AirLLM y Colibrì permiten ejecutar modelos de lenguaje que, en condiciones normales, no cabrían en la VRAM ni en la memoria RAM de un ordenador doméstico. Sin embargo, no solucionan el problema de la misma manera, no están orientados a los mismos modelos y tampoco ofrecen el mismo nivel de rendimiento.

AirLLM utiliza principalmente una estrategia de carga progresiva de capas o expertos, mientras que Colibrì convierte la VRAM, la RAM y el almacenamiento NVMe en una jerarquía de memoria administrada específicamente para modelos Mixture-of-Experts.

Esta diferencia es fundamental. AirLLM es una solución relativamente genérica que podemos utilizar con muchas familias de modelos de Hugging Face. Colibrì, en cambio, es un motor especializado que intenta extraer el máximo rendimiento posible de modelos MoE gigantes, cargando desde el almacenamiento únicamente los expertos necesarios para procesar cada token.

Estado del artículo: julio de 2026. Ambos proyectos continúan evolucionando, especialmente Colibrì, por lo que debemos comprobar siempre la documentación actual antes de instalar versiones concretas o descargar modelos de cientos de gigabytes.

El problema que intentan resolver AirLLM y Colibrì

Los parámetros de un modelo de lenguaje deben estar disponibles en algún tipo de memoria para poder realizar la inferencia. Normalmente intentamos colocar el modelo completo en la VRAM de una o varias tarjetas gráficas porque la VRAM proporciona mucho más ancho de banda que la memoria RAM y, especialmente, que una unidad NVMe.

El problema aparece cuando el modelo es más grande que la memoria disponible. Un modelo denso de 70.000 millones de parámetros almacenado en FP16 puede necesitar aproximadamente 140 GB solamente para sus pesos. A esto debemos añadir la caché KV, los buffers temporales, las activaciones y la memoria utilizada por el propio motor de inferencia.

Una cuantización de 8 bits reduce aproximadamente a la mitad el espacio ocupado por los pesos, mientras que una cuantización de 4 bits puede reducirlo aproximadamente a una cuarta parte. No obstante, incluso un modelo de 70.000 millones de parámetros cuantizado a 4 bits puede necesitar alrededor de 35 o 45 GB, dependiendo del formato, los metadatos y la técnica de cuantización utilizada.

Cuando el modelo no cabe completamente en la VRAM podemos repartirlo entre GPU y CPU. Motores como llama.cpp permiten colocar determinadas capas en la GPU y mantener el resto en la RAM. Esta solución funciona correctamente mientras la suma de la VRAM y la RAM resulte suficiente para almacenar el modelo.

AirLLM y Colibrì intentan ir más lejos. Ambos permiten que una parte importante de los pesos permanezca en el almacenamiento y se cargue solamente cuando sea necesaria.

```
Ejecución convencional:

Modelo completo
      |
      v
VRAM o RAM
      |
      v
Cálculo


Ejecución mediante streaming:

Modelo completo
      |
      v
SSD o NVMe
      |
      v
Carga parcial de pesos
      |
      v
RAM o VRAM
      |
      v
```

```
Cálculo
```

El precio que pagamos es evidente: **el almacenamiento es mucho más lento que la RAM y la VRAM**. Conseguir que un modelo entre en la memoria disponible no significa necesariamente que podamos utilizarlo a una velocidad interactiva.

Por tanto, debemos separar siempre dos conceptos:

- **Capacidad:** conseguir que el modelo pueda ejecutarse sin agotar la memoria.
- **Rendimiento:** conseguir que el modelo genere tokens con una velocidad razonable.

AirLLM y Colibrì destacan principalmente en el primer punto. El rendimiento dependerá enormemente del tipo de modelo, la cantidad de RAM, la VRAM disponible, el ancho de banda del almacenamiento y la frecuencia con la que deban cargarse pesos desde el disco.

Modelos densos y modelos Mixture-of-Experts

Para entender las diferencias entre ambos proyectos debemos distinguir entre modelos densos y modelos Mixture-of-Experts, también conocidos como MoE.

En un modelo denso, prácticamente todos los parámetros de todas las capas participan en el procesamiento de cada token. Aunque ejecutemos las capas secuencialmente, cada token debe atravesar todo el modelo.

```
Token
  |
  v
Capa 1 completa
  |
  v
Capa 2 completa
  |
  v
Capa 3 completa
  |
  v
...
  |
  v
Capa final
```

Esto significa que, si mantenemos el modelo en el almacenamiento, tendremos que leer una parte muy grande de sus pesos para generar cada nuevo token.

