CAGE – Grupo de Investigación

@sc3uis

SC3 – Super Cómputo y Cálculo Científico – UIS

IA desde HPC: la IA no es magia

Redes neuronales desde cero con C++, OpenMP, MPI y CUDA

Módulo 8 – Semillero CAGