CAGE – Grupo de Investigación

@sc3uis

SC3 – Super Cómputo y Cálculo Científico – UIS

Modelos de lenguaje, Transformers y PyTorch

Cómo funcionan los LLM, qué son sus capas y cómo los soporta PyTorch

Módulo 9 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

- ➔ Explicar qué es un **modelo de lenguaje** y por qué predecir el siguiente *token* basta para generar texto.
- ➔ Entender la capa que cambió todo: la **atención** (*self-attention*), programada desde cero en C++.
- ➔ Conocer el **bloque Transformer** y las capas modernas: *multi-head*, RoPE, GQA, MoE, FlashAttention.
- ➔ Escribir y entrenar un **mini GPT** en **PyTorch** y comparar con lo hecho a mano.
- ➔ Ver cómo PyTorch se apoya en **C++, OpenMP, BLAS, CUDA/ROCm y MPI/NCCL**: todo lo visto en el semillero.
- ➔ Calcular **FLOPs, memoria y comunicación** de un LLM: la IA como problema de HPC.

Agenda

1. Modelos de lenguaje: *tokens*, *embeddings* y generación
2. La atención: la capa que cambió todo
3. El bloque Transformer y las capas modernas
4. PyTorch: un mini GPT completo
5. PyTorch por dentro: C++, OpenMP, CUDA, ROCm y MPI
6. Escala: FLOPs, memoria, *prefill* y *decode*
7. Retos finales y cierre del semillero

1

Modelos de lenguaje: tokens, embeddings y generación

¿Qué es un modelo de lenguaje?

- Un **modelo de lenguaje** asigna una probabilidad a cada posible **siguiente *token*** dado el texto anterior:

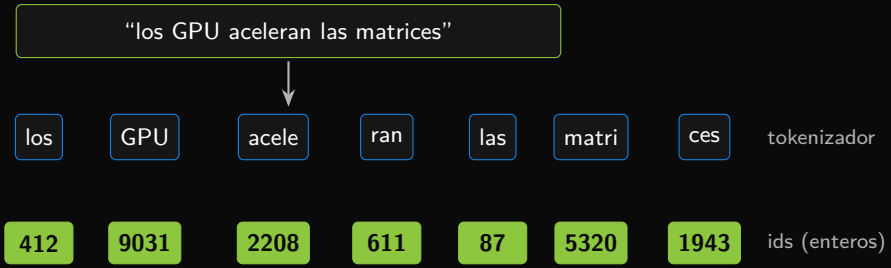
$$p(x_t \mid x_1, \ldots, x_{t-1})$$

- Se **entrena** con la misma *loss* del módulo 8 (entropía cruzada): maximizar la probabilidad del token correcto en un corpus enorme.
- Para **generar**: se muestrea un token, se agrega a la entrada y se repite.

★ La idea en una línea

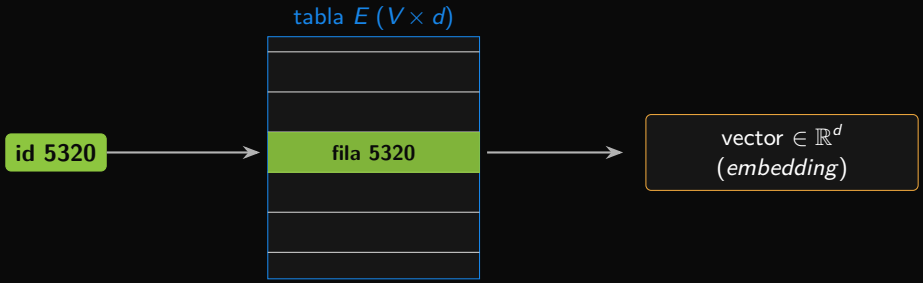
Un LLM es una red neuronal (como la del módulo 8) que **clasifica** entre decenas de miles de "clases": los *tokens* del vocabulario. Chatear es **clasificar muchas veces seguidas**.

Tokens: del texto a números



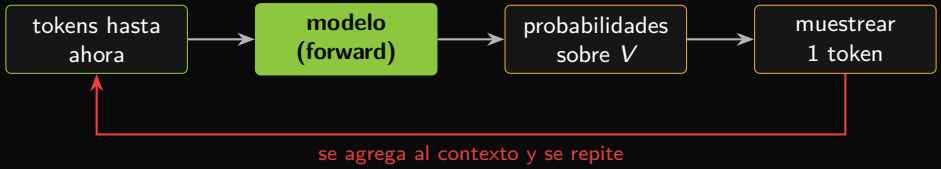
- Los **tokens** son fragmentos de texto (palabras, partes de palabras, símbolos). Un tokenizador (p. ej. BPE) los obtiene de los datos.
- El modelo **solo ve enteros**. Ejemplo **ilustrativo**: los cortes y los ids reales dependen del tokenizador.
- Vocabularios típicos: decenas de miles de tokens; nuestro mini GPT usa **caracteres** (23 símbolos).

Embeddings: cada token es un vector



- El *embedding* es **una tabla de consulta**: la fila del id. Es una matriz de parámetros **aprendida** por *backpropagation*.
- Tokens con uso parecido terminan con vectores parecidos: el significado queda en la **geometría**.
- A la entrada se suma una **posición** (qué lugar ocupa el token); lo veremos más adelante.

El bucle de generación (autoregresivo)



- **Una pasada completa del modelo por cada token generado**: por eso generar es costoso y secuencial.
- Detalle HPC: para no recalcular el pasado en cada paso se guardan las claves y valores (*KV cache*); lo veremos en la sección de escala.
- El texto generado **no es un cálculo distinto**: es el *forward* del módulo 8, repetido.

Muestreo y temperatura

$$p_i = \frac{e^{z_i/\tau}}{\sum_j e^{z_j/\tau}}$$

- z: los *logits* que produce el modelo; τ: **temperatura**.
- $\tau \to 0$: casi siempre el token más probable (*greedy*), texto repetitivo y seguro.
- $\tau = 1$: la distribución del modelo.
- $\tau > 1$: más variedad y más errores.

- **top-k** y **top-p**: se descartan los tokens poco probables antes de muestrear.
- Es la misma *softmax* del módulo 8, con un divisor.
- Con la misma entrada el modelo es **determinista**; la aleatoriedad la pone el **muestreo**.

```
p = F.softmax(logits / 0.8, dim=-1)
tok = torch.multinomial(p, 1)
```

El modelo de lenguaje más pequeño: bigramas en C++

```cpp
std::vector<double> cnt(V * V, 1.0);          // cnt[a][b]: veces que b sigue a a
for (size_t i = 0; i + 1 < texto.size(); ++i)
    cnt[id[texto[i]] * V + id[texto[i+1]]] += 1;
for (int a = 0; a < V; ++a) {                 // normalizar cada fila: p(b | a)
    double s = 0; for (int b = 0; b < V; ++b) s += cnt[a*V+b];
    for (int b = 0; b < V; ++b) cnt[a*V+b] /= s;
}
// generar: muestrear segun la fila del caracter actual
double r = rand() / (double)RAND_MAX, acc = 0; int b = 0;
for (; b < V - 1; ++b) { acc += cnt[a*V+b]; if (r < acc) break; }
```

```
vocabulario 22, loss 2.292 (azar: ln(V) = 3.091)
ldctrtod.fulesrhyl les ldobaac.bbs cten cl p yal.yo lecpau oelbusotresndcls y.enaaoafm
```

Es **un modelo de lenguaje real**: una tabla $V \times V$. Mejora el azar (2.29 vs 3.09), pero solo mira **un carácter** de contexto: genera ruido.

El límite: el contexto

- El bigrama usa **1 token** de contexto. Para generar texto coherente hay que mirar **cientos o miles**.
- Una tabla con todos los contextos posibles **explota**: V^n entradas para n tokens.
- Hace falta una función **compacta y aprendida** que resuma el contexto en un vector: una **red neuronal**.
- Primeras soluciones: MLP con ventana fija; redes recurrentes (RNN/LSTM), que leen **un token a la vez** (difíciles de paralelizar y de entrenar con textos largos).

★ La pregunta

¿Cómo combinar información de **todos** los tokens anteriores, **en paralelo**, con parámetros aprendidos? Respuesta de 2017: la **atención** (Vaswani et al., *Attention Is All You Need*).

¿Por qué funciona?

- **Predecir bien el siguiente token obliga a modelar el mundo que describe el texto**: gramática, hechos, estilo, razonamiento aparente. Es una tarea de *self-supervised learning*: los datos traen su propia etiqueta.
- **Representaciones**: las capas construyen vectores cada vez más abstractos (*embedding* $\rightarrow$ sintaxis $\rightarrow$ significado).
- **Escala**: la *loss* baja de forma **predecible** al aumentar parámetros, datos y cómputo (*scaling laws*; Kaplan et al., 2020; Hoffmann et al., 2022).
- **Hardware**: la arquitectura es casi toda GEMM, ideal para GPU; por eso fue posible escalarla.

❄ Honestidad científica

Sabemos **cómo se calcula** y **cómo se entrena**; entender **por qué** emergen ciertas capacidades sigue siendo investigación abierta (*interpretability*). Un LLM puede equivocarse con seguridad: se verifica.

Reto individual 1: tu propio modelo de lenguaje

■ Reto 1 – Bigramas y trigramas

1. Compila y ejecuta `bigrama.cpp`. Cambia el texto por un párrafo más largo (por ejemplo, la documentación del cluster).
2. Mide la *loss* (entropía cruzada) en un texto **que no usaste** para contar. ¿Qué pasa con y sin suavizado?
3. Extiéndelo a **trigramas** (contexto de 2 caracteres). ¿Mejora la *loss*? ¿Qué pasa con el tamaño de la tabla?
4. Implementa **temperatura** al muestrear y compara el texto con $\tau = 0.5$ y $\tau = 1.5$.

2
La atención: la capa que cambió todo

Query, Key, Value: la idea

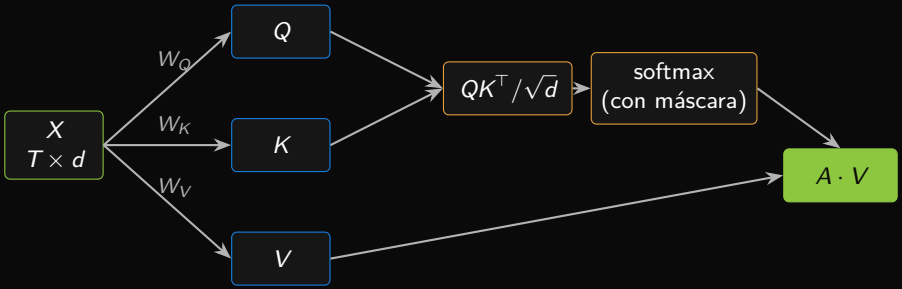
- Cada token i produce tres vectores con tres matrices aprendidas: $q_i = x_i W_Q$, $k_i = x_i W_K$, $v_i = x_i W_V$.
- **Query**: "qué estoy buscando". **Key**: "qué ofrezco". **Value**: "qué contenido entrego".
- La **afinidad** entre i y j es el producto punto $q_i \cdot k_j$.
- Los pesos de atención son un *softmax* de esas afinidades; la salida de i es la **suma ponderada** de los v_j.

★ En una fórmula

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V$$

Dos GEMM y una *softmax*. $\sqrt{d}$ evita que los productos punto crezcan con la dimensión.

Atención paso a paso



- Tres GEMM de proyección (XW_Q, XW_K, XW_V), una GEMM $T \times T$ ($QK^\top$), *softmax* por filas y otra GEMM (AV).
- La matriz de atención A es de $T \times T$: el costo crece con el **cuadrado** de la longitud del contexto.

Atención desde cero en C++

```
gemm(X, Wq, Q, T, d, d);  gemm(X, Wk, K, T, d, d);  gemm(X, Wv, V, T, d, d);      // Q, K, V = X W
for (int i = 0; i < T; ++i) {
    float mx = -1e30f, sum = 0;
    for (int j = 0; j < T; ++j) {
        float s = 0; for (int p = 0; p < d; ++p) s += Q[i*d+p] * K[j*d+p];        // Q K^T
        S[i*T+j] = (j <= i) ? s / std::sqrt((float)d) : -1e30f;                   // escala + mascara
        if (S[i*T+j] > mx) mx = S[i*T+j];
    }
    for (int j = 0; j < T; ++j) { S[i*T+j] = std::exp(S[i*T+j] - mx); sum += S[i*T+j]; }
    for (int j = 0; j < T; ++j) S[i*T+j] /= sum;                                  // softmax por fila
}
gemm(S, V, O, T, d, T);                                                           // salida = A V
```

Con $T = 5$ tokens y $d = 4$ (valores deterministas, `atencion.cpp`). Es exactamente la fórmula; **no hay nada más**. La *softmax* usa el truco de restar el máximo (módulo 8).

La matriz de atención

pesos de atención A (calculados con `atencion.cpp`)



Cada fila suma 1.

La mitad superior es 0: **máscara causal** (un token no ve el futuro).

El token t_0 solo se mira a sí mismo.

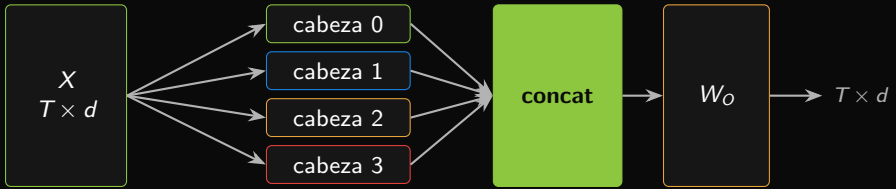
Verificación: C++ y PyTorch dan lo mismo

```
Q, K, V = X @ Wq, X @ Wk, X @ Wv
S = (Q @ K.T) / math.sqrt(d)
S = S.masked_fill(torch.triu(torch.ones(T, T).bool(), 1), float('-inf'))
A = torch.softmax(S, dim=-1)
O = A @ V
O2 = F.scaled_dot_product_attention(Q[None], K[None], V[None], is_causal=True)[0]
```

```
$ ./atencion                 O fila 4: 0.0049 -0.0174 -0.0204 -0.0007
$ python3 atencion_torch.py   O fila 4: 0.0049 -0.0174 -0.0204 -0.0007
max |O - sdpa| = 5.96e-08     # la funcion fusionada de PyTorch coincide
```

- ➔ Las **15 entradas no nulas de** A (25 con la máscara) y la salida O coinciden. PyTorch no hace magia: hace **esto**, con kernels optimizados.
- ➔ `scaled_dot_product_attention` es la versión **fusionada** (en GPU puede usar FlashAttention).

Multi-head attention



- Se parte d en h cabezas de tamaño d/h: cada una atiende a **algo distinto** (sintaxis, posición, referencias…).
- Cálculo **igual** que una sola cabeza, pero con h **GEMM independientes** (por *batch*): paralelismo gratis para la GPU.
- El costo en FLOPs es el mismo que con una cabeza de tamaño d.

Posición: la atención no sabe el orden

- La atención trata el contexto como un **conjunto**: si permutas los tokens, las afinidades son las mismas. Hay que **inyectar la posición**.
- **Sinusoidal** (Transformer original) y **posiciones aprendidas** (nuestro mini GPT): se **suma** un vector de posición al *embedding*.
- **RoPE** (Su et al., 2021): **rota** q y k según la posición; así $q_i \cdot k_j$ depende de la **distancia** $i - j$. Es la opción de la mayoría de LLM actuales.
- Extender el contexto (4K $\rightarrow$ 128K tokens) es en gran parte **un problema de posiciones y de memoria**.

Reto individual 2: atención con tus manos

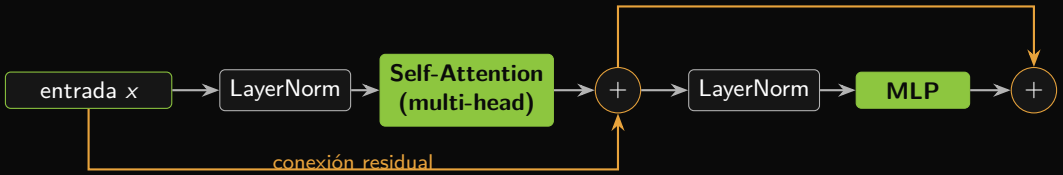
■ Reto 2 – Atención en C++

1. Compila y ejecuta `atencion.cpp` y `atencion_torch.py`; verifica que A y O coinciden.
2. Quita la **máscara causal**: ¿cómo cambia A? ¿Qué tipo de modelo sería (pista: BERT vs GPT)?
3. Quita el factor $1/\sqrt{d}$ con $d = 64$ y valores grandes: ¿qué le pasa a la *softmax*?
4. Implementa **2 cabezas** ($d = 4$, cada una de tamaño 2) y concatena. Verifica contra `nn.MultiheadAttention` o contra tu versión de PyTorch.

3

El bloque Transformer y las capas modernas

El bloque Transformer



- Un bloque: **atención** (mezcla información **entre tokens**) + **MLP** (procesa **cada token** por separado), cada uno con *LayerNorm* y suma residual.
- Un LLM **apila decenas de bloques idénticos** (cada uno con sus propios pesos).

Residual y normalización: por qué están ahí

Conexión residual: $y = x + f(x)$

- ➔ El gradiente tiene un **camino directo** hacia atrás: permite entrenar con decenas o cientos de capas.
- ➔ Cada capa solo **corrige** la representación.

LayerNorm / RMSNorm

- ➔ Normaliza cada vector (media y varianza; RMSNorm solo la escala): mantiene los valores en un **rango estable**.
- ➔ Son operaciones **memory bound**: pocas operaciones por byte (módulo 8). Por eso se **fusionan** con otras.

En el módulo 8, $dA_1 = dZ_2 W_2^\top$ ya mostraba el patrón: el *backward* reutiliza lo del *forward*. Aquí *autograd* lo hace por ti.

El MLP: dos GEMM

$$\mathrm{MLP}(x) = W_2\,\phi(W_1 x + b_1) + b_2$$

- Dimensión oculta típica: $4d$ (o $\approx \frac{8}{3}d$ con SwiGLU).
- ϕ: **GELU** (GPT) o **SwiGLU** (LLaMA y similares).
- Es **la misma capa del módulo 8** (tanh $\rightarrow$ GELU), con más ancho.
- Contiene **dos tercios** de los parámetros del bloque.

★ Reparto de parámetros

Por bloque con d de ancho: atención $\approx 4d^2$ (W_Q, W_K, W_V, W_O) y MLP $\approx 8d^2$ ($d\times 4d$ y $4d\times d$): $\approx 12\,d^2$ **por bloque**.

Un LLM completo (decoder-only)



logits sobre V tokens
$\rightarrow$ softmax $\rightarrow$ siguiente token

N bloques idénticos
(7B: $N = 32$; 70B: $N = 80$)

tabla de $V \times d$ parámetros

Contar parámetros

$$P \approx \underbrace{N \cdot 12\, d^2}_{\text{bloques}} + \underbrace{V d}_{\text{embedding}}$$

Mini GPT ($d = 64$, $N = 2$, $V = 23$, contexto 48): PyTorch informa **106 112** parámetros. Con la cuenta exacta (atención, MLP, LayerNorm, posiciones) **coincide**.

Modelo "de 7B" (tipo LLaMA: $d = 4096$, $N = 32$, $V = 32000$):

$$32 \cdot 12 \cdot 4096^2 + 32000 \cdot 4096 \approx 6.6 \times 10^9$$

★ Orden de magnitud

La fórmula $12\, d^2 N$ da el tamaño **aproximado**; las variantes reales (SwiGLU, GQA, *embeddings* sin compartir) cambian unos puntos porcentuales.

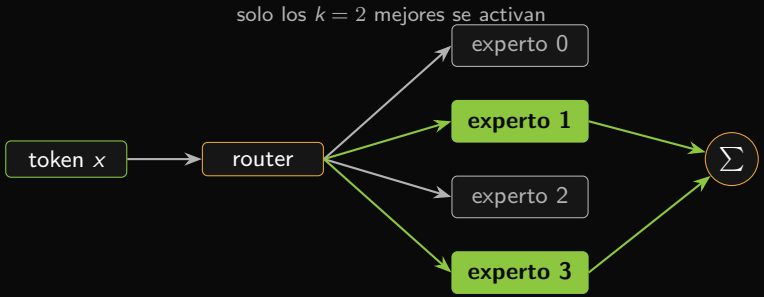
Las capas “nuevas”: qué problema resuelve cada una

Capa	Qué cambia	Problema que ataca
RoPE	rota q y k según la posición	posiciones relativas y contextos largos
GQA / MQA	varias cabezas de q comparten k y v	memoria del *KV cache*
FlashAttention	calcula la atención por bloques sin guardar A	*memory bound* en $T \times T$
MoE	muchos MLP (expertos); cada token usa solo k	más parámetros sin que los FLOPs crezcan con E
SSM (Mamba)	recurrencia aprendida, costo lineal en T	costo cuadrático de la atención
LoRA	entrena matrices de bajo rango sobre pesos fijos	*fine-tuning* barato

Su 2021 (RoPE); Ainslie 2023 (GQA); Dao 2022 (FlashAttention); Shazeer 2017 y Fedus 2021 (MoE); Gu y Dao 2023 (Mamba); Hu 2021 (LoRA).

Ver referencias.

Mixture of Experts (MoE)



- El MLP se reemplaza por E **MLP** (expertos) y un *router* elige k por token: parámetros $\times E$, cómputo por token $\times k$ (frente a un MLP denso).
- Reto HPC: los expertos viven en **GPU distintas**; los tokens viajan con `all-to-all` (una colectiva análoga a las de MPI, módulo 5). Aparece el *load imbalance*.

FlashAttention: un problema de jerarquía de memoria

- La atención ingenua escribe S y A ($T \times T$) en **memoria global** y las vuelve a leer: **memory bound**.
- FlashAttention (Dao et al., 2022) procesa Q, K, V por **tiles** en *shared memory*, con *softmax* **en línea**, y **nunca guarda** A.
- Mismo resultado exacto, menos tráfico a memoria global: es **el *tiling* del matmul del módulo 6** aplicado a la atención.

★ Lección HPC

Dos algoritmos con los **mismos FLOPs** pueden tener rendimientos muy distintos según cuántos **bytes** muevan. Lo que decide es la **intensidad aritmética**, no el conteo de operaciones.

Reto individual 3: dimensiona un Transformer

■ Reto 3 – Cuentas de un LLM

1. Con $P \approx 12\,d^2 N + Vd$, calcula los parámetros de $(d, N) = (768, 12)$, $(2048, 24)$ y $(4096, 32)$ con $V = 32000$.
2. Para $T = 2048$ y $d = 4096$, ¿cuántos bytes ocupa la matriz A de **una** cabeza en FP32? ¿Y de las 32 cabezas?
3. Compara el costo de $QK^\top$ ($\sim 2T^2 d$ FLOP) con el de las proyecciones ($\sim 8Td^2$). ¿A partir de qué T domina la atención?
4. Con $E = 8$ expertos y $k = 2$, ¿cuántos parámetros y cuántos FLOPs por token tiene el MLP respecto a uno denso?