Los modelos MoE funcionan de otra manera. Cada bloque MoE contiene numerosos expertos, pero un router selecciona solamente una pequeña cantidad de ellos para procesar cada token.

```
Token
  |
  v
Router
  |
  +----> Experto 7
  |
  +----> Experto 42
  |
  +----> Experto 193
  |
  v
Combinación de resultados
```

Un modelo puede tener cientos de miles de millones o incluso billones de parámetros totales, pero activar solamente una

fracción de ellos para cada token. Esto permite aumentar la capacidad total del modelo sin multiplicar en la misma proporción el cálculo realizado durante cada inferencia.

No obstante, los expertos que se activan pueden cambiar de un token a otro. Si no caben en la RAM o en la VRAM, debemos localizarlos, leerlos desde el almacenamiento, colocarlos en memoria y ejecutar sus matrices.

AirLLM puede trabajar tanto con modelos densos como con modelos MoE. Colibrì se concentra específicamente en explotar la estructura dispersa de los modelos MoE.

Cómo funciona AirLLM

AirLLM apareció inicialmente como una biblioteca capaz de ejecutar un modelo Llama de 70.000 millones de parámetros utilizando una tarjeta gráfica con aproximadamente 4 GB de VRAM.

Su estrategia original se basa en la inferencia por capas. Como las capas de un Transformer se ejecutan de manera secuencial, AirLLM no mantiene todo el modelo dentro de la GPU. En su lugar, conserva solamente la capa que está calculando en ese momento.

```
Almacenamiento
      |
      v
Cargar capa 1 en VRAM
      |
      v
Ejecutar capa 1
      |
      v
Liberar capa 1
      |
      v
Cargar capa 2 en VRAM
      |
      v
Ejecutar capa 2
      |
      v
Liberar capa 2
      |
      v
Continuar hasta la última capa
```

Si un modelo tiene 80 capas y cada una ocupa aproximadamente 1,6 GB, AirLLM no necesita almacenar los 128 GB simultáneamente en la GPU. Solamente necesita memoria suficiente para cargar una capa, mantener la caché KV y reservar los buffers utilizados durante el cálculo.

Esto permite reducir enormemente la VRAM necesaria, pero no elimina los pesos del modelo. Los pesos siguen ocupando espacio en el almacenamiento y deben transferirse repetidamente.

Separación del modelo por capas

Los modelos distribuidos mediante Hugging Face suelen estar divididos en archivos de varios gigabytes. Estos archivos no tienen por qué coincidir con los límites de cada capa. Un mismo archivo puede contener partes de varias capas y una capa puede encontrarse repartida entre varios archivos.

Leer un archivo de 10 GB para recuperar una capa de 1,6 GB desperdiciaría ancho de banda y memoria. Por eso AirLLM realiza una preparación inicial en la que reorganiza el modelo y guarda cada capa por separado.

AirLLM utiliza Safetensors y técnicas de mapeo de memoria para evitar copias innecesarias. Después de la conversión, puede cargar solamente los pesos correspondientes a la capa que necesita ejecutar.

Esta preparación tiene una consecuencia importante: **podemos necesitar espacio suficiente para almacenar temporalmente el modelo original y la copia reorganizada por capas**.

AirLLM ofrece una opción para eliminar el modelo original después de realizar la separación. Esto reduce el espacio permanente necesario, pero debemos asegurarnos de que la conversión haya terminado correctamente antes de borrar los archivos originales.

Uso del dispositivo meta

AirLLM utiliza el dispositivo meta disponible en PyTorch y Hugging Face Accelerate. Este dispositivo permite crear la estructura lógica del modelo sin cargar inmediatamente sus pesos en memoria.

Podemos pensar en el dispositivo meta como una representación vacía del modelo:

```
Estructura del modelo:
  Capa 1
  Capa 2
  Capa 3
  ...
  Capa 80

Pesos cargados inicialmente:
  Ninguno
```

Cuando llega el momento de ejecutar una capa, AirLLM carga sus pesos reales en el dispositivo correspondiente. Después de utilizarla, puede liberar esa memoria y continuar con la siguiente.

Prefetch de capas

La carga desde el almacenamiento puede solaparse parcialmente con el cálculo. Mientras la GPU ejecuta una capa, AirLLM puede comenzar a cargar la siguiente.

```
GPU:     calcular capa 20
NVMe:    cargar capa 21

GPU:     calcular capa 21
NVMe:    cargar capa 22
```

Este prefetch no elimina el coste de las lecturas, pero puede ocultar una parte de la latencia si el tiempo de cálculo de la capa es suficiente para cubrir la carga de la siguiente.

Compresión de pesos en AirLLM

AirLLM puede utilizar pesos sin cuantizar, pero también dispone de compresión de 8 y 4 bits.

