optimizadores eficientes como AdamW [45], LAMB [46] o Adafactor [47] han sido diseñados para manejar escalas masivas de parámetros con menor coste de convergencia.

Por último, Schacht y Lanquillon [48] analiza cómo el parameter-efficient tuning, que consiste en ajustar solo una pequeña parte de los parámetros de un LLM manteniendo el resto congelado, y el transfer learning, que aprovecha el conocimiento de un modelo ya entrenado en una tarea amplia para aplicarlo a otra más específica, reducen de forma significativa los costes energéticos y económicos del entrenamiento

completo de modelos fundacionales, grandes modelos entrenados en datos generales que sirven como base para múltiples aplicaciones. Además, insiste en la importancia de reutilizar estos modelos fundacionales para aplicaciones concretas.

2.3.3 Optimización en inferencia

Cuando hablamos de optimizar un LLM en inferencia, nos referimos a mejorar el momento en que el modelo ya entrenado genera respuestas, aplicando métodos que reduzcan la latencia y el coste computacional en la predicción, garantizando respuestas más rápidas y económicas sin sacrificar precisión.

Una de las técnicas más conocidas en la fase de inferencia es el Key-Value Caching, consiste en que el modelo almacena los resultados de autoatención intermedios, de esta manera, el modelo atiende a los nuevos tokens generados y no tiene que calcular la atención completa. Matizando, en el artículo [49] se propone una arquitectura de gestión por cabezal de atención que permite hacer inferencia en contextos de hasta un millón de tokens en una sola GPU de consumo.

Por otro lado, Chen, W., Zhang, Y., Zhou, Z., Li, Y., & Wang, C. [50] investigan el método Staged Speculative Decoding en el que utilizan un modelo auxiliar o etapas intermedias para predecir múltiples tokens candidatos en paralelo, para luego verificarlos por el modelo principal. De esta manera consiguen aceleraciones de hasta un x3.16, sin perder la calidad de las salidas. Asimismo, en [51] se elimina la necesidad de tener un modelo adicional. Integran el proceso especulativo en un único pipeline, consiguiendo aceleraciones parecidas.

Finalmente, en el artículo [52] mencionan el uso de algoritmos especializados para mejorar la eficiencia de los cálculos de atención. Demuestran que FlashAttention reduce la complejidad de la operación de atención gracias a una computación por bloques y acceso directo a memoria de GPU.

2.3.4 Optimización de infraestructura

Cuando hablamos de optimizar un LLM a nivel de infraestructura, nos referimos a gestionar de manera más eficiente los recursos de hardware y software que lo soportan, empleando configuraciones y arquitecturas distribuidas que reduzcan el consumo

energético y los costes operativos mientras se mantiene la capacidad de escalar a gran escala.

Una faceta esencial es el despliegue eficaz. La portabilidad de modelos se puede simplificar y encapsular los ambientes de ejecución con herramientas de contenedorización, como Singularity o Docker [53]. De igual manera, marcos de model serving como FastAPI o TorchServe permiten desarrollar arquitecturas de inferencia flexibles, ya sea en modo streaming o batch. El enfoque de inferencia sin servidor está adquiriendo impulso gracias a la posibilidad de escalar recursos de manera dinámica dependiendo del volumen de trabajo.

Tal y como investigan Li, H., Zhang, Q., Chen, J., & Xu, H. [54], herramientas de compilación y optimización para hardware específico como ONNX Runtime, TensorRT u OpenVINO traducen los modelos a representaciones optimizadas con kernels especializados, reduciendo la latencia y consumo energético. OpenVINO produce representaciones intermedias (IR) optimizadas para NPU y CPU, lo que permite realizar inferencias aceleradas y eficientes en hardware de borde [55]. Este método fusiona hardware compacto y software especializado, lo que lo hace una opción estratégica para democratizar la IA generativa.

2.4 Hardware Edge: Intel NUC como caso de estudio

El Intel NUC (Next Unit of Computing) es una línea de equipos de formato reducido desarrollada por Intel, ofrecen un rendimiento competitivo en un espacio físico muy compacto. Estos dispositivos destacan por su bajo consumo energético, facilidad de despliegue y capacidad de actualización en componentes clave como memoria RAM o almacenamiento. Además, son compatibles con distintos sistemas operativos, lo que hace que sean una opción versátil para aplicaciones en entornos de edge computing [56], [57].

Debido a la integración de aceleradores específicos como las Intel Movidius VPUs, el uso de NUCs en escenarios de cómputo distribuido se ha visto potenciado. Esta integración junto con el ecosistema de herramientas OpenVINO permiten optimizar la inferencia de modelos de visión por computadora en condiciones de hardware limitado [58].

2.4.1 Ventajas del Intel NUC en entornos Edge

En primer lugar, referenciando la eficiencia energética, los NUCs pueden instalarse en escenarios donde la disipación térmica y la disponibilidad de energía son restrictivas, debido a su diseño compacto y bajo consumo [57].

Centrándonos en la flexibilidad y escalabilidad, estos equipos tienen una gran capacidad de personalización, tanto en hardware como en software. Soportan múltiples estructuras de inteligencia artificial y son compatibles con soluciones de virtualización ligera, favoreciendo despliegues escalables [59].

2.4.2 Limitaciones frente a GPUs y aceleradores especializados

Respecto a la limitación del rendimiento computacional, los NUCs no alcanzan capacidades de cómputo paralelo masivo de las GPUs dedicadas específicamente al entrenamiento e inferencia de modelos complejos de inteligencia artificial. En este sentido, las GPUs siguen siendo la referencia en cargas intensivas de IA [60].

En ausencia de aceleradores adicionales, el rendimiento de los NUCs es insuficiente para aplicaciones que demandan procesamiento en tiempo real sobre modelos de gran tamaño. En cambio, plataformas diseñadas específicamente para edge AI (ej. Nvidia Jetson, Google Coral) superan estas limitaciones gracias a su integración de GPUs embebidas o NPUs dedicadas [61].

2.5 Justificación de la investigación

El crecimiento de la IA generativa está fuertemente relacionado con el avance de las infraestructuras de cómputo en la nube, especialmente a GPUs y aceleradores dedicados. Modelos de IA tales como GPT-3 [8] o Stable Diffusion [13], tienen la necesidad de hacer uso de recursos computacionales a gran escala, estos sobrepasan las capacidades de los dispositivos de propósito general. La situación actual ha permitido avances importantes, pero a su vez tiene limitaciones en lo que se refiere a coste, dependencia de la nube, privacidad de los datos y latencia [62].

Actualmente existe un creciente interés por trasladar la inferencia de modelos generativos a entornos Edge, tales como dispositivos locales con un bajo consumo y un

tamaño reducido, como puede ser el Intel NUC. Tales entornos son especialmente atractivos para aplicaciones que requieran un procesamiento en tiempo real, dentro de estos podemos encontrar visión artificial en robótica o la asistencia conversacional en entornos industriales, también en entornos donde sea especialmente importante el aislamiento de los datos. Sin embargo, en los estudios realizados se destaca la existencia de una brecha importante: una gran parte de las técnicas desarrolladas para la optimización se han diseñado, evaluado y validado siempre teniendo en cuenta el hardware de altas prestaciones, habiendo dejado de lado el impacto que tiene en los dispositivos de prestaciones limitadas [31], [63].

Esta investigación se fundamenta en varios aspectos:

- La necesidad de la optimización en inferencia Edge: hay ciertas técnicas como pueden ser la cuantización, el pruning o la distillation, que han demostrado reducir de manera significativa los requerimientos de memoria y de cómputo [64], [38], pero la efectividad es un parámetro que varía según la arquitectura empleada y el hardware de ejecución. Evaluar el rendimiento que tiene en un Intel NUC nos proporciona una información práctica importante que todavía es escasa en los estudios realizados.
- La contribución a la eficiencia energética: la utilización de la nube para desplegar modelos trae consigo un gran consumo energético, concentrado en grandes centros de datos [65]. La ejecución de los modelos en dispositivos compactos puede ser una alternativa más sostenible, siempre y cuando se apliquen las optimizaciones adecuadas para conseguirlo.
- El impacto económico y de accesibilidad: reducir la necesidad de utilizar grandes infraestructuras de un alto coste facilita el acceso a la IA generativa, permitiendo que pequeñas empresas, centros educativos y zonas sin acceso a una conexión con la nube puedan hacer uso de ella.
- La brecha en la investigación aplicada: a pesar de la existencia de frameworks para la optimización, tales como Intel OpenVINO o ONNX Runtime, la mayoría de los estudios comparativos se centran en los

benchmarks sintéticos o en los modelos discriminativos. En este TFG se busca llenar ese espacio olvidado aplicando dichas herramientas a los modelos generativos, midiendo el impacto que tienen en la latencia, el consumo de memoria y la viabilidad práctica que tienen en un dispositivo Edge real.

En este sentido, esta investigación lo que busca es aportar datos empíricos al campo de la optimización de la IA generativa en hardware con recursos limitados. Estudios como los de Khan et al. [4] y Chen et al. [5] destacan la importancia y la urgencia de abordar la eficiencia energética en los LLMs, ya sea a través del uso de técnicas de optimización algorítmica o mediante la selección del correcto hardware especializado. Sin embargo, la mayoría de estas investigaciones se ha llevado a cabo en entornos de cómputo con altas prestaciones, dejando de lado el análisis de los dispositivos tipo Edge. Realizando evaluaciones de distintas técnicas y herramientas haciendo uso de un Intel NUC, en este TFG se pretende identificar qué combinaciones resultan más eficaces y, a su vez, ofrecer un punto de partida para futuros desarrollos en este campo que busquen un equilibrio entre precisión, eficiencia y aplicabilidad real.

Capítulo 3 - Herramientas

3.1 Ecosistema de software para IA generativa

El ecosistema contemporáneo de IA generativa incorpora una variedad de herramientas y frameworks que permiten completar el ciclo total de entrenamiento, optimización y exportación de modelos.

3.1.1 Hugging Face

Una plataforma que reúne modelos, datasets y librerías abiertas, destacando particularmente Transformers, la cual proporciona un acceso integral a arquitecturas preentrenadas como T5, BERT o GPT.

3.1.2 PyTorch

Framework de trabajo primordial para la investigación y el entrenamiento, escogido por su capacidad de integración con Transformers y su flexibilidad.

3.1.3 ONNX

Formato estándar que facilita la interoperabilidad entre frameworks, y que incluye ONNX Runtime, un acelerador que permite ejecuciones optimizadas en CPUs y otros tipos de unidades de procesamiento.

3.1.4 Intel OpenVINO

Conjunto de herramientas especializado en optimización para hardware Intel, que usa métodos como fusión de capas o cuantización para aumentar el rendimiento en entornos Edge.

3.2 Entorno de experimentación

3.2.1 Hardware utilizado

El escenario previsto para el despliegue se basa en un Intel NUC con procesador Intel® Core™ Ultra 9 185H (CPU de última generación con 16 núcleos, 6 de ellos

performance cores, 8 de ellos efficient cores y 2 de ellos low power efficient cores, 22 hilos, y una frecuencia turbo máxima de 5.1GHz, con 24MB de caché), una GPU integrada Intel Arc y una NPU integrada Intel AI Boost específica para la aceleración de IA. Así se representa un entorno Edge realista, fácil de replicar y con la ventaja de tener tres tipos de aceleradores integrados en la misma máquina.

3.2.2 Software utilizado

El sistema operativo empleado fue Ubuntu 24.04.3 LTS con Python 3.12.3. También se utilizaron las siguientes librerías y frameworks:

- PyTorch (2.8) y Transformers (≥4.53.1) para cargar y generar texto, implementar técnicas de cuantización y establecer una línea base de rendimiento sobre CPU frente a exportaciones e inferencias optimizadas.
- ONNX (≥1.19) y ONNX Runtime (≥1.22.1) para poder ejecutar y exportar los modelos en grafo intermedio.
- OpenVINO Runtime (≥2024.6.0) para conseguir optimizaciones específicas sobre hardware Intel.
- Para gestión de entornos y dependencias, pip (≥24.0) y venv.
- Y como librerías auxiliares: psutil (7.0) para perfilar la memoria y la CPU, y pandas (2.3.2) y matplotlib (3.10.5) para el análisis y la visualización.
- Para medición de consumo, paquete turbostat.

3.3 Modelos seleccionados

3.3.1 Modelos de texto

El principal modelo empleado fue DistilGPT-2 (≈82M parámetros), que es una versión reducida del famoso GPT-2 y optimizada por Hugging Face, el cual es idóneo para ejecutar en CPUs y un buen representante de tareas de lenguaje generativo. A su vez, Tiny-GPT-2, el cual es una variante aún más simplificada (≈2M parámetros), fue utilizado en fases de validación, ya que es útil como control rápido en exportaciones y pruebas preliminares.

3.3.2 Modelos de imagen

El modelo empleado fue Stable Diffusion 1.5, arquitectura que genera imágenes basada en difusión latente, y la cual representa un estándar en este tipo de tareas y permite evaluar el impacto de las optimizaciones en un escenario de mayor complejidad que el lenguaje. A su vez, también se utilizó Stable Diffusion Turbo, una variante optimizada para inferencias rápidas, que resultó especialmente útil en fases de validación y pruebas comparativas de rendimiento.

3.3.3 Modelos de audio (planificados)

El modelo considerado fue Whisper Small, que es una versión más pequeña del sistema de reconocimiento y transcripción de voz desarrollado por OpenAI. Este modelo es un referente en tareas de audio y hubiera permitido analizar la viabilidad de las mejoras en un ámbito diferente al de texto e imagen. También se consideró el empleo de Whisper Tiny, una versión más liviana (≈39M parámetros), que habría sido beneficiosa como modelo de validación inicial y control rápido para las etapas de exportación. No obstante, no fue posible incorporar estos modelos a los experimentos debido a restricciones técnicas relacionadas con el soporte del entorno y los recursos disponibles.

Capítulo 4 - Metodología

En esta sección se va a describir el procedimiento seguido para analizar las técnicas de optimización aplicadas a los modelos generativos escogidos. Se va a abordar la optimización de la inferencia por dos vías complementarias: la compresión por destilación y la optimización del runtime. La siguiente figura resume el flujo común, a grandes rasgos, de optimización e inferencia:

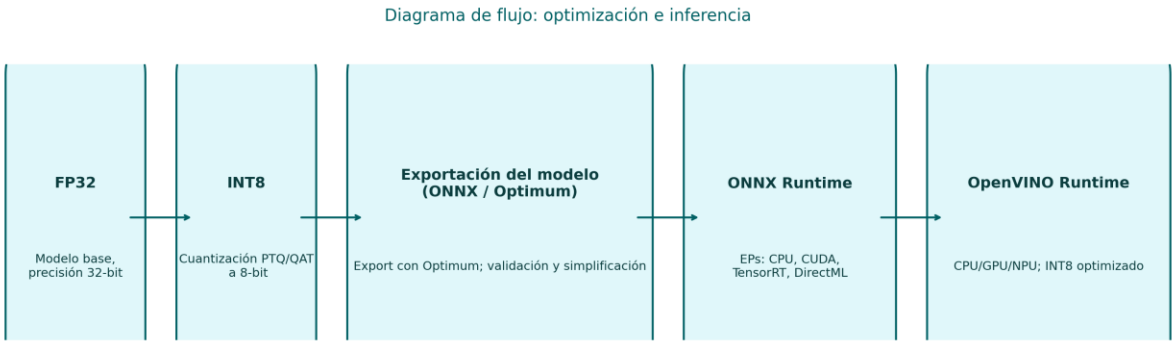

Figura 4-1. Diagrama de flujo de la metodología a seguir

Primero, como referencia, ejecutaremos pruebas en precisión simple FP32. En segundo lugar, aplicaremos cuantización dinámica INT8 post-training, que no necesita de reentreno: esto cuantiza los pesos de las capas lineales a 8 bits y cuantiza y decuantiza activaciones al vuelo, reduciendo así memoria y haciendo más baratas las operaciones en CPU. Como tercer paso, se lleva a cabo la exportación del modelo. En texto buscaremos exportar el modelo DistilGPT-2 a ONNX (que es un grafo intermedio portable que sirve de entrada para ambos runtimes) de forma manual y con ejes dinámicos. En imagen exportaremos el modelo Stable Diffusion con Hugging Face Optimum (empaqueta CLIP/UNet/VAE), tanto a ONNX para ONNX Runtime como a IR (Intermediate Representation, formato intermedio de modelo que usa OpenVINO para compilar y ejecutar redes) para OpenVINO. En cuarto lugar, ejecutaremos el grafo anteriormente exportado en ONNX Runtime, que es un motor de inferencia para ejecutar modelos en formato ONNX de forma rápida y portable en multitud de plataformas. Y como último paso, ejecutaremos el modelo ONNX/IR previamente exportado con OpenVINO Runtime, un runtime de Intel con optimizaciones que permite

la selección de device, el cual compila el grafo para CPU/GPU/NPU, elige kernels del plugin del dispositivo, y hace las transformaciones adicionales necesarias.

Tanto en FP32 como en INT8 y ONNX Runtime, se escoge CPU como acelerador, y en OpenVINO Runtime se ponen a prueba los tres aceleradores (CPU/GPU/NPU) si es posible.

4.1 Preparación del entorno

Crear un entorno virtual e instalar lo siguiente:

- `pip install torch transformers torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu`
- `pip install --upgrade-strategy eager "optimum[onnxruntime]" "optimum[openvino]" onnx onnxsim onnxruntime openvino diffusers transformers accelerate safetensors sentencepiece psutil numpy pillow`

Por otra parte, se comparte un repositorio de GitHub[1] donde se incluirán los distintos scripts y lógica de archivos del proyecto. A su vez, todos los scripts cuentan con cambio de parámetros ejemplificado en un archivo de texto en el repositorio, para distintos tipos de ejecuciones.

[1] https://github.com/pgit99/TFG-Optimizaci-n-de-modelos-de-IA-generativa-en-Hardware-de-ltima-generaci-n

4.2 Métricas registradas

Categoría	Campo (CSV)	Descripción	Unidades
Tiempos de inferencia	elapsed_s	Tiempo total del *run* (excluye el *warm-up*).	s
Tiempos de inferencia	ms_per_token	Tiempo medio por token nuevo. En ORT/OV se normaliza usando tokens-per-run.	ms/token
Tiempos de inferencia	throughput_tokens_s	Rendimiento medio en tokens por segundo.	tokens/s
Recursos del sistema	cpu_proc_percent	Uso de CPU del proceso de inferencia	%
Recursos del sistema	cpu_sys_percent	Uso de CPU total del sistema durante la ejecución	%
Recursos del sistema	mem_mb	Memoria consumida por el proceso	MB
Energía / térmica	avg_mhz	Frecuencia media del paquete de CPU	MHz
Energía / térmica	pkg_temp_c	Temperatura del paquete de CPU	°C
Energía / térmica	pkg_watt	Potencia media del paquete de CPU	W
Energía / térmica	energy_j	Energía estimada consumida durante la inferencia	J

Tabla 4-1. Métricas DistilGPT-2

Categoría	Campo (CSV)	Descripción	Unidades
Tiempos de inferencia	total_s	Tiempo total de la generación por imagen	s
Tiempos de inferencia	overall_throughput_img_s	Throughput efectivo del run completo	imágenes/s
Recursos del sistema	cpu_proc_percent	Uso de CPU del proceso de inferencia	%
Recursos del sistema	cpu_sys_percent	Uso de CPU total del sistema durante la ejecución	%
Recursos del sistema	mem_mb	Memoria consumida por el proceso	MB
Energía / térmica	avg_mhz	Frecuencia media del paquete de CPU	MHz
Energía / térmica	pkg_temp_c	Temperatura del paquete de CPU	°C
Energía / térmica	pkg_watt	Potencia media del paquete de CPU	W
Energía / térmica	energy_j	Energía estimada consumida durante la inferencia	J

Tabla 4-2. Métricas Stable Diffusion 1.5 / Turbo

4.3 DistilGPT-2

Para asegurar consistencia, en todos los casos usamos el mismo prompt en inglés ("Hey, this is a benchmark"), ya que DistilGPT-2 está entrenado principalmente en ese idioma, fijamos el límite máximo de tokens nuevos a 300, el tamaño del batch a 1, los hilos de CPU a utilizar a 16 y el número de ejecuciones a 20. El dispositivo seleccionado por defecto será CPU, y en la fase de OpenVINO, compararemos su rendimiento con GPU y NPU.

4.3.1 PyTorch FP32 (referencia)

En primer lugar, se descargan tanto el modelo como el tokenizer desde Hugging Face y se cargan con precisión simple. Se fijan los hilos para CPU (vía `OMP_NUM_THREADS`, `MKL_NUM_THREADS` y `torch.set_num_threads`) y a continuación se realiza el warm-up, se fija `torch.manual_seed(0)` por run, se mide generación greedy (lo que se vuelca al CSV) y se añade un segmento extra "bonito" sample a un archivo de texto que almacena las generaciones (no afecta a las métricas). Estas mediciones funcionan aplicando una generación autoregresiva (`with torch.inference_mode(): x = model.generate(…)`) y luego calculando los tokens realmente generados al comparar longitudes de salida con longitudes de entrada.

Ejecución: `python3 scripts/benchmark_pytorch_fp32.py --turbostat`

4.3.2 PyTorch INT8 (cuantización dinámica)

Partiendo del flujo anterior, se aplica cuantización dinámica sobre las capas lineares del decoder GPT-2 (proyecciones de atención y MLP por bloque, y `lm_head`) mediante `torch.quantization.quantize_dynamic(base, {torch.nn.Linear}, dtype = torch.qint8)` y se mantiene el resto del flujo de medición expuesto en el apartado anterior.

Ejecución: `python3 scripts/benchmark_pytorch_int8.py --turbostat`

4.3.3 Exportación previa a ONNX

Antes de ejecutar los benchmarks en ONNX Runtime y OpenVINO, se exporta el modelo a ONNX (que es la entrada común para ONNX Runtime y OpenVINO) con un wrapper que expone logits (`forward(…): return logits`), se desactiva `use_cache` y se fuerza atención "eager" para coherencia del grafo. El export se hace mediante `torch.onnx.export`, siendo por defecto opset 17, y ejes dinámicos en `batch` y `sequence`. Se añaden metadatos útiles (opset, prompt usado para el dummy, destino previsto cpu/xpu/npu, etc.) al ONNX y se intenta simplificar el grafo si es posible.

Ejecución: `python3 scripts/export_onnx.py --turbostat`

4.3.4 ONNX Runtime

En esta fase, se carga el ONNX exportado en el paso anterior y se crea una sesión con el execution provider elegido (que a efectos prácticos funcionaría como la selección de device, por defecto en nuestra experimentación es CPU) y se fija el paralelismo intra/inter-op (con `if n_threads: ...`). Este benchmark, al contrario que los anteriores, no realiza generación autoregresiva, sino que ejecuta una inferencia del grafo por run sobre tensores sintéticos (o desde `input-npy`) y utiliza `tokens-per-run` para normalizar a ms/token y tokens/s.

Ejecución: `python3 scripts/benchmark_onnx.py onnx/model.onnx --threads 16 --turbostat perrun`

4.3.5 OpenVINO Runtime (desde ONNX)

En esta última fase, se lee el ONNX ya exportado previamente con OpenVINO, y antes de compilar, se fijan shapes dinámicos, tanto para las entradas de rango 2 (texto, `seq-len=128` por defecto) como para las entradas de rango 4 (imagen, `height=224` y `width=224` por defecto). Tras el `reshape()`, se compila el modelo en función del dispositivo elegido (AUTO/CPU/GPU/NPU/MULTI) y se crea una petición de inferencia (`compiled.create_infer_request()`), que como en el paso previo, funciona con tensores sintéticos (o `input-npy`) y se usa `tokens-per-run` solo para escalar las métricas por token.

Ejecución: `python3 scripts/benchmark_openvino_from_onnx.py onnx/model.onnx --device CPU --turbostat perrun`

4.4 Stable Diffusion (1.5 / Turbo)

Para asegurar consistencia, en todos los casos usamos un conjunto fijo de cinco prompts ("a picture of Ash Ketchum and his Pokemons swimming in the beach", "ultra-detailed photo of a siamese cat, dramatic lighting", "streets of Seoul at night, pixel art", "a super mario bros style castle on a floating island, volumetric fog", "minimal product render of some juicy watermelon chopped on shiny marble") y semillas de 42 a 46, con resolución 512x512. También fijamos Euler como scheduler, con 8 pasos de denoising (2

si estamos en Turbo) y fijamos CFG a 7.5 (0 si estamos en Turbo, por la naturaleza del propio modelo).

4.4.1 PyTorch FP32 (referencia)

En primer lugar, se descarga automáticamente el modelo desde Hugging Face y se carga en el pipeline base de StableDiffusion, en CPU y con precisión simple (`pipe = StableDiffusionPipeline.from_pretrained(args.model,torch_dtype=torch.float32)`). La inferencia es desglosada explícitamente en tres etapas: primero la codificación de texto (tokenización y text_encoder), después el bucle de denoising en UNet (con o sin classifier-free guidance), y por último decodificación VAE a imagen. Se puede ver implementada más a fondo en la función `run_sd_one_image_components(...)`, donde se miden los tiempos por bloque y también se iteran los timesteps del scheduler.

Ejecución: `python3 scripts/01_sd_pytorch_fp32.py --model runwayml/stable-diffusion-v1-5 --guidance 7.5 --turbostat`

4.4.2 PyTorch INT8 (cuantización dinámica)

A partir de lo realizado para FP32, se aplica cuantización dinámica a las capas lineares tanto del Text Encoder como de la UNet con `quantize_dynamic(pipe.text_encoder/unet,{nn.Linear}, dtype=torch.qint8)` y nos aseguramos de que los módulos estén en CPU antes de cuantizar (con el motor de cuantización x86, `torch.backends.quantized.engine = "fbgemm"`, y con el envío explícito de los mismos, `pipe.text_encoder/unet = pipe.text_encoder/unet.to("cpu")`). El resto del flujo anteriormente descrito (tokenización, UNet, VAE) y la toma de tiempos se mantiene igual, facilitando su comparación con FP32.

Ejecución: `python3 scripts/02_sd_pytorch_int8_linear.py --model runwayml/stable-diffusion-v1-5 --guidance 7.5 --turbostat`

4.4.3 Exportación previa con Optimum-CLI

Antes de ejecutar los benchmarks en ONNX Runtime y OpenVINO, los modelos se exportaron a ONNX (formato neutral e intercambiable) e IR (Intermediate Representation, el formato optimizado de OpenVINO, compuesto por un grafo y pesos) con la CLI de Optimum (Hugging Face Optimum, el toolkit de aceleración y optimización de Hugging Face) para reducir el overhead en tiempo de ejecución (menos sobrecarga de arranque) y fijar optimizaciones y compilación específica del dispositivo (mejor rendimiento). Esto también presenta una mejora en la trazabilidad y la reproducibilidad de la experimentación, ya que el mismo ONNX/IR y la misma configuración da pie a resultados consistentes entre distintas máquinas.

Ejecución ONNX: `optimum-cli export onnx --model runwayml/stable-diffusion-v1-5 --task stable-diffusion sd15_onnx`

Ejecución OpenVINO: `optimum-cli export openvino --model runwayml/stable-diffusion-v1-5 --task stable-diffusion --weight-format fp32 sd15_openvino_fp32`

4.4.4 ONNX Runtime (Optimum-Intel)

En este paso se trabajará con el modelo exportado a ONNX previamente y se crea el pipeline con `ORTStableDiffusionPipeline.from_pretrained(...)`. Se permite elegir el execution provider (CPU/DML/CUDA, aunque en nuestra experimentación se fijará CPU), que es el backend que ejecuta el grafo, y se fija el paralelismo intra-operador `(so.intra_op_num_threads = int(args.threads))`. A su vez, se crea una llamada unificada que también será usada por OpenVINO (en PyTorch se replicó la misma parametrización, pero con un recorrido por componentes en vez de una sola pipe). Soporta `export=needs_export` por si se desea hacer exportación del modelo al comienzo del script.

Ejecución: `python3 scripts/03_sd_onnxruntime.py --model sd15_onnx --guidance-scale 7.5 --threads 16 --turbostat`

4.4.5 OpenVINO Runtime (Optimum-Intel)

En este paso se trabajará con el IR exportado previamente y se crea el pipeline con `OVStableDiffusionPipeline.from_pretrained(...)`, configurando nuestra ejecución (mediante `ov_config`) con precision hints (FP32/FP16/BF16), model caching, performance hints (latency/throughput) y streams, e hilos. Aquí también se implementa la selección de device (CPU/GPU/NPU) y lo utilizaremos para comparar su rendimiento. El grafo se fija a un shape estático (`reshape(…)`) y se compila (`compile()`) para así maximizar la eficiencia. Como en el paso anterior, se crea una llamada unificada y permite `export=needs_export`.

Ejecución: `python3 scripts/04_sd_openvino.py --model sd15_openvino_fp32 --device CPU --ov-precision-hint f32 --perf-hint latency --num-streams 1 --guidance-scale 7.5 --ov-cache-dir ov_model_cache_fp32 --threads 16 --turbostat`

Capítulo 5 - Resultados

Todos los resultados que se van a ver a continuación han sido obtenidos mediante la ejecución de los scripts con turbostat, tras comprobar que añade a los resultados un overhead prácticamente nulo y aporta a su vez datos de consumo muy interesantes para nuestro propósito. Cabe destacar que turbostat refleja únicamente las métricas de energía del CPU package, no incluye GPU ni NPU. Vamos a analizar estos resultados obtenidos a continuación.

5.1 DistilGPT-2

Como nota previa al análisis, PyTorch y ONNX son autoregresivos (token-a-token) y OpenVINO se midió con un flujo normalizado por grafo, así que los resultados no son comparables 1:1.

Métrica	FP32	INT8	ONNX
elapsed_s	2.644	2.068	0.085
ms_per_token	8.815	6.893	85.273
throughput_tokens_s	113.5	145.14	11.78
cpu_proc_percent	798.6	700.1	816.3
cpu_sys_percent	36.6	32.1	38.0
mem_mb	1119.1	1169.6	875.0
avg_mhz	1497.0	1363.0	1108.0
pkg_temp_c	79.9	81.3	47.7
pkg_watt	50.58	50.6	25.69
energy_j	133.69	104.58	2.2

Latency — median (ms/token)	8.879	6.856	86.542
Latency — mean	8.815	6.893	85.273
Latency — p95	9.027	7.141	90.936
Throughput — median	112.63	145.85	11.56
Throughput — mean	113.5	145.14	11.78
Throughput — p95	118.11	148.84	13.14
Power — median	50.62	50.88	26.66
Power — mean	50.58	50.6	25.69
Power — p95	54.33	54.98	35.04

Tabla 5-1. Comparativa FP32 CPU vs INT8 CPU vs ONNX CPU

Para facilitar la lectura de estos resultados, veremos las métricas más importantes en las siguientes tablas: latencia en ms/token (el tiempo medio para generar un token nuevo), throughput en tokens/s (el ritmo de generación) y la potencia media de esta generación en W.