4
PyTorch: un mini GPT completo

Tensores y autograd: tu backward, automático

```
x = torch.randn(4096, 64)
W1 = torch.randn(64, 128, requires_grad=True)       # registra las operaciones
W2 = torch.randn(128, 4, requires_grad=True)
loss = F.cross_entropy(torch.tanh(x @ W1) @ W2, y)  # forward: crea el grafo
loss.backward()                                     # backward: regla de la
    cadena
print(W1.grad.shape, W2.grad.shape)                 # dW1, dW2: lo que hicimos a
    mano
```

- ➔ `backward()` recorre el grafo **al revés** y aplica la derivada de cada operación: las fórmulas de la tabla del módulo 8, automatizadas (*autograd*).
- ➔ `nn.Linear` es una GEMM más sesgo: $XW + b$. `nn.Sequential` apila capas.
- ➔ `opt.step()` es el *gradient descent* (SGD, Adam…) que escribimos en cuatro líneas.

Mini GPT (1/2): el bloque

```
class Bloque(nn.Module):
    def __init__(s):
        super().__init__()
        s.ln1, s.ln2 = nn.LayerNorm(D), nn.LayerNorm(D)
        s.qkv, s.proj = nn.Linear(D, 3 * D), nn.Linear(D, D)              # Q,K,V juntas = 1 GEMM
        s.mlp = nn.Sequential(nn.Linear(D, 4 * D), nn.GELU(), nn.Linear(4 * D, D))
    def forward(s, x):
        b, t, _ = x.shape
        q, k, v = s.qkv(s.ln1(x)).split(D, dim=-1)
        q, k, v = (z.view(b, t, H, D // H).transpose(1, 2) for z in (q, k, v))   # cabezas
        a = F.scaled_dot_product_attention(q, k, v, is_causal=True)            # softmax(QK^T/sqrt(d))V
        x = x + s.proj(a.transpose(1, 2).reshape(b, t, D))                     # residual 1
        return x + s.mlp(s.ln2(x))                                             # residual 2
```

Es **el diagrama del bloque Transformer** escrito en 11 líneas: *LayerNorm*, atención *multi-head*, residual, MLP, residual.

Mini GPT (2/2): el modelo y el entrenamiento

```
class MiniGPT(nn.Module):
    def __init__(s):
        super().__init__()
        s.tok, s.pos = nn.Embedding(V, D), nn.Embedding(T, D)
        s.bloques = nn.Sequential(*[Bloque() for _ in range(L)])
        s.ln, s.out = nn.LayerNorm(D), nn.Linear(D, V, bias=False)
    def forward(s, idx):
        x = s.tok(idx) + s.pos(torch.arange(idx.shape[1]))
        return s.out(s.ln(s.bloques(x)))                          # logits [b, t, V]

m = MiniGPT(); opt = torch.optim.AdamW(m.parameters(), lr=3e-3)
for paso in range(301):
    x, y = batch()
    loss = F.cross_entropy(m(x).view(-1, V), y.view(-1))          # siguiente token
    opt.zero_grad(); loss.backward(); opt.step()                  # backward + AdamW
```

$D = 64$, $H = 4$ cabezas, $L = 2$ bloques, contexto $T = 48$, *batch* 32. Entrena en **segundos en CPU**.

Entrenamiento y generación

```
$ python3 mini_gpt.py
parametros: 106112  (azar: loss = ln V = 3.135)
paso   0  loss 3.238
paso 100  loss 0.842
paso 200  loss 0.139
paso 300  loss 0.109
tiempo: 6.3 s  (73304 tokens/s)
l planificador decide cuando se ejecuta. los procesos se comuntenren mo alabajo ...
```

- ➔ La *loss* cae de $\approx \ln V$ (azar) a 0.11. El texto generado **imita** el corpus.
- ➔ Cuidado: el corpus es **diminuto y repetido**; el modelo **memoriza** (*overfitting*). Un LLM real ve billones de tokens distintos.
- ➔ Ejemplo en una máquina de 2 núcleos: **mide** en guane y en una GPU.

Reto individual 4: entrena y explora

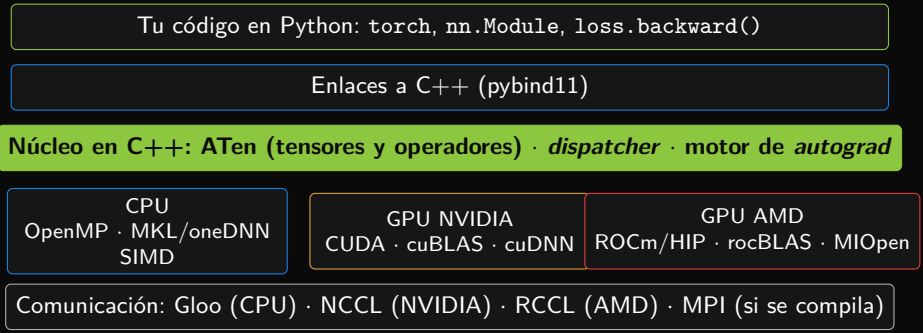
■ Reto 4 – Tu mini GPT

1. Ejecuta `mini_gpt.py`. Cambia D, H, L y el contexto T: tabla de parámetros, *loss* y *tokens/s*.
2. Entrénalo con **otro texto** (por ejemplo la documentación de SC3) y separa un 10 % para validación. ¿Cuándo empieza el *overfitting*?
3. Prueba temperaturas 0.3, 0.8 y 1.5 en la generación y describe qué cambia.
4. Cambia el *embedding* de posición aprendido por uno **sinusoidal**. ¿Cambia la *loss*?

5

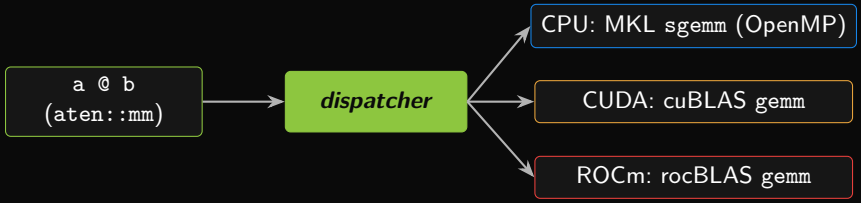
PyTorch por dentro: C++, OpenMP, CUDA, ROCm y MPI

La pila de PyTorch



Python es solo la **interfaz**. El **trabajo numérico** corre en C++/CUDA/HIP/BLAS: los mismos bloques que usaste en los módulos 3 a 8.