En este caso, la compresión no solamente intenta reducir la memoria. También intenta reducir la cantidad de información que debe leerse desde el almacenamiento. Si el cuello de botella es el disco, reducir a la mitad o a una cuarta parte los bytes leídos puede mejorar considerablemente la velocidad.

El flujo sería similar al siguiente:

```
Pesos FP16:
  1,6 GB por capa

Pesos de 8 bits:
  aproximadamente 800 MB por capa

Pesos de 4 bits:
```

```
  aproximadamente 400 MB por capa
```

Las cantidades reales dependen del formato y de los metadatos adicionales. Además, cualquier cuantización puede introducir una pérdida de precisión. No debemos interpretar que ejecutar AirLLM con compresión de 4 bits mantiene exactamente la misma calidad que ejecutar el modelo original.

AirLLM y los modelos MoE

Las versiones modernas de AirLLM ya no se limitan al streaming de capas completas. También pueden aplicar streaming por experto en determinados modelos MoE.

En lugar de cargar todos los expertos de una capa MoE, AirLLM puede cargar los expertos que el router haya seleccionado para el token actual. Este mecanismo permite ejecutar modelos con cantidades enormes de parámetros totales utilizando poca VRAM.

El repositorio de AirLLM indica compatibilidad con familias como:

- Llama 2, Llama 3, Llama 3.1, Llama 3.3 y Llama 4.
- Qwen, Qwen 2, Qwen 2.5 y Qwen 3, incluidos algunos modelos MoE y FP8.
- DeepSeek V2, DeepSeek V3 y DeepSeek R1.
- Mistral y Mixtral.
- Phi.
- Gemma.
- ChatGLM.
- Baichuan.
- InternLM.
- Yi.

AirLLM utiliza AutoModel para detectar automáticamente la arquitectura del modelo. Sin embargo, debemos interpretar la expresión “compatible” con cautela. Los modelos que utilizan código remoto, operadores personalizados, Flash Attention obligatorio o formatos nuevos pueden necesitar versiones concretas de Transformers, PyTorch, CUDA y dependencias adicionales.

El repositorio afirma que puede ejecutar modelos como Qwen3-235B utilizando unos 3 GB de VRAM, Llama 3.1 405B con unos 8 GB y DeepSeek-V3 con aproximadamente 12 GB. También anuncia streaming por experto para modelos MoE todavía más grandes. Estas cifras describen la VRAM ocupada y no la memoria total, el espacio de almacenamiento ni la velocidad de generación.

Cómo funciona Colibrì

Colibrì adopta una estrategia más especializada. No intenta ejecutar cualquier Transformer dividiéndolo genéricamente por capas. Su objetivo es administrar modelos MoE gigantes como si la VRAM, la RAM y el almacenamiento fueran diferentes niveles de una única memoria.

```
Nivel más rápido y pequeño:
  VRAM

Nivel intermedio:
  RAM

Nivel más lento y grande:
  NVMe
```

```
Jerarquía completa:
  VRAM <-> RAM <-> NVMe
```

Colibrì decide qué partes del modelo deben residir en cada nivel. Las partes utilizadas continuamente se mantienen en la memoria más rápida disponible. Los expertos utilizados con frecuencia se almacenan en caché. Los expertos fríos permanecen en el NVMe hasta que el router los necesita.

El caso principal del proyecto es GLM-5.2, un modelo MoE con aproximadamente 744.000 millones de parámetros totales y unos 40.000 millones de parámetros activos por token.

Según la organización utilizada por Colibrì:

- La parte densa contiene aproximadamente 17.000 millones de parámetros.
- La parte densa cuantizada a INT4 ocupa alrededor de 9,9 GB y permanece residente.
- El modelo contiene 19.456 expertos enrutables.
- Los expertos ocupan aproximadamente 370 GB en el almacenamiento.
- Cada experto cuantizado ocupa alrededor de 19 MB.
- Los expertos variables seleccionados para un token pueden implicar alrededor de 11 GB de pesos.

La parte densa incluye elementos que siempre deben ejecutarse, como la atención, los embeddings y los expertos compartidos. Mantenerla residente evita tener que volver a leerla para cada token.

Los expertos enrutables son diferentes. El router decide cuáles deben procesar cada token y Colibrì intenta obtenerlos desde el nivel más rápido posible.

```
El router solicita un experto
             |
             v
¿Está en VRAM?
   | Sí                    | No
   v                       v
Ejecutar             ¿Está en RAM?
                        | Sí        | No
                        v           v
                     Ejecutar    Leer desde NVMe
                                     |
                                     v
                                Guardar en caché
                                     |
                                     v
                                  Ejecutar
```

Caché LRU y almacén de expertos frecuentes

Colibrì utiliza una caché LRU para conservar los expertos utilizados recientemente. Cuando la memoria se llena y es necesario cargar un experto nuevo, el motor puede expulsar uno que lleve más tiempo sin utilizarse.