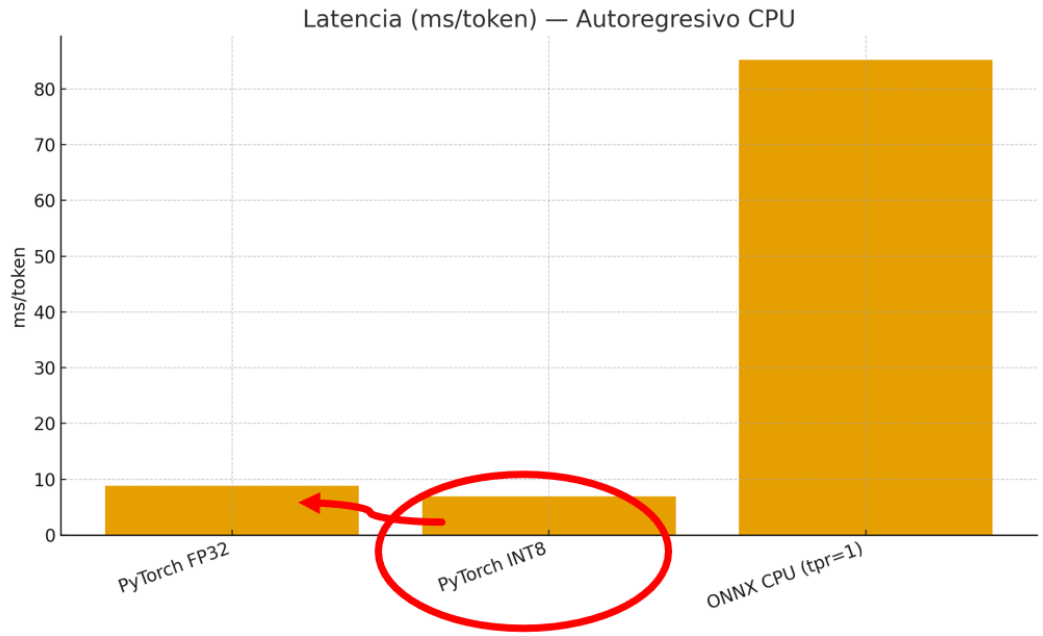

Figura 5-1. Latencia FP32 CPU vs INT8 CPU vs ONNX CPU

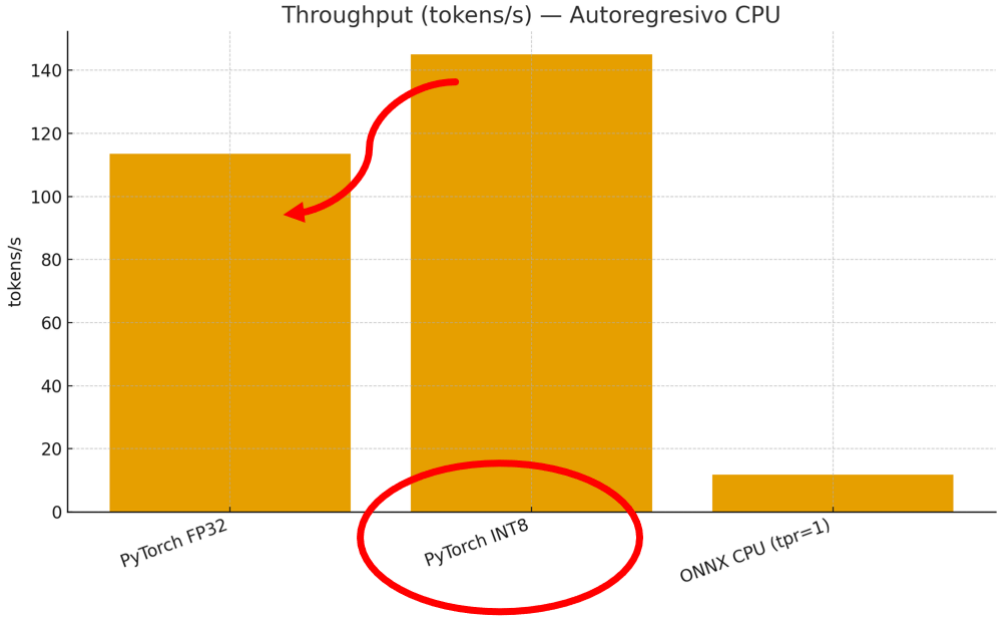

Figura 5-2. Throughput FP32 CPU vs INT8 CPU vs ONNX CPU

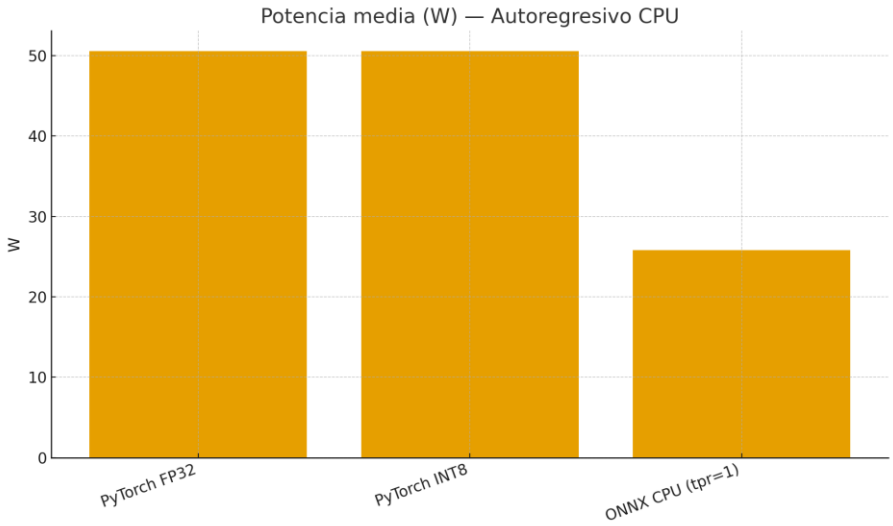

Figura 5-3. Potencia FP32 CPU vs INT8 CPU vs ONNX CPU

Con estos resultados se puede observar que INT8 reduce los milisegundos/token y aumenta los tokens/segundo gracias a su aceleración del camino crítico del decodificado, obteniendo así unas mejoras de un 22% en latencia y de un 28% en throughput frente a FP32. Sin embargo, la potencia no aumenta, así que también mejora la eficiencia energética. FP32 mantiene una buena estabilidad de latencia, pero rinde menos y consume ligeramente más energía que INT8. Sin embargo, ONNX no amortiza bien el coste por token en este caso y pierde en todo, ya que ejecuta un grafo sin cuantización ni fusiones especializadas en FP32 con el bucle fuera del grafo y arrastrando overhead por paso, al estar los runs normalizados con tokens_per_run = 1.

Métrica	OpenVINO CPU	OpenVINO GPU	OpenVINO NPU
elapsed_s	0.039	0.014	0.039
ms_per_token	0.13	0.047	0.129
throughput_tokens_s	7753.23	21560.52	7771.27
cpu_proc_percent	119.1	4.4	2.8
cpu_sys_percent	6.6	1.3	1.3

mem_mb	936.9	952.8	828.2
avg_mhz	16.0	4.0	6.0
pkg_temp_c	44.1	44.8	41.3
pkg_watt	4.48	3.36	4.58
energy_j	0.17	0.05	0.18
tokens_generated	300.0	300.0	300.0
Latency — median	0.124	0.047	0.128
Latency — mean	0.13	0.047	0.129
Latency — p95	0.145	0.051	0.132
Throughput — median	8089.27	21212.58	7790.67
Throughput — mean	7753.23	21560.52	7771.27
Throughput — p95	8251.96	24637.34	7930.59
Power — median	3.69	3.38	4.19
Power — mean	4.48	3.36	4.58
Power — p95	8.31	3.74	5.8

Tabla 5-2. Comparativa OpenVINO CPU vs GPU vs NPU

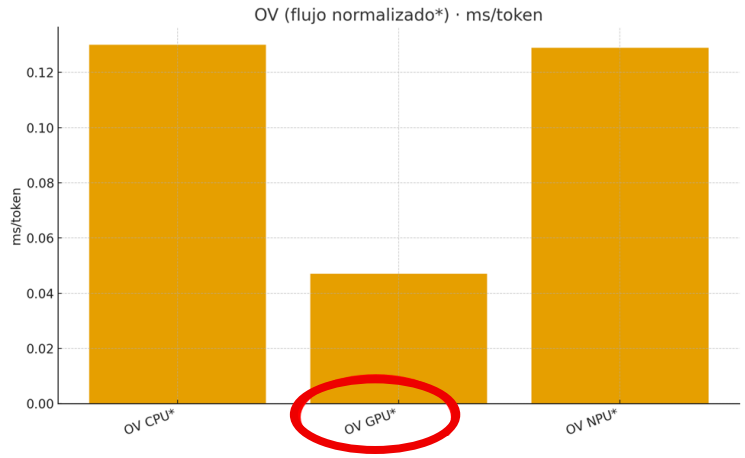

Figura 5-4. Latencia OpenVINO CPU/GPU/NPU

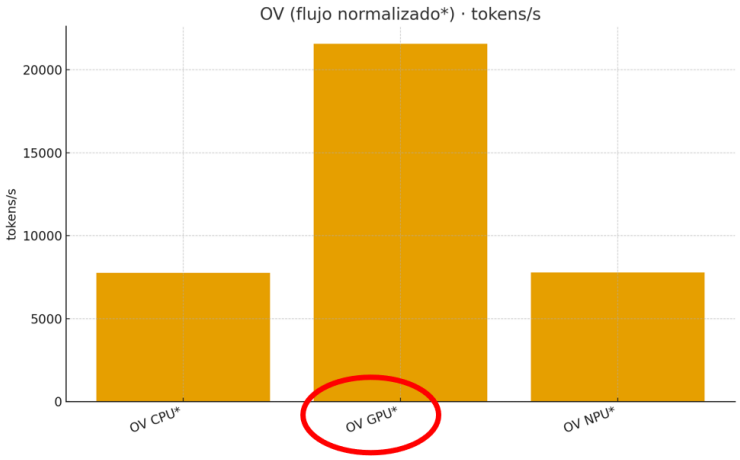

Figura 5-5. Throughput OpenVINO CPU/GPU/NPU

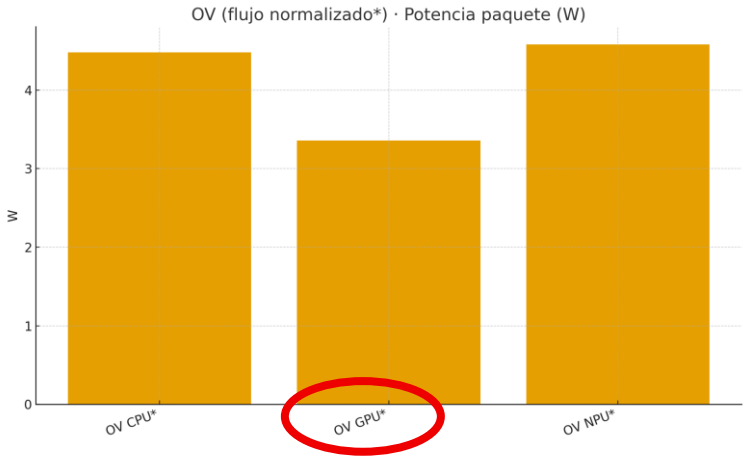

Figura 5-6. Potencia OpenVINO CPU/GPU/NPU

En OpenVINO, usamos un flujo normalizado por grafo, debido a errores al intentar llevar a cabo esa autorregresión vista anteriormente, por eso el ms/token en GPU sale muy bajo, pero no representa el decode real token a token.