Una operación, muchos kernels: el dispatcher



- El mismo código Python (`a @ b`) llama a **kernels distintos** según el **dispositivo** del tensor (`cpu`, `cuda`) y su tipo (`float32`, `bfloat16`).
- Cambiar de CPU a GPU es `.to("cuda")`: el *dispatcher* elige la implementación.
- En una compilación para AMD (ROCm) la API sigue siendo `torch.cuda`: usa HIP por debajo (módulo 7).

Perfilando un paso de entrenamiento (CPU)

```
hilos intra-op (OpenMP): 2 | CUDA disponible: False
Name                                        Self CPU %   Self CPU   # of Calls
aten::mm                                        18.78%    3.833ms           19
aten::addmm                                      7.20%    1.470ms            8
Optimizer.step#AdamW.step                        7.07%    1.444ms            1
aten::_scaled_dot_product_flash_atten...         6.71%    1.369ms            2
aten::_scaled_dot_product_flash_atten...         6.11%    1.246ms            2   (backward)
aten::gelu_backward                              4.35%    0.888ms            2
aten::index_select                               3.87%    0.791ms            2
aten::embedding                                  3.65%    0.746ms            2
Self CPU time total: 20.413ms
```

- ➔ `aten::mm` y `aten::addmm` son **GEMM** (forward y backward): el **cuello de botella** de siempre. Los nombres son los operadores de C++.
- ➔ Hay `_backward` para cada operación: *autograd* ejecutó el *backward* que programamos a mano.
- ➔ Perfil con `torch.profiler` (`perfil_torch.py`); este es un paso de ejemplo en CPU de 2 núcleos.

¿Con qué está compilado? (torch.__config__)

```
$ python3 -c "import torch; print(torch.__config__.show())"
  - GCC 13.3    - C++ Version: 202002
  - Intel(R) oneAPI Math Kernel Library Version 2024.2    # BLAS de CPU
  - Intel(R) MKL-DNN v3.12.0                               # oneDNN
  - OpenMP 201511 (a.k.a. OpenMP 4.5)                      # hilos intra-operador
  BLAS_INFO=mkl, CUDA_VERSION=13.0, CUDNN_VERSION=9.24.0
  USE_CUDA=1, USE_NCCL=ON, USE_GLOO=ON, USE_MPI=OFF, USE_OPENMP=ON, USE_ROCM=OFF
```

- ➔ Salida real de la distribución `pip` usada para preparar este módulo: **C++20 con GCC**, **OpenMP**, **MKL**, **CUDA/cuDNN**, **NCCL** y **Gloo**.
- ➔ `USE_MPI=OFF`: el MPI de PyTorch solo existe si se **compila desde fuente** contra tu OpenMPI; los *wheels* usan NCCL/Gloo.
- ➔ Revisa lo mismo en el PyTorch que instales en el cluster.

En CPU: OpenMP y BLAS (módulos 3 y 4)

- Cada operador grande (GEMM, convolución) usa un *thread pool* basado en **OpenMP**: `torch.get_num_threads()`, `torch.set_num_threads(n)`, `OMP_NUM_THREADS`.
- La GEMM la hace una biblioteca BLAS (MKL u OpenBLAS): bloques, **SIMD** (AVX2/AVX-512) y *multi-threading*: mucho más rápida que nuestros tres bucles.
- Mismas reglas que en el módulo 4: en un nodo del cluster **fija** los hilos a los *cores* reservados por SLURM (`--cpus-per-task`) y `OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK`.
- Muchos operadores pequeños **no** escalan: Amdahl, *false sharing* y *memory bound* aplican igual.

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
python3 mini_gpt.py
```

En GPU: asincronía, streams y memoria (módulo 6)

```
x = torch.randn(8192, 8192, device="cuda")          # cudaMalloc (con cache)
t0 = torch.cuda.Event(enable_timing=True)
t1 = torch.cuda.Event(enable_timing=True)
t0.record()
y = x @ x                                           # lanza el kernel y vuelve (
    asincrono)
t1.record()
torch.cuda.synchronize()                            # como cudaDeviceSynchronize
print(t0.elapsed_time(t1), "ms")                    # cudaEventElapsedTime
```

- ➔ Los operadores de GPU son **asíncronos**: Python encola *kernels* y sigue. Medir sin `synchronize()` da tiempos **falsos**.
- ➔ **Streams** y **eventos** de PyTorch son los de CUDA; la copia a GPU es más rápida con memoria *pinned* (`pin_memory=True`).
- ➔ PyTorch usa un **caching allocator**: no llama a `cudaMalloc` en cada tensor. `nvidia-smi` ve memoria reservada, no la que está en uso.

Código ilustrativo: ejecútalo en `gpu_titan` (aquí no hay GPU).

Fusión de kernels: torch.compile y Triton

- ➔ Cada operación elemental (+, GELU, LayerNorm) es un *kernel* que **lee y escribe** memoria global: **memory bound**.
- ➔ `torch.compile(model)` captura el grafo y **fusiona** operaciones: un solo *kernel* para varias.
- ➔ En GPU genera **Triton** (un lenguaje para escribir *kernels* por bloques que se compila a código de GPU); en CPU genera **C++ con OpenMP**.