También registra la frecuencia de utilización de los expertos. Los expertos más utilizados pueden quedar fijados en RAM o VRAM para evitar lecturas repetidas desde el almacenamiento.

Esto significa que el rendimiento puede mejorar cuando ejecutamos cargas de trabajo parecidas durante cierto tiempo. Si realizamos consultas relacionadas con programación, es posible que determinados expertos se activen con frecuencia y terminen residiendo en un nivel rápido.

Colibrì guarda esta información en un archivo de uso. De esta forma, parte del aprendizaje de la caché puede mantenerse entre ejecuciones.

Prefetch mediante anticipación del router

Colibrì ejecuta el router con anticipación para intentar predecir los expertos que necesitará la siguiente capa. Mientras se calcula la capa actual, un hilo puede comenzar a leer desde el almacenamiento los expertos de la capa siguiente.

```
Capa actual:
  ejecutar expertos seleccionados

En paralelo:
  ejecutar router de la siguiente capa
  localizar expertos necesarios
  comenzar lectura desde NVMe
```

El proyecto indica que el enrutamiento presenta suficiente estructura como para que esta anticipación resulte útil. No obstante, un error de predicción o un cambio poco frecuente en los expertos seleccionados puede provocar una lectura bajo demanda y detener temporalmente el cálculo.

Lectura directa y operaciones asíncronas

Colibrì intenta reducir el número de operaciones de entrada y salida. Las matrices pertenecientes a un mismo experto se guardan de manera contigua para poder recuperarlas con una única lectura.

También utiliza una cola asíncrona para cargar expertos mientras el procesador o la GPU ejecutan los que ya están disponibles.

En algunos sistemas puede utilizar acceso directo al almacenamiento para evitar la caché de páginas del sistema operativo. Esta opción puede mejorar el rendimiento con determinados NVMe, pero también puede empeorarlo con unidades sin DRAM, almacenamiento virtualizado o dispositivos con bajo rendimiento sostenido. Debemos medir ambas configuraciones en nuestro equipo.

Uso de dos unidades NVMe

Colibrì puede utilizar dos copias del mismo modelo almacenadas en unidades diferentes. El motor distribuye las lecturas entre ambas unidades según el ancho de banda disponible.

```
                +----> NVMe 1: expertos A, C, E
Solicitud -------+
                +----> NVMe 2: expertos B, D, F
```

No se trata de dividir el modelo en dos mitades distintas. Cada unidad puede contener una copia completa, aunque también puede utilizarse una copia parcial en el segundo disco. El motor asigna cada experto a una unidad mediante una función determinista y evita almacenar dos veces el mismo experto en caché.

La mejora real depende del rendimiento de ambas unidades y de si el procesador puede consumir los datos con suficiente rapidez. Duplicar el ancho de banda del almacenamiento no garantiza duplicar los tokens por segundo, porque pueden aparecer nuevos cuellos de botella en la RAM, el procesador o la GPU.

Caché KV comprimida

Además de los pesos, los modelos necesitan guardar información sobre los tokens anteriores en la caché KV. Esta caché crece con la longitud del contexto y puede consumir una cantidad importante de memoria.

Colibrì aprovecha la arquitectura MLA de GLM-5.2 para mantener un estado KV comprimido. El proyecto indica que almacena 576 valores de coma flotante por token en lugar de 32.768, lo que supone una reducción de aproximadamente 57 veces para esa estructura concreta.

La caché puede persistirse en el almacenamiento. Una conversación cerrada puede reabrirse sin repetir todo el procesamiento inicial del contexto, siempre que utilicemos el mismo estado y la misma configuración.

Decodificación especulativa con MTP

GLM-5.2 incluye una cabeza MTP capaz de proponer varios tokens futuros. El modelo principal verifica esas propuestas mediante una ejecución agrupada.

```
Cabeza MTP:
  propone token A
  propone token B
  propone token C

Modelo principal:
  verifica A, B y C

Resultado:
  acepta varios tokens
  o descarta desde el primer error
```

Cuando la tasa de aceptación es elevada, podemos obtener más de un token por cada ejecución completa del modelo. Cuando la tasa es baja, el coste de generar y verificar borradores puede superar el beneficio.

Colibrì utiliza una cabeza MTP almacenada en INT8. El propio proyecto advierte que una cabeza MTP en INT4 puede reducir drásticamente la aceptación de los borradores.

CPU, CUDA y Metal

El núcleo de Colibrì está implementado en C y no necesita Python durante el cálculo principal. El motor puede ejecutarse solamente mediante CPU, pero también dispone de niveles opcionales para GPU.