Sin embargo, nos sirve para poder comparar las diferencias de rendimiento entre los distintos devices: GPU lidera y CPU y NPU quedan parejas, pero muy por detrás.

No observamos una mejora en NPU como se esperaría porque en el decode token a token trabajamos con micro lotes, y la NPU del NUC está pensada para throughput sostenido y baja potencia, no para latencia pico. Además, hay fallbacks a CPU que añaden copias y por consecuencia algo de latencia. También penaliza la ruta desde ONNX, dado que el export y el grafo, al ser ONNX y no IR, aún no tiene todas las fusiones optimizadas al máximo.

Con los resultados obtenidos, podemos determinar que la cuantización dinámica INT8 es la opción más rápida y eficiente, pudiendo quizás ser mejorada con una buena optimización del modelo para OpenVINO Runtime.

5.2 Stable Diffusion (1.5 / Turbo)

Como ya fue mencionado antes, para la realización de estas pruebas comparativas se eligió como scheduler Euler, ya que ofrece un punto medio entre calidad y velocidad con resultados consistentes.

Métrica	1.5	Turbo
total_s	159.883	41.5764
overall_throughput_img_s	0.0312729	0.120261
energy_per_image_j	1276.94	377.17
energy_j	6384.72	1885.85
pkg_watt	39.926	45.238
cpu_proc_percent	46.6464	44.81

cpu_sys_percent	46.82	44.94
mem_mb	4891.53	5727.27
avg_mhz	981.2	1105.6
pkg_temp_c	90.2	87.8
latency_total_ms_median	32235.8	8289.37
throughput_img_s_median	0.0310214	0.120636
pkg_watt_median	41.1	46.26
latency_total_ms_mean	31976.5	8315.27
throughput_img_s_mean	0.0312799	0.120305
pkg_watt_mean	39.926	45.238
latency_total_ms_p95	32436.1	8521.66
throughput_img_s_p95	0.031929	0.123344
pkg_watt_p95	48.268	53.498

Tabla 5-3. Comparativa FP32 CPU: SD 1.5 vs SD Turbo

Los resultados base obtenidos, sin aplicar ningún tipo de optimización, indican que SD Turbo es aproximadamente 3.9 veces más rápido que SD 1.5, ya que está diseñado para, mediante técnicas de destilación, mantener una calidad razonable con menos pasos, lo que reduce drásticamente el tiempo de UNet y por extensión la latencia total. El uso de CPU es algo mayor en Turbo porque se concentra más trabajo en menos tiempo, pero el tiempo por imagen es mucho menor, así que tenemos mejor eficiencia energética (aproximadamente 4 veces más eficiente por imagen), y el consumo de RAM baja, probablemente por tener menos buffers retenidos al reducir los pasos totales. SD Turbo es sin duda la opción dominante para rendimiento y eficiencia en CPU, pero tiene un coste cualitativo sobre la calidad de las imágenes generadas.

Métrica	1.5	Turbo
total_s	143.917	37.9192
overall_throughput_img_s	0.0347422	0.131859
energy_per_image_j	1529.16	413.553
energy_j	7645.81	2067.77
pkg_watt	53.112	54.53
cpu_proc_percent	45.4882	45.5473
cpu_sys_percent	45.62	45.68
mem_mb	5303.53	6363.61
avg_mhz	1388.6	1423
pkg_temp_c	88.2	84.8
latency_total_ms_median	28886.9	7660.74
throughput_img_s_median	0.0346178	0.130536
pkg_watt_median	53.68	53.65
latency_total_ms_mean	28783.5	7583.85
throughput_img_s_mean	0.0347445	0.131897
pkg_watt_mean	53.112	54.53
latency_total_ms_p95	28994.1	7710.81
throughput_img_s_p95	0.0351633	0.134853

pkg_watt_p95	55.768	58.342

Tabla 5-4. Comparativa INT8 CPU: SD 1.5 vs SD Turbo

Métrica (mejora x)	1.5 gain x	Turbo gain x	1.5 INT8	1.5 FP32	Turbo INT8	Turbo FP 32
Speedup latencia mediana (x)	1.11593	1.08206	28886.9	32235.8	7660.74	8289.37
Ganancia throughput medio (x)	1.11076	1.09636	0.0347445	0.0312799	0.131897	0.120305
Mejora energía/imagen (x)	0.835061	0.912023	1529.16	1276.94	413.553	377.17
Reducción memoria (x)	0.922315	0.900003	5303.53	4891.53	6363.61	5727.27
Reducción uso CPU proceso (x)	1.02546	0.983813	45.4882	46.6464	45.5473	44.81
Reducción potencia media paquete (x)	0.751732	0.829598	53.112	39.926	54.53	45.238
Reducción temperatura media (x)	1.02268	1.03538	88.2	90.2	84.8	87.8

Tabla 5-5. INT8 vs FP32 - Mejoras (X>1 mejora)

Al aplicar esta segunda fase podemos observar que INT8 da ligeras mejoras de latencia y de throughput, pero no mejora, e incluso a veces empeora, la eficiencia

energética por imagen. Esto se debe a que el INT8 de PyTorch (que es exclusivo de CPU) suele explotar rutas vectoriales más agresivas, lo que da lugar a un mayor trabajo por ciclo y a una mayor potencia instantánea. Si la latencia no baja lo suficiente (lo que se da en nuestro caso por las capas no lineares que no pudieron ser cuantizadas), el aumento de memoria compensa negativamente y acaba con más energía consumida por imagen.

Métrica	1.5	Turbo
total_s	988.905	328.1
overall_throughput_img_s	0.0252805	0.0761962
energy_per_image_j	2502.11	844.456
energy_j	62552.7	21111.4
pkg_watt	63.2636	64.3696
cpu_proc_percent	71.036	69.4467
cpu_sys_percent	71.092	69.496
mem_mb	11804.7	11713.9
avg_mhz	2213.44	2246.6
pkg_temp_c	99.08	99
latency_total_ms_median	39390.5	13067.8
throughput_img_s_median	0.0253869	0.0765239
pkg_watt_median	63	64.23
latency_total_ms_mean	39556.2	13124

throughput_img_s_mean	0.0252853	0.0762241
pkg_watt_mean	63.2636	64.3696
latency_total_ms_p95	40377.9	13496.6
throughput_img_s_p95	0.0257363	0.0779255
pkg_watt_p95	65.678	71.672

Tabla 5-6. Comparativa ONNX Runtime CPU: SD 1.5 vs SD Turbo

Métrica (mejora x)	**1.5 gain x**	**Turbo gain x**	**1.5 ONNX**	**1.5 INT8**	**Turbo ONNX**	**Turbo INT8**
Speedup latencia mediana (x)	0.733347	0.58623	39390.5	28886.9	13067.8	7660.74
Ganancia throughput medio (x)	0.727751	0.577905	0.0252853	0.0347445	0.0762241	0.131897
Mejora energía/imagen (x)	0.61115	0.489727	2502.11	1529.16	844.456	413.553
Reducción memoria (x)	0.449273	0.543254	11804.7	5303.53	11713.9	6363.61
Reducción uso CPU proceso (x)	0.640354	0.655859	71.036	45.4882	69.4467	45.5473
Reducción potencia media paquete (x)	0.839535	0.847139	63.2636	53.112	64.3696	54.53

Reducción temperatura media (x)	0.89019	0.856566	99.08	88.2	99	84.8

Tabla 5-7. ONNX vs INT8 - Mejoras (X>1 mejora)

Se puede observar que, en este paso, ONNX no aplica ninguna mejora sobre el anterior modelo con cuantización dinámica, al revés, lo empeora en todo. Esto se debe a que estos ONNX se están ejecutando en precisión FP32 sobre CPU con ninguna cuantización o fusión especializada, por lo que solo se observa el incremento de cómputo y de ancho de banda que ONNX conlleva sin ninguna mejora añadida.

Métrica	1.5 CPU	1.5 GPU	Turbo CPU	Turbo GPU
total_s	779.321	124.636	267.895	33.6302
overall_throughput_img_s	0.0320792	0.200585	0.0933202	0.743379
energy_per_image_j	1999.34	290.353	672.496	50.8037
energy_j	49983.5	7258.82	16812.4	1270.09
pkg_watt	64.1564	58.2372	62.7952	37.7676
cpu_proc_percent	42.3364	4.042	43.1567	0.292
cpu_sys_percent	42.472	4.064	43.288	0.332
mem_mb	15450.1	4540.36	16797.9	3768.56
avg_mhz	1555.48	1321.76	1526.92	4.96
pkg_temp_c	94.64	88.04	94.92	59.8
latency_total_ms_median	31415.8	4984.94	10776.4	1346.71
throughput_img_s_median	0.0318311	0.200604	0.0927958	0.742549

pkg_watt_median	64.4	62.84	62.21	37.65
latency_total_ms_mean	31172.9	4985.42	10715.8	1345.21
throughput_img_s_mean	0.0320885	0.200588	0.093346	0.743405
pkg_watt_mean	64.1564	58.2372	62.7952	37.7676
latency_total_ms_p95	31702.4	5026.11	10903.7	1353.14
throughput_img_s_p95	0.0331458	0.201869	0.0965287	0.752235
pkg_watt_p95	67.396	72.308	68.51	38.272

Tabla 5-8. Comparativa OpenVINO Runtime: SD 1.5 CPU vs SD 1.5 GPU vs SD Turbo CPU vs SD Turbo GPU

Métrica (mejora x)	1.5 gain x	Turbo gain x	1.5 OV GPU	1.5 INT8	Turbo OV GPU	Turbo INT8
Speedup latencia mediana (x)	5.79483	5.68847	4984.94	28886.9	1346.71	7660.74
Ganancia throughput medio (x)	5.77324	5.63624	0.200588	0.0347445	0.743405	0.131897
Mejora energía/imagen (x)	5.26657	8.14022	290.353	1529.16	50.8037	413.553
Reducción memoria (x)	1.16808	1.6886	4540.36	5303.53	3768.56	6363.61
Reducción uso CPU proceso (x)	11.2539	155.984	4.042	45.4882	0.292	45.5473
Reducción potencia media paquete (x)	0.911994	1.44383	58.2372	53.112	37.7676	54.53
Reducción temperatura media (x)	1.00182	1.41806	88.04	88.2	59.8	84.8

Tabla 5-9. OpenVINO GPU vs INT8 - Mejoras (X>1 mejora)

Para facilitar la lectura de estos resultados, veremos las métricas más importantes en las siguientes tablas: el tiempo que tarda la inferencia de una imagen en segundos y la energía consumida en la misma en Julios.