```
modelo = MiniGPT().cuda()
modelo = torch.compile(modelo)   # fusiona kernels
# misma llamada de siempre:
logits = modelo(x)
```

★ Mismo principio

Es la **fusión** de `bias+tanh` del módulo 8, hecha por un compilador.

Distribuido: DDP es el allreduce del módulo 8

```
dist.init_process_group("gloo", rank=rank, world_size=size)    # "nccl" en GPU
m = DDP(modelo())                                              # copia del modelo en cada rank
X1, y1 = X[rank*n:(rank+1)*n], y[rank*n:(rank+1)*n]            # cada rank: su shard
F.cross_entropy(m(X1), y1).backward()                          # backward lanza el allreduce
opt.step()                                                     # misma actualizacion en todos
```

```
$ python3 ddp_gloo.py        # 2 procesos en CPU (gloo) vs. entrenamiento secuencial
max |w_ddp - w_secuencial| = 8.94e-08
```

- ➔ Es **exactamente** el *data parallelism* con `MPI_Allreduce` (módulo 8): cada *rank* calcula su gradiente y se suman. El resultado coincide con el secuencial.
- ➔ DDP hace *overlap* del *allreduce* con el *backward*: va enviando los gradientes por *buckets* mientras calcula las capas anteriores.

Backends de comunicación

Backend	Dónde	Comentario
`gloo`	CPU (y GPU básica)	Viene en los *wheels*; sirve para probar y para CPU
`nccl`	GPU NVIDIA	NVLink + InfiniBand; el estándar en GPU
`nccl` (ROCm)	GPU AMD (RCCL)	En *builds* ROCm el backend también se llama `nccl` y usa RCCL
`mpi`	CPU/GPU	Solo si PyTorch se compila contra tu MPI

- ➔ `torchrun` define `RANK`, `WORLD_SIZE` y `MASTER_ADDR`; con `srun` se derivan de `SLURM_PROCID` y `SLURM_NTASKS`: lo mismo que `MPI_Comm_rank`/`size` del módulo 5.
- ➔ Con SLURM: un *rank* por GPU (`--ntasks-per-node`=GPU por nodo), igual que en MPI + CUDA.

Todo el semillero dentro de PyTorch

Módulo	Dónde aparece en PyTorch
1 SLURM	`sbatch`/`srun` lanzan los procesos de entrenamiento
2 Linux	entornos, `module`, `conda`/contenedores, variables de entorno
3 C/C++	ATen, el *dispatcher* y el motor de *autograd* están en C++
4 OpenMP	hilos intra-operador en CPU (`OMP_NUM_THREADS`)
5 MPI	`torch.distributed`: *rank*, *world size*, colectivas (*allreduce*, *all-to-all*)
6 CUDA	kernels, streams, eventos, cuBLAS/cuDNN, NCCL
7 AMD ROCm	*build* ROCm: misma API `torch.cuda`, HIP, rocBLAS, RCCL
8 IA desde HPC	GEMM, *backward*, DDP/*data parallelism*

Plantilla SLURM: PyTorch en una GPU o varias

```
#!/bin/bash
#SBATCH --job-name=minigpt
#SBATCH --partition=gpu_titan            # o amd (ROCm)
#SBATCH --nodes=1 --ntasks-per-node=1 --cpus-per-task=4
#SBATCH --gres=gpu:1                     # confirma el nombre del recurso (scontrol show node)
#SBATCH --time=00:30:00