Actualmente podemos encontrar:

- Ejecución mediante CPU y OpenMP.
- Backend CUDA para tarjetas NVIDIA.
- Almacenamiento de expertos residentes en VRAM.
- Pipeline que mantiene determinados estados en la GPU entre capas.
- Backend Metal experimental para Apple Silicon.
- Distribución NUMA para sistemas con varios sockets.

La frase “sin GPU” significa que la GPU no es obligatoria. No significa que una CPU doméstica vaya a generar rápidamente tokens de un modelo de 744.000 millones de parámetros.

Diferencias principales entre AirLLM y Colibrì

CARACTERÍSTICA	AIRLLM	COLIBRÌ
Objetivo principal	Ejecutar muchos modelos grandes con poca memoria instantánea	Ejecutar modelos MoE gigantes mediante una jerarquía VRAM, RAM y NVMe
Tipo de modelos	Modelos densos y MoE	Modelos MoE
Compatibilidad	Amplia compatibilidad con arquitecturas de Hugging Face	Compatibilidad limitada a modelos implementados expresamente
Modelos principales	Llama, Qwen, DeepSeek, Mistral, Mixtral, Gemma, Phi y otros	GLM-5.2 y OLMoE
Estrategia para modelos densos	Streaming por capas	No es su objetivo

CARACTERÍSTICA	AIRLLM	COLIBRÌ
Estrategia para modelos MoE	Streaming genérico de expertos según la arquitectura	Streaming, caché, prefetch y colocación especializada de expertos
Runtime	Python, PyTorch, Transformers y Hugging Face	Núcleo en C con herramientas auxiliares en Python
Dependencias	Relativamente numerosas	Núcleo de inferencia con dependencias mínimas
Formato de entrada	Modelos de Hugging Face y Safetensors	Contenedor convertido específicamente para Colibrì
Cuantización	Opcional, con compresión de 4 u 8 bits	Modelo principal distribuido en INT4 y cabeza MTP en INT8
Preparación del modelo	Separación y reorganización por capas o expertos	Conversión a un contenedor optimizado para acceso por experto
GPU	Normalmente utilizada, aunque existe soporte para CPU	Opcional, con CPU, CUDA y Metal experimental
Administración de caché	Prefetch de las siguientes partes del modelo	LRU, expertos fijados, registro de uso y niveles VRAM/RAM/NVMe
Uso de varios SSD	No constituye una característica central	Puede distribuir lecturas entre dos copias del modelo
API compatible con OpenAI	No es la función central del proyecto	Incluye servidor compatible con la API de OpenAI
Interfaz web	No constituye su objetivo principal	Incluye panel web y visualización de expertos
Flexibilidad	Alta	Reducida
Especialización	Moderada	Muy alta

AirLLM resulta más parecido a una extensión extrema del ecosistema Hugging Face. Colibrì se parece más a un sistema operativo especializado para colocar pesos de modelos MoE entre diferentes tipos de memoria.

Requisitos reales de almacenamiento y memoria

Una cifra de VRAM reducida puede dar una impresión equivocada sobre los requisitos totales. Si AirLLM ejecuta un modelo de 70.000 millones de parámetros usando 4 GB de VRAM, el modelo sigue necesitando decenas o cientos de gigabytes en el almacenamiento.

Durante la preparación inicial podemos llegar a tener:

```
Modelo original
+
Copia separada por capas
+
Caché de Hugging Face
+
Archivos temporales
```

El espacio necesario dependerá de la precisión original, la compresión elegida y la posibilidad de eliminar los archivos originales después de la conversión.

Colibrì presenta requisitos más concretos para GLM-5.2:

- Aproximadamente 25 GB de RAM como mínimo funcional.
- Aproximadamente 9,9 GB residentes para la parte densa en INT4.
- Unos 372 GB para el contenedor preconvertido.

- Un NVMe rápido, preferiblemente con buen rendimiento sostenido.
- Espacio adicional para cachés, estados, registros y posibles copias.
- Otros 372 GB si queremos mantener una réplica completa en un segundo NVMe.

La conversión desde el modelo FP8 original puede involucrar alrededor de 756 GB de pesos de origen. El conversor de Colibrì procesa los archivos por fragmentos para evitar que necesitemos guardar simultáneamente el modelo FP8 completo y el contenedor final.