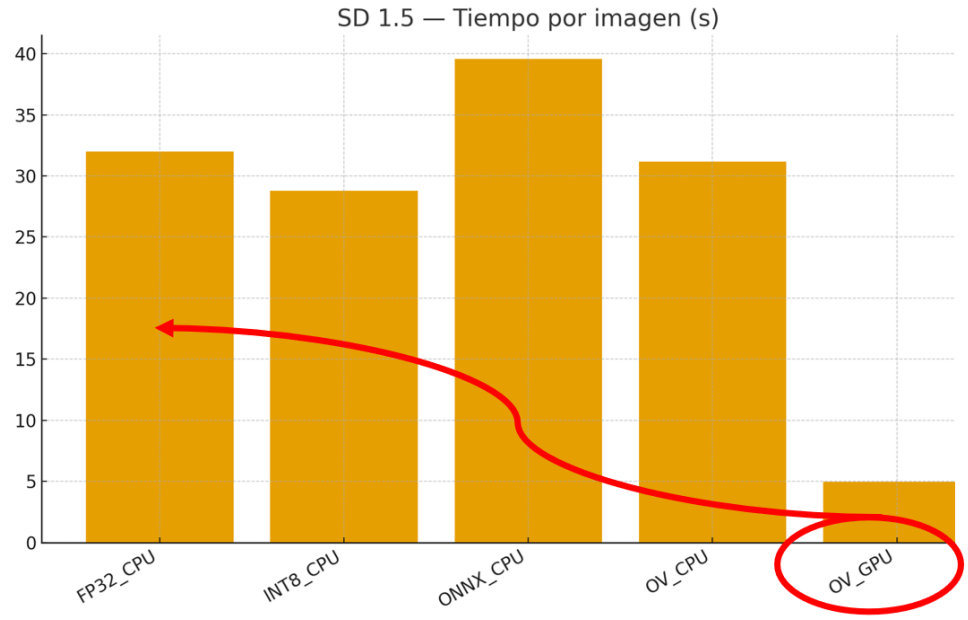

Figura 5-7. Tiempo inferencia SD 1.5

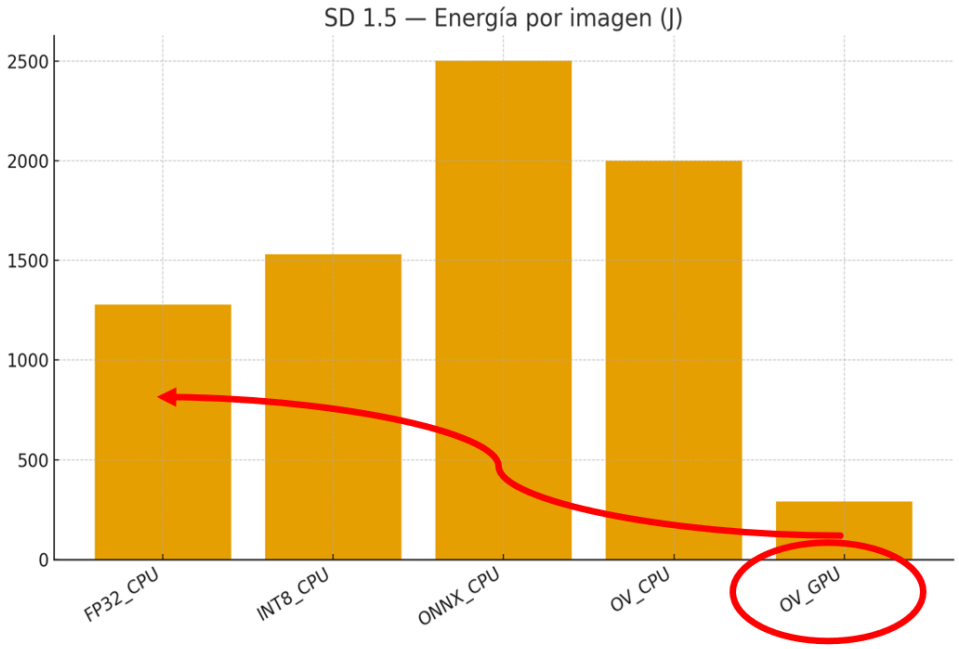

Figura 5-8. Energía inferencia SD 1.5

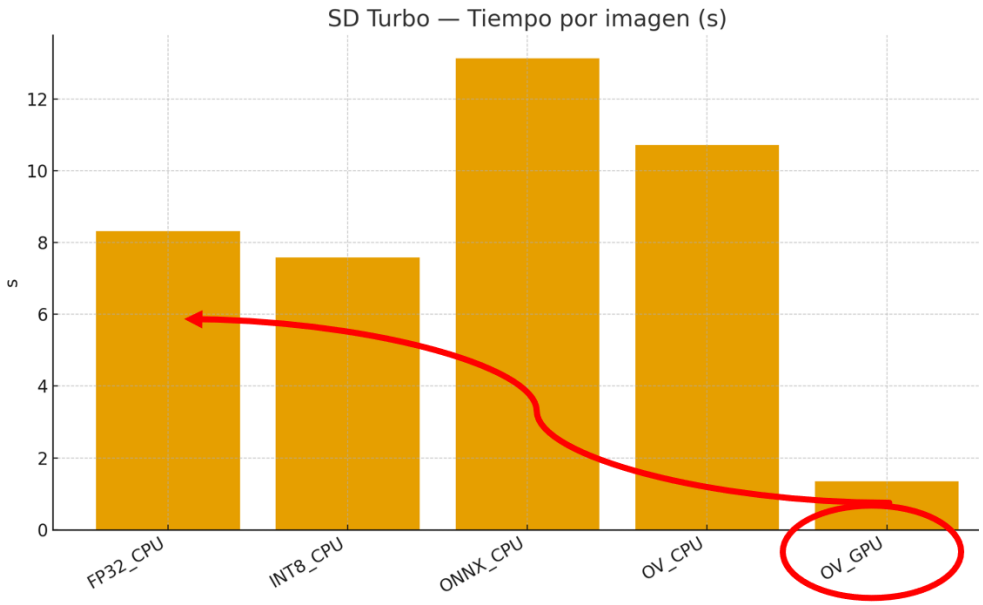

Figura 5-9. Tiempo inferencia SD Turbo

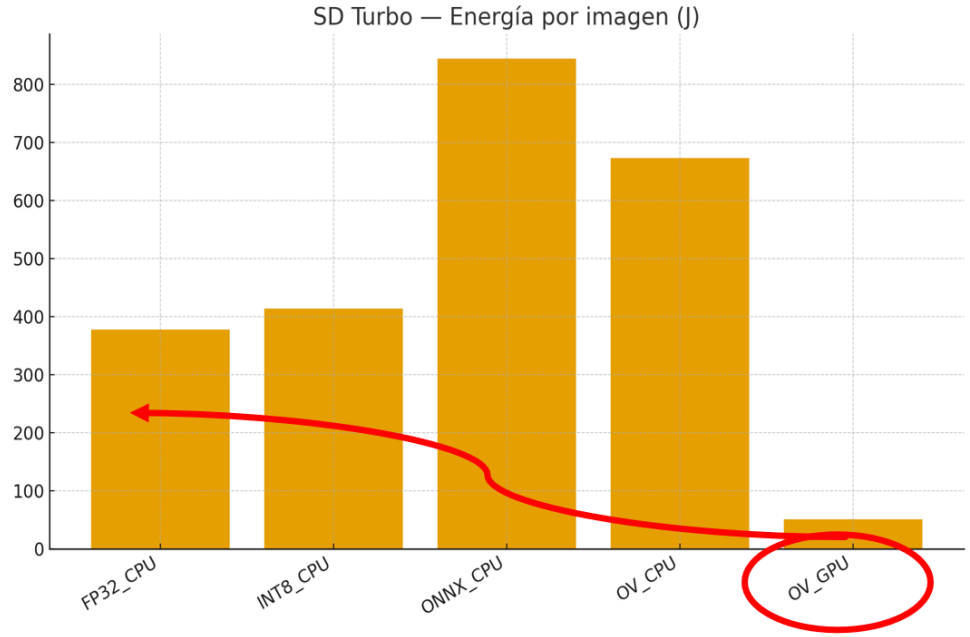

Figura 5-10. Energía inferencia SD Turbo

Una vez entramos en territorio OpenVINO y seleccionamos como device GPU, vemos que arrasa en todo. Es casi 6 veces más rápido y casi 7 veces más eficiente que OpenVINO en CPU, ya que la carga la pasa a llevar la GPU en su mayor parte (trasladando el bucle pesado, la UNet, a la GPU con kernels optimizados), liberando así CPU y RAM, lo que ayuda a multitarea y a estabilidad térmica. Si además comparamos

con escenarios anteriores, las mejoras de OpenVINO GPU son de prácticamente igual calibre: usando Stable Diffusion 1.5 en FP32 como referencia, OpenVINO GPU recorta el tiempo de aproximadamente 32 segundos a 5 segundos, siendo casi 6.5 veces mejor, y con una mejora de casi 4.5 veces en el apartado del consumo energético. Por su parte, en la vista general, OpenVINO CPU queda intermedio en tiempo y alto en energía.

En SD Turbo el comportamiento es el mismo, aunque con métricas reducidas en proporción al ahorro de recursos dados por la naturaleza del modelo.