source ~/miniconda3/etc/profile.d/conda.sh && conda activate torch   # o: module load conda / apptainer
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
python3 mini_gpt.py                      # varias GPU: torchrun --nproc_per_node=2 train.py
```

Plantilla no probada en guane. Confirma cómo se instala PyTorch (`conda`, `apptainer` o `pip`), qué versión de CUDA/ROCm acepta el *driver* y el nombre del recurso `--gres`.

Reto individual 5: mira por debajo de PyTorch

■ Reto 5 – PyTorch sin magia

1. Ejecuta `perfil_torch.py` en un nodo del cluster. ¿Qué porcentaje del tiempo es `aten::mm`/`addmm`? Repite con 1, 2, 4 y 8 hilos.
2. Imprime `torch.__config__.show()` en el cluster: ¿qué BLAS, qué CUDA y qué backends de comunicación trae?
3. Ejecuta `ddp_gloo.py` con 2 y 4 procesos. Verifica que los pesos coinciden con el entrenamiento secuencial.
4. En `gpu_titan`: mide `x @ x` (8192×8192) con y sin `torch.cuda.synchronize()`. ¿Cuál es el tiempo correcto? Calcula TFLOP/s.

6

Escala: FLOPs, memoria, prefill y decode

Cuánto cómputo cuesta entrenar

- **Forward**: $\approx 2P$ FLOP por token (cada parámetro: una multiplicación y una suma).
- **Backward**: $\approx 2\times$ el *forward*. En total:

$$\text{FLOPs}_{\text{entrenar}} \approx 6\,P\,D$$

con $D =$ tokens de entrenamiento (Kaplan et al., 2020).

- Ejemplo **ilustrativo**: $P = 7 \times 10^9$, $D = 10^{12}$ tokens $\Rightarrow 4.2 \times 10^{22}$ FLOP.
- Con 1000 GPU que sostienen 2×10^{14} FLOP/s cada una: $\approx 2.1 \times 10^5$ s ≈ 2.4 días.

★ La fórmula sirve para planear

Tiempo $\approx \dfrac{6PD}{\text{GPUs} \times \text{FLOP/s sostenido}}$.

El FLOP/s **sostenido** es una fracción del pico (*utilization*): memoria, comunicación y *kernels* pequeños lo bajan.

La memoria: pesos, estados y KV cache

Entrenar: ≈ 16 bytes por parámetro con Adam y *mixed precision* (pesos, gradientes y estados; ZeRO, 2020). Un modelo de 7B: ≈ 112 GB $\Rightarrow$ no cabe en una GPU de 80 GB: hay que **repartir** los estados.

Inferir: pesos (2 bytes por parámetro en FP16: ≈ 14 GB para 7B) **más el KV cache**.

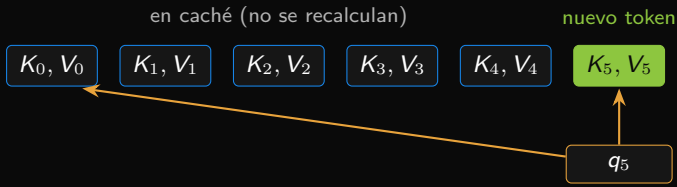
★ KV cache

Por token se guardan K y V de cada capa:

$$2 \cdot N \cdot d \cdot \text{bytes}$$

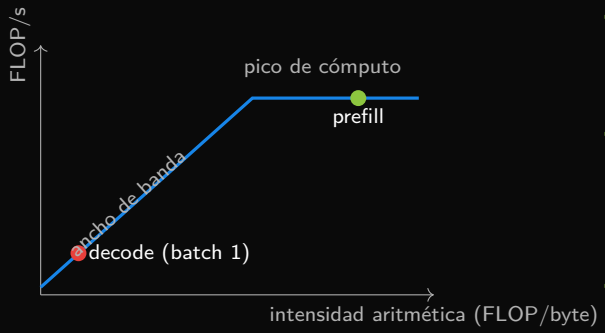
Con $N = 32$, $d = 4096$, FP16: **0.5 MiB por token**; 4096 tokens $\approx$ **2 GiB por secuencia** (atención *multi-head* completa). GQA lo reduce.

KV cache: no recalcular el pasado



- En cada paso de generación solo se calcula q, k, v del **token nuevo**; q atiende a todos los K, V guardados.
- Se cambia **cómputo por memoria**: sin *cache*, cada token costaría recalcular todo el contexto.
- El *cache* crece con la longitud y con el número de usuarios: **es el límite de memoria** de los servidores de LLM.

Prefill y decode: compute bound vs memory bound



- **Prefill** (procesar el *prompt*): muchos tokens a la vez ⇒ GEMM grandes, **compute bound**.
- **Decode** (un token por paso): hay que leer **todos los pesos** para $2P$ FLOP ⇒ ≈ 1 FLOP/byte con FP16 y *batch* 1: **memory bound**.
- Remedios: *batching* de muchos usuarios, ***quantization*** (INT8/INT4), *KV cache* eficiente.

Paralelismo de un LLM: lo que ya conoces, combinado

Estrategia	Qué reparte	Comunicación
Data (DDP)	datos; modelo replicado	*allreduce* de gradientes
FSDP / ZeRO	pesos, gradientes y estados	*allgather* y *reduce-scatter*
Tensor	cada GEMM entre varias GPU	*allreduce* por capa (NVLink)
Pipeline	grupos de capas por GPU	activaciones punto a punto
Expert (MoE)	expertos por GPU	*all-to-all* de tokens

Los sistemas grandes **combinan** varias ("3D parallelism"). Todo **depende** de la red: NVLink dentro del nodo, InfiniBand entre nodos. Las colectivas son análogas a las de MPI (módulo 5).

Reto individual 6: presupuesto de un modelo

■ Reto 6 – Hazlo con números

1. Para un modelo de 13B parámetros entrenado con 2×10^{12} tokens: calcula los FLOPs ($6PD$) y el tiempo en 256 GPU a 2×10^{14} FLOP/s sostenidos.
2. Calcula la memoria de entrenamiento (16 bytes por parámetro) y la de inferencia en FP16. ¿Cuántas GPU de 80 GB necesitas en cada caso?
3. Para $N = 40$, $d = 5120$, FP16: KV cache por token y por secuencia de 8192 tokens. ¿Cuántas secuencias caben en 40 GB?
4. Estima la intensidad aritmética del *decode* con *batch* 1 y con *batch* 32. ¿Qué pasa con el rendimiento?

7
Reto grupal y cierre del semillero

Reto grupal: un mini GPT, de C++ a la GPU

★ Reto grupal – Equipos de 3 o 4 personas

1. Entrenen el **mini GPT** con **su propio corpus** y un *split* de validación.
2. Midan *loss*, *tokens/s* y utilización en **CPU** (1 a 16 hilos) y **GPU** (`gpu_titan` y, si pueden, `amd`).
3. Ejecuten con **DDP** en 2 a 4 procesos y verifiquen que coincide con el secuencial.
4. Perfilen con `torch.profiler`: ¿qué fracción es GEMM? Prueben `torch.compile` y expliquen la diferencia.
5. Escalen D, H, L y T: ¿cuándo deja de caber o de escalar? Expliquen con FLOPs, memoria y comunicación.
6. *Extra*: su **kernel CUDA** (módulo 6) de atención, o *KV cache* en la generación.

Entregables y criterios

Entregables

- ➔ Repositorio con código, `requirements` y scripts `sbatch`.
- ➔ Gráficas: *loss*, *tokens/s*, *speedup*.
- ➔ Informe: dónde está el cuello de botella en cada sistema.
- ➔ Exposición del equipo: **"cómo funciona un LLM, sin magia"**.

Criterios

- ➔ Los resultados se pueden **reproducir** y verificar.
- ➔ Mediciones justas (sincronizar la GPU, repeticiones).
- ➔ Uso correcto de FLOPs, memoria y comunicación en el análisis.
- ➔ Claridad al explicar atención, bloque y escala.

Referencias

- A. Vaswani et al. *Attention Is All You Need*. NeurIPS, 2017.
- J. Kaplan et al. *Scaling Laws for Neural Language Models*. 2020.
- J. Hoffmann et al. *Training Compute-Optimal Large Language Models*. 2022.
- J. Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2021.
- T. Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. NeurIPS, 2022.
- J. Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023.
- N. Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017.
- W. Fedus, B. Zoph, N. Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. JMLR, 2022.
- A. Gu, T. Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2023.
- E. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021.
- S. Rajbhandari et al. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. SC20.
- A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. NeurIPS, 2019.
- Documentación de PyTorch: pytorch.org/docs (· `torch.distributed`, `torch.compile`, `torch.profiler`).

El recorrido completo del semillero



- Del **cluster** y la línea de comandos, al **código** eficiente; de los **hilos** a los **nodos** y a las **GPU**.
- Y la **IA actual** como aplicación: GEMM, *autograd*, atención y comunicación colectiva.
- El hilo conductor: **medir**, **verificar** y entender qué hace la máquina.

Cierre

★ La IA no es magia

Un LLM es una red neuronal que predice el siguiente token. Su capa clave, la atención, son **dos GEMM y una softmax**. PyTorch automatiza el *backward* y llama a C++, OpenMP, CUDA, ROCm y NCCL: **lo mismo que aprendieron en estos módulos**. Lo que lo hace posible es el HPC.

¡Gracias!

sc3.operaciones@uis.edu.co