Podemos clasificar aproximadamente las configuraciones de Colibrì de la siguiente manera:

CONFIGURACIÓN	FUNCIONAMIENTO ESPERADO
25 GB de RAM y CPU	El modelo funciona, pero depende continuamente del NVMe
64 GB de RAM y CPU	Podemos almacenar más expertos frecuentes y reducir algunas lecturas
128 GB de RAM y CPU	La caché caliente puede proporcionar una mejora importante
GPU con 16 GB de VRAM	Podemos mantener una cantidad limitada de expertos y operaciones en GPU
Varias GPU con mucha VRAM	Podemos acercarnos a la residencia completa de expertos
Dos NVMe rápidos	Podemos sumar parte del ancho de banda de lectura

Instalación básica de AirLLM en Debian

Antes de instalar AirLLM debemos disponer de Python, PyTorch y, si utilizamos una tarjeta NVIDIA, una versión de PyTorch compatible con nuestro controlador y nuestra instalación de CUDA.

```
sudo apt update
sudo apt install -y python3 python3-pip git
python3 -m pip install airllm --break-system-packages
```

Podemos comprobar si PyTorch reconoce correctamente una GPU NVIDIA:

```
python3 -c 'import torch; print("CUDA disponible:", torch.cuda.is_available()); print("Versión CUDA de PyTorch:",
torch.version.cuda)'
```

Para utilizar la compresión de pesos podemos instalar BitsAndBytes:

```
python3 -m pip install --upgrade bitsandbytes --break-system-packages
```

El siguiente script carga Qwen3-32B mediante AutoModel y activa la compresión de 4 bits:

```
#!/usr/bin/env python3

from airllm import AutoModel

cModelo = "Qwen/Qwen3-32B"
cMaxLongitud = 128

vModelo = AutoModel.from_pretrained(
  cModelo,
  compression="4bit"
)

aTextos = [
```

```
  "Explica la diferencia entre un modelo denso y un modelo MoE."
]

dTokens = vModelo.tokenizer(
  aTextos,
  return_tensors="pt",
  return_attention_mask=False,
  truncation=True,
  max_length=cMaxLongitud,
  padding=False
)

vSalida = vModelo.generate(
  dTokens["input_ids"].cuda(),
  max_new_tokens=256,
  use_cache=True,
  return_dict_in_generate=True
)

vTexto = vModelo.tokenizer.decode(
  vSalida.sequences[0],
  skip_special_tokens=True
)

print(vTexto)
```

La primera ejecución puede tardar bastante porque AirLLM tendrá que descargar, revisar y reorganizar el modelo. Debemos vigilar el espacio disponible antes de comenzar.

Podemos indicar una ubicación específica para los fragmentos reorganizados:

```
vModelo = AutoModel.from_pretrained(
  cModelo,
  compression="4bit",
  layer_shards_saving_path="/mnt/nvme/AirLLM/Qwen3-32B"
)
```

En modelos muy recientes también podemos necesitar dependencias específicas. Por ejemplo, algunos modelos exigen determinadas versiones de Transformers, Flash Attention, Compressed Tensors o una versión concreta de CUDA. AirLLM no puede eliminar las restricciones impuestas por el código original del modelo.

Instalación básica de Colibrì en Debian

Podemos utilizar una versión precompilada o compilar Colibrì desde el repositorio. Para compilarlo necesitamos GCC o Clang con soporte para OpenMP.

```
sudo apt update
sudo apt install -y git gcc make libgomp1 python3 python3-pip
git clone https://github.com/JustVugg/colibri
cd colibri/c
./setup.sh
```

El instalador comprueba el compilador, verifica OpenMP, compila el motor y ejecuta pruebas básicas.

Después necesitamos descargar el contenedor preparado para Colibrì o convertir el modelo original. El contenedor recomendado para GLM-5.2 utiliza cuantización INT4 por grupos y una cabeza MTP en INT8.

Podemos iniciar la descarga y la conversión mediante:

```
./coli convert --model /mnt/nvme/Modelos/GLM-5.2-Colibri-INT4
```

Antes de ejecutar el modelo conviene realizar una comprobación completa:

```
COLI_MODEL=/mnt/nvme/Modelos/GLM-5.2-Colibri-INT4 ./coli doctor --deep
```

Podemos consultar cómo distribuirá Colibrì el modelo entre VRAM, RAM y almacenamiento:

```
COLI_MODEL=/mnt/nvme/Modelos/GLM-5.2-Colibri-INT4 ./coli plan
```

Para abrir una conversación interactiva:

```
COLI_MODEL=/mnt/nvme/Modelos/GLM-5.2-Colibri-INT4 ./coli chat
```

Para iniciar el servidor compatible con la API de OpenAI:

```
./coli serve --model /mnt/nvme/Modelos/GLM-5.2-Colibri-INT4
```

También podemos iniciar simultáneamente la API y el panel web:

```
./coli web --model /mnt/nvme/Modelos/GLM-5.2-Colibri-INT4
```

Si disponemos de una segunda copia del modelo en otro NVMe podemos configurar ambas ubicaciones:

```
COLI_MODEL=/mnt/nvme0/GLM-5.2-Colibri-INT4 COLI_MODEL_MIRROR=/mnt/nvme1/GLM-5.2-Colibri-INT4 COLI_DISK_WEIGHTS=9,3
./coli chat
```

El valor 9,3 representa una proporción aproximada entre el ancho de banda de ambas unidades. Si no lo indicamos, Colibrì puede medir el rendimiento durante el arranque.