Por otra parte, no se pudo probar el acelerador NPU en estos modelos de imagen, porque el plugin NPU no soporta todavía la operación `GroupNormalization` que tiene lugar en el grafo UNet de SD 1.5 y SD Turbo. Los errores indican que el framework de OpenVINO sabe qué es, pero el backend NPU no tiene kernel para ella y la compilación falla. Para validar el pipeline, se hizo una prueba funcional en NPU con un modelo de mosaicos de tipo transferencia de estilo rápida (red convolucional imagen – imagen), pero no aportó evidencia concluyente de mejora en latencia o energía, así que no la incluiremos.

Con los resultados obtenidos, podemos determinar que OpenVINO GPU es la opción más rápida y eficiente, además de liberar CPU (que puede resultar útil para otras tareas en paralelo) y reducir RAM respecto a PyTorch FP32, PyTorch INT8, ONNX y OpenVINO CPU.

Como punto final se ejecutaron algunas pruebas más con el scheduler DPM++2M, que es mucho más avanzado y moderno, ofreciendo imágenes de mejor calidad a costa de una reducción en la velocidad. A su vez, se aumentaron los pasos a 30 para ver más mejoría, si cabe, en la calidad de imagen. En la siguiente tabla podemos ver ese coste en eficiencia.

Métrica	OV GPU (antes - Euler)	OV GPU (nuevo - DPM++2M)	Cambio % (nuevo vs antes)
Steps	8	30	
Latencia mediana (ms)	4984.94	16010.70	221.2%
Throughput global (img/s)	0.2006	0.0624	-68.9%
Energía/imagen (J) [CPU package]	290.35	872.71	200.6%
Potencia media paquete (W) [CPU]	58.24	54.49	-6.4%
CPU proceso (%)	4.0420	1.2933	-68.0%
Memoria media (MB)	4540.36	4539.58	-0.0%
Temperatura media (°C) [CPU]	88.04	84.64	-3.9%

Tabla 5-10. SD 1.5 Euler vs SD 1.5 DPM++2M

Por otra parte, en las siguientes figuras se puede observar el aumento de calidad en la generación de las imágenes en SD 1.5 entre ambos schedulers y también con respecto a SD Turbo (estas últimas únicamente con Euler), siendo de izquierda a derecha: SD 1.5 DPM++2M, SD 1.5 Euler, SD Turbo Euler.

Figura 5-11. A picture of Ash Ketchum and his Pokemons swimming in the beach

Figura 5-12. Ultra-detailed photo of a siamese cat, dramatic lighting

Figura 5-13. Streets of Seoul at night, pixel art

Figura 5-14. A super mario bros style castle on a floating island, volumetric fog

Figura 5-15. Minimal product render of some juicy watermelon chopped on shiny marble

Capítulo 6 - Conclusiones y trabajo futuro

Este Trabajo Fin de Grado ha permitido explorar la optimización de modelos generativos en hardware de última generación mediante la aplicación de un mismo flujo experimental a tareas de texto (DistilGPT-2) e imagen (Stable Diffusion 1.5 y Turbo). Se ha demostrado que la exportación a OpenVINO puede permitir reducir de forma significativa los tiempos de inferencia respecto a PyTorch y ONNX, manteniendo una calidad de salida equivalente y en la mayoría de los casos superándola.

En el caso del modelo de texto, la cuantización dinámica sí que supone una ventaja de rendimiento y eficiencia frente a OpenVINO, ONNX, y FP32. Sin embargo, en OpenVINO compilando un IR nativo con sus respectivas fusiones y con un decode equivalente autorregresivo, se esperaría que la GPU tuviese esa ventaja. De todas formas, ya es una optimización sencilla que aplicar en los despliegues en entornos Edge.

En el caso de los modelos de imagen, tanto runtime como dispositivo son factores decisivos, y se llegó a una conclusión clara: entre los distintos backends, OpenVINO en GPU es la opción con mayor rendimiento y eficiencia, además de liberar casi por completo la CPU. En nuestras pruebas se comprobó que ofrece la mejor latencia con mucha diferencia, con respecto a las otras rutas CPU probadas. Esto concluye que la configuración más recomendable es OpenVINO GPU, priorizando SD Turbo para máxima rapidez (elección útil para prototipados, interactividad…) y SD 1.5 cuando la calidad de las imágenes generadas es prioritaria.

Se demuestra así que los flujos de optimización aplicados en este proyecto permiten ejecutar modelos generativos de diferente índole en entornos Edge, que, en condiciones normales, resultarían demasiado costosos, y proporcionan estrategias para llevar a cabo una reducción del consumo y aumentar la tan necesaria eficiencia energética en fase de inferencia.

Trabajo futuro

Se proponen las siguientes líneas de trabajo para el seguimiento de esta investigación:

- Aplicar diversas optimizaciones a los grafos exportados sin Optimum: como podrían ser IOBinding o KV-cache preasignada y ver si así se ve mejoría en modelos de texto tanto en ONNX Runtime como en OpenVINO Runtime.
- Evaluar el paralelismo real en modo throughput: los scripts de la parte de modelos de imagen en OpenVINO ejecutan inferencias de manera secuencial, sería interesante la implementación de batching o peticiones asíncronas y ver si así aumenta el rendimiento en imágenes por segundo.
- Uso de diferentes datasets ya implementados en investigación: como podría ser GenAI-Bench y ver su comportamiento en nuestro flujo de trabajo.
- Ampliación a otros modelos generativos: la replicación de la metodología utilizada en el proyecto con modelos de audio o con otros tipos de modelos mejoraría la validez de las conclusiones y facilitaría un análisis más profundo sobre la efectividad de las optimizaciones.
- Evaluación objetiva de la calidad: la combinación de la evaluación subjetiva con métricas cuantitativas, como CLIPScore o FID, le daría al análisis comparativo de los resultados una mayor rigurosidad.
- Automatización del flujo experimental: si todos los scripts se integran en una herramienta unificada que realice pruebas, reúna resultados y produzca tablas y gráficos de manera automática, se facilitaría la posibilidad de repetir el proyecto y se ampliaría su aplicabilidad a otros contextos.

Introduction

Motivation

Generative Artificial Intelligence is one of the most innovative branches of artificial intelligence, as stated in the article "Generative Artificial Intelligence: A Systematic Review and Applications" [1]. Unlike classical artificial intelligence, which focuses on classification or prediction models, generative artificial intelligence is capable of producing original material in multiple formats (text, images, video, audio, code, etc.) in response to a user's command or instruction. Among generative artificial intelligence, large-scale language models, known as LLMs (Large Language Models), stand out.

LLMs are models based on self-supervised learning, meaning the model is trained with input data (unlabeled text). In this way, the model learns to predict the next tokens. What differentiates LLMs from other models based on this type of learning is that LLMs are trained with immense volumes of textual data, enabling them to learn distributed representations of language. To achieve this, the transformer architecture has been essential. This is a neural network model based on attention mechanisms that process sequences in parallel, allowing the capture of long-range dependencies in textual sequences. In this way, they are able to generate coherent and contextualized text.

LLMs have revolutionized artificial intelligence, expanding from research environments to everyday applications as widely known as ChatGPT. As Anand Gokul points out in his article "LLMs and AI: Understanding its reach and impact" [2], LLMs are transforming the way we communicate and understand language, with an impact that extends to both professional and personal domains.

At present, the ongoing discussion on the sustainability and long-term impact of artificial intelligence (hereafter referred to as AI) is mainly centered on the training phase of the models, as it entails an excessive demand for energy resources, water sources, and material resources. Using a large-scale model such as GPT-3, for example, required more than 1,200 MWh and emitted approximately 550 tons of carbon dioxide. In addition, it consumed more than 700,000 liters of water for thermal conditioning. However, recent studies indicate that the environmental impact of the inference phase is more critical,

since, unlike training, inference is continuous and large-scale, accounting for 90% of the total life-cycle cost of LLMs [3]. This highlights the current importance of optimizing this inference phase in terms of consumption and sustainability, a key point in our study.

In this line of argument, Jegham et al. [3], How hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference, represents one of the first systematized procedures to quantify the environmental footprint of inference in state-of-the-art language models. The research study covers thirty models from OpenAI, Anthropic, Meta, and DeepSeek. Among the most notable results, it is highlighted that O3 and DeepSeek-R1 are the most resource-intensive, exceeding 33 Wh per long query, which is equivalent to 70 times the consumption of GPT-4.1 nano, one of the most efficient models. Conversely, Claude-3.7 Sonnet stands out as the most sustainable model, as it achieves a balance between capability and environmental sustainability.

A fundamental characteristic reflected in the study is the central role of deployment infrastructure. For example, GPT-4º mini, although a smaller-scale model, is less efficient than GPT-4º when run on less advanced hardware (A100 vs. H100/H200). Similarly, DeepSeek models reveal oversized water footprints due to their data centers in China, with higher PUE values and less eco-friendly cooling systems.

The results show that AI sustainability depends not only on algorithm design but also on external variables such as hardware type, data center efficiency, and energy sources, among other contributing factors. Likewise, the importance is emphasized of implementing standards to certify transparency and regulate the environmental footprint per query, encouraging the use of renewable energy and promoting environmentally sustainable cooling technologies.

Goals

The purpose of this paper is to explore and evaluate the optimization of generative AI model inference on next-generation hardware, paying special attention to their efficiency in edge environments and their computational sustainability. Based on this motivation, the following objectives are proposed:

General Objective

To analyze and validate an optimization pipeline for generative AI models in text and image domains, employing open frameworks and specialized runtimes, with the goal of improving performance and efficiency on next-generation Intel hardware.

Specific Objectives

- Implement an experimental environment based on Python and Ubuntu that integrates the main libraries and frameworks of the current ecosystem: PyTorch, Hugging Face Transformers, ONNX, and OpenVINO.
- Select and run representative models for language (DistilGPT-2 and Tiny-GPT-2) and image (Stable Diffusion 1.5 and Stable Diffusion Turbo), comparing their behavior under different optimization scenarios.
- Evaluate the impact of quantization techniques (FP32 vs. INT8) on latency, throughput, memory consumption, and CPU usage.
- Analyze model portability through export to ONNX and execution in ONNX Runtime, contrasting results against PyTorch.
- Measure the performance gains achieved with OpenVINO Runtime on Intel hardware, highlighting its relevance for efficient deployment in edge devices.
- Reflect on the sustainability of generative AI, considering the energy and material impact of the inference phase and the importance of efficient infrastructures.