Rendimiento y cuellos de botella

No podemos comparar directamente los tokens por segundo de AirLLM y Colibrì sin utilizar el mismo modelo, la misma cuantización y el mismo hardware. Además, AirLLM admite modelos muy diferentes, mientras que las cifras principales de Colibrì corresponden a GLM-5.2.

Podemos analizar, no obstante, los cuellos de botella de cada arquitectura.

Cuello de botella de AirLLM con modelos densos

En un modelo denso debemos utilizar todas las capas para generar cada token. Si las capas permanecen en el NVMe, una generación puede requerir leer una cantidad de datos cercana al tamaño completo del modelo.

Supongamos un modelo cuantizado que ocupa 40 GB y una unidad capaz de proporcionar 4 GB/s sostenidos:

```
40 GB de pesos / 4 GB por segundo
=
10 segundos mínimos de lectura por token
```

El resultado teórico sería de aproximadamente 0,1 tokens por segundo antes de añadir el tiempo de cálculo, las sincronizaciones

y otras operaciones.

El prefetch puede ocultar parte de la lectura, pero solamente si el cálculo tarda lo suficiente y el almacenamiento mantiene el ancho de banda esperado.

Cuello de botella de Colibrì

Colibrì evita leer el modelo completo porque solamente carga los expertos seleccionados. Sin embargo, en GLM-5.2 los expertos variables de un token pueden representar alrededor de 11 GB.

Con un ancho de banda real de 1 GB/s:

```
11 GB / 1 GB por segundo
=
11 segundos de lectura

Velocidad máxima aproximada:
1 / 11
=
0,09 tokens por segundo
```

Esta estimación coincide con el orden de magnitud publicado para una configuración fría con 25 GB de RAM: aproximadamente entre 0,05 y 0,1 tokens por segundo.

Cuando los expertos necesarios ya se encuentran en RAM o VRAM, el almacenamiento deja de limitar esa parte de la ejecución. Por eso la temperatura de la caché modifica tanto el rendimiento.

Cifras publicadas por Colibrì

El proyecto ha publicado, entre otras, las siguientes mediciones comunitarias:

HARDWARE	VELOCIDAD APROXIMADA
Equipo de desarrollo con 25 GB de RAM	Entre 0,05 y 0,1 tokens por segundo en frío
Equipo CPU con 128 GB de RAM	Alrededor de 1,8 tokens por segundo con caché caliente
Portátil con una RTX 5070 Ti	Alrededor de 1,07 tokens por segundo
Seis RTX 5090 con residencia completa	Entre 5,8 y 6,8 tokens por segundo

Estas cifras no constituyen una garantía de rendimiento. Proceden del propio proyecto y de pruebas comunitarias realizadas con configuraciones diferentes. La velocidad depende del procesador, la RAM, el número de canales de memoria, la GPU, el NVMe, la temperatura de la caché, el contexto, el prompt, la configuración especulativa y los expertos seleccionados.

Tiempo hasta el primer token

Los tokens por segundo no describen por completo la experiencia. También debemos medir el tiempo hasta el primer token.

El procesamiento inicial del prompt, conocido como prefill, puede ser muy costoso. Si enviamos un contexto largo, el motor debe procesar todos los tokens antes de comenzar la generación.

Una configuración puede generar razonablemente rápido después de calentar la caché, pero tardar mucho en responder por primera vez. En servidores interactivos debemos controlar al menos:

- Tiempo de carga del modelo.
- Tiempo hasta el primer token.
- Velocidad de procesamiento del prompt.

- Velocidad de generación.
- Lecturas desde el almacenamiento por token.
- Porcentaje de aciertos de la caché.
- Uso de RAM y VRAM.
- Tasa de aceptación de la decodificación especulativa.

AirLLM, Colibrì y llama.cpp

AirLLM y Colibrì no sustituyen automáticamente a llama.cpp. Los tres proyectos resuelven problemas relacionados, pero están optimizados para situaciones diferentes.

llama.cpp utiliza modelos GGUF y ofrece kernels altamente optimizados para CPU, CUDA, HIP, Vulkan, Metal y otras plataformas. También permite dividir el modelo entre CPU y GPU.

Cuando un modelo cabe completamente en la suma de nuestra RAM y nuestra VRAM, llama.cpp suele ser la opción más práctica:

- Tiene una compatibilidad de hardware muy amplia.
- Ofrece numerosas cuantizaciones.
- Incluye herramientas de consola y servidor.
- Dispone de una API compatible con OpenAI.
- Puede utilizar CPU y GPU simultáneamente.
- No necesita leer repetidamente todas las capas desde el almacenamiento si el modelo permanece residente.

AirLLM comienza a tener sentido cuando el modelo no cabe en la memoria disponible y aceptamos sacrificar velocidad para poder ejecutarlo.

Colibrì comienza a tener sentido cuando queremos ejecutar un modelo MoE gigantesco específicamente soportado y podemos proporcionar cientos de gigabytes de NVMe, una cantidad razonable de RAM y suficiente paciencia.

SITUACIÓN	OPCIÓN MÁS RAZONABLE
El modelo GGUF cabe completamente en RAM y VRAM	llama.cpp
Queremos máxima compatibilidad con hardware doméstico	llama.cpp
Queremos ejecutar un modelo Hugging Face demasiado grande para nuestra memoria	AirLLM
Queremos probar muchas arquitecturas diferentes	AirLLM
Queremos mantener la precisión original del modelo y aceptamos mucha entrada y salida	AirLLM sin compresión
Queremos ejecutar GLM-5.2 con unos 25 GB de RAM	Colibrì
Disponemos de uno o dos NVMe rápidos y queremos aprovechar la dispersión MoE	Colibrì
Queremos estudiar visualmente la activación de expertos	Colibrì
Necesitamos respuestas interactivas rápidas en hardware modesto	Un modelo más pequeño ejecutado con llama.cpp

Limitaciones que debemos tener presentes

Una cifra baja de VRAM no representa el consumo total. AirLLM puede utilizar pocos gigabytes de VRAM mientras consume cientos de gigabytes de almacenamiento. Colibrì puede ejecutar GLM-5.2 con 25 GB de RAM, pero necesita

aproximadamente 372 GB para el modelo cuantizado.

Ejecutar un modelo no significa utilizarlo cómodamente. Una velocidad de 0,05 tokens por segundo equivale a un token cada 20 segundos. Una respuesta de 100 tokens podría tardar más de media hora.

El rendimiento del NVMe anunciado por el fabricante no siempre coincide con el rendimiento sostenido. Las cifras máximas suelen corresponder a lecturas secuenciales, colas determinadas y cachés internas. Las lecturas reales de expertos pueden producir patrones diferentes y provocar calentamiento o reducción térmica.

Más RAM puede ser más útil que una GPU pequeña. En Colibrì, aumentar la RAM permite mantener más expertos en caché y evitar lecturas desde el almacenamiento. Una GPU de 8 o 16 GB puede acelerar algunas operaciones, pero no puede almacenar por sí sola cientos de gigabytes de expertos.

La cuantización afecta a la calidad. AirLLM puede evitarla, pero entonces aumenta el volumen de datos. Colibrì utiliza un contenedor INT4 para GLM-5.2. El hecho de que el motor mantenga los mismos pesos y decisiones del router independientemente del nivel de memoria no convierte esos pesos INT4 en equivalentes al modelo FP8 original.

La compatibilidad de AirLLM depende del ecosistema Python. Un cambio en Transformers, PyTorch, CUDA, Flash Attention o en el código remoto del modelo puede romper una configuración que funcionaba anteriormente.

La compatibilidad de Colibrì depende de implementaciones específicas. Aunque su estrategia general podría aplicarse a muchos modelos MoE, cada arquitectura necesita soporte para sus tensores, atención, router, tokenizador, formato y operadores.

Las cachés mejoran cargas repetitivas, no eliminan el peor caso. Si una consulta activa expertos poco frecuentes, Colibrì tendrá que leerlos desde el almacenamiento. Una caché caliente para programación puede no estar caliente para derecho, poesía, matemáticas o chino.

Los contextos largos continúan consumiendo memoria y tiempo. Reducir los pesos residentes no elimina la caché KV, el procesamiento del prompt ni el coste de la atención.

En términos prácticos, AirLLM es la alternativa más flexible cuando nuestro objetivo es demostrar que un modelo muy grande puede ejecutarse en hardware con poca VRAM. Colibrì es la alternativa más sofisticada cuando queremos explotar específicamente la estructura de un modelo MoE gigantesco y tratar el almacenamiento como un nivel activo de memoria.

Para un uso diario, normalmente obtendremos una experiencia mejor seleccionando un modelo más pequeño que pueda permanecer en RAM o VRAM. Para investigación, experimentación, análisis de arquitecturas MoE o ejecución local de modelos que de otro modo serían completamente inaccesibles, AirLLM y Colibrì abren posibilidades que un motor de inferencia convencional no puede ofrecer.