Work plan

The development of this work was structured into a series of interconnected phases that enabled progress from theoretical review to practical experimentation and results analysis. The planning considered both the conceptual dimension and the technical implementation, in order to ensure coherence between motivations, objectives, and the results obtained.

The project was planned with attention to both technical implementation and conceptual grounding, and was structured into the following phases:

1) Review of the state of the art and definition of the software ecosystem

A literature review was conducted on the current ecosystem surrounding generative AI, in order to identify the tools to be employed.

2) Configuration of the experimental environment

A test environment was implemented to provide access to the libraries and frameworks required for the correct execution of the experimentation phase.

3) Selection and preparation of representative models

Different language and image models were selected for use, and their optimization was planned.

4) Execution of experiments and optimization scenarios

Runs were carried out in multiple scenarios, first establishing a baseline, and then as the model was optimized and fine-tuned.

5) Comparative analysis of results

The data obtained was processed using the planned analysis tools and visualized, allowing comparisons to be generated across models and environments.

6) Discussion and conclusions

Finally, the results were discussed in relation to the reviewed literature, highlighting the performance improvements achieved and potential future lines of work.

Conclusions and future work

This thesis has explored the optimization of generative models on next-gen hardware by applying the same experimental flow to text (DistilGPT-2) and image (Stable Diffusion 1.5 and Turbo) tasks. It has been shown that exporting to OpenVINO can significantly reduce inference times compared to PyTorch and ONNX, while maintaining equivalent output quality and, in most cases, surpassing it.

In the case of the text model, dynamic quantization does offer a performance and efficiency advantage over OpenVINO, ONNX, and FP32. However, in OpenVINO, when compiling a native IR with its respective fusions and with an equivalent autoregressive decode, the GPU would be expected to have that advantage. In any case, it is already a simple optimization to apply in Edge environment deployments.

In the case of image models, both runtime and device are decisive factors, and a clear conclusion was reached: among the different backends, OpenVINO on GPU is the option with the highest performance and efficiency, in addition to almost completely freeing up the CPU. Our tests showed that it offers the best latency by far, compared to the other CPU routes tested. This leads to the conclusion that the most recommended configuration is OpenVINO GPU, prioritizing SD Turbo for maximum speed (useful for prototyping, interactivity, etc.) and SD 1.5 when the quality of the generated images is a priority.

This demonstrates that the optimization workflows applied in this project allow running generative models of various types in edge environments, which, under normal conditions, would be too costly, and provide strategies for reducing consumption and increasing the much-needed energy efficiency in the inference phase.

Future Work

The following lines of work are proposed for the follow-up to this research:

• Apply various optimizations to the graphs exported without Optimum: such as IOBinding or pre-allocated KV-cache and see if this improves text models in both ONNX Runtime and OpenVINO Runtime.

• Evaluate real parallelism in throughput mode: the scripts in the image model part of OpenVINO execute inferences sequentially. It would be interesting to implement batching or asynchronous requests and see if this increases performance in images per second.

• Use of different datasets already implemented in research: such as GenAI-Bench and see how they behave in our workflow.

• Extension to other generative models: Replicating the methodology used in the project with audio models or other types of models would improve the validity of the conclusions and facilitate a more in-depth analysis of the effectiveness of the optimizations.

• Objective quality assessment: combining subjective assessment with quantitative metrics, such as CLIPScore or FID, would make the comparative analysis of the results more rigorous.

• Automation of the experimental flow: if all scripts were integrated into a unified tool that automatically performs tests, collects results, and produces tables and graphs, it would be easier to repeat the project and extend its applicability to other contexts.

BIBLIOGRAFÍA

[1] S. S. Sengar, A. B. Hasan, S. Kumar, y F. Carroll, «Generative artificial intelligence: A systematic review and applications», *Multimedia Tools and Applications*, vol. 84, n.º 21, pp. 23661-23700, 2024, doi: 10.1007/s11042-024-20016-1.

[2] A. Gokul, «LLMs and AI: Understanding its reach and impact». 2023.

[3] N. Jegham, M. Abdelatti, L. Elmoubarki, y A. Hendawi, «How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference», *arXiv preprint*, 2025, [En línea]. Disponible en: https://arxiv.org/abs/2505.09598

[4] S. Khan, M. Motie, S. Kocak, y M. Raza, «Energy-Efficient Large Language Models: Challenges and Opportunities». 2025.

[5] J. Chen y others, «Measuring Energy Efficiency of LLM Inference on GPUs». 2024.

[6] A. Vaswani y others, «Attention is All You Need», en *Proc. NeurIPS*, 2017.

[7] A. Radford y others, «Language Models are Unsupervised Multitask Learners». 2019.

[8] T. Brown y others, «Language Models are Few-Shot Learners», en *Proc. NeurIPS*, 2020.

[9] H. Touvron y others, «LLaMA: Open and Efficient Foundation Language Models». 2023.

[10] I. Goodfellow y others, «Generative adversarial nets», en *Proc. NeurIPS*, 2014.

[11] T. Karras, S. Laine, y T. Aila, «A Style-Based Generator Architecture for Generative Adversarial Networks», en *Proc. CVPR*, 2019.

[12] P. Dhariwal y A. Nichol, «Diffusion Models Beat GANs on Image Synthesis», en *Proc. NeurIPS*, 2021.

[13] R. Rombach y others, «High-Resolution Image Synthesis with Latent Diffusion Models», en *Proc. CVPR*, 2022.

[14] A. van den Oord y others, «WaveNet: A Generative Model for Raw Audio». 2016.

[15] R. Prenger, R. Valle, y B. Catanzaro, «WaveGlow: A Flow-based Generative Network for Speech Synthesis». 2019.

[16] Y. Wang y others, «Tacotron: Towards End-to-End Speech Synthesis». 2017.

[17] P. Wang y others, «VALL-E: Neural Codec Language Models are Zero-Shot Text to Speech Synthesizers». 2023.

[18] J. Alayrac y others, «Flamingo: a Visual Language Model for Few-Shot Learning». 2022.

[19] OpenAI, «GPT-4 Technical Report». 2023.

[20] DeepMind, «Gemini: A Family of Highly Capable Multimodal Models». 2023.

[21] J. Casadiego, «Edge AI: Opportunities and Challenges». 2023.

[22] L. M. Candanedo, V. Feldheim, y D. Deramaix, «Edge computing for real-time applications». 2020.

[23] J. Elordi, «Privacy and Security in Edge AI Deployments». 2023.

[24] J. G. de Duenas, «Optimizing Energy Efficiency in Edge AI». 2023.

[25] A. Martin, «Secure and Efficient Wireless Edge Nodes». 2024.

[26] D. A. Puertas, «Model Compression for Edge AI». 2024.

[27] J. Intriago, «Edge AI for Industry 4.0 Applications». 2025.

[28] F. R. Enciso, «Sustainable AI in Edge Computing». 2025.

[29] S. Nasar y A. Al-Batahari, «Efficient AI Models for IoT and Edge Devices». 2025.

[30] A. Joshua y others, «Integer Quantization for Efficient AI». 2025.

[31] S. Han, H. Mao, y W. J. Dally, «Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding», en *Proc. ICLR*, 2016.

[32] P. Soni y others, «Quantization for IoT Neural Networks». 2025.

[33] K. Manoj y others, «Low-Latency Quantized AI for Wireless Systems». 2025.

[34] S. Han, J. Pool, J. Tran, y W. J. Dally, «Learning both Weights and Connections for Efficient Neural Networks», en *Proc. NeurIPS*, 2015.

[35] C. Marvellous y others, «Energy-Efficient Wearable AI with Pruning». 2025.

[36] R. Harris, «Pruning for Autonomous Drone Vision Systems». 2025.

[37] G. Hinton, O. Vinyals, y J. Dean, «Distilling the Knowledge in a Neural Network», en *Proc. NeurIPS Workshop*, 2015.

[38] V. Sanh, L. Debut, J. Chaumond, y T. Wolf, «DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter». 2019.

[39] A. Noura y others, «Knowledge Distillation for IoT Object Detection». 2025.

[40] P. Aach y others, «Distillation for Edge AI in HPC Systems». 2025.

[41] J. Dean y others, «Large Scale Distributed Deep Networks», en *Proc. NIPS*, 2012.

[42] M. Shoeybi y others, «Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism». 2019.

[43] P. Micikevicius y others, «Mixed Precision Training», en *Proc. ICLR*, 2018.

[44] T. Chen y others, «Training Deep Nets with Sublinear Memory Cost». 2016.

[45] I. Loshchilov y F. Hutter, «Decoupled Weight Decay Regularization», en *Proc. ICLR*, 2019.

[46] Y. You y others, «Large Batch Optimization for Deep Learning: Training BERT in 76 minutes», en *Proc. ICLR*, 2020.

[47] N. Shazeer y M. Stern, «Adafactor: Adaptive Learning Rates with Sublinear Memory Cost», en *Proc. ICML*, 2018.

[48] A. Schacht y M. Lanquillon, «Efficient Tuning of Large Language Models». 2024.

[49] X. Author y others, «Efficient Key-Value Caching for Million-Token Contexts». 2025.

[50] W. Chen, Y. Zhang, Z. Zhou, Y. Li, y C. Wang, «Accelerating LLM Inference with Staged Speculative Decoding». 2023.

[51] X. Author y others, «Integrated Speculative Decoding without Auxiliary Models». 2024.

[52] T. Dao y others, «FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness». 2022.

[53] M. Zeeshan, «Containerized AI Deployment with Docker and Singularity». 2024.

[54] H. Li, Q. Zhang, J. Chen, y H. Xu, «Efficient Sparse Inference Software Accelerator for Transformer-based Language Models on CPUs». 2023.

[55] S. Nimmagadda, «Optimized Edge AI Inference with OpenVINO». 2025.

[56] Intel, «Intel NUC Product Overview». 2023.

[57] InfluxData, «Edge AI with Intel NUC». 2023.

[58] E. Computing, «Intel NUC with Movidius and OpenVINO». 2023.

[59] S. Computing, «Virtualization on Intel NUCs for Edge AI». 2023.

[60] OnLogic, «GPU vs NUC for AI Workloads». 2023.

[61] P. Tobiasz y others, «Comparing Edge AI Platforms: Jetson, Coral, and NUC». 2023.

[62] X. Zhou y others, «Privacy and Latency Challenges in Cloud AI». 2023.

[63] E. Frantar y D. Alistarh, «Edge Limitations of Model Optimization». 2023.

[64] B. Jacob y others, «Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference», en *Proc. CVPR*, 2018.

[65] D. Patterson y others, «Carbon Emissions and Energy of AI Datacenters». 2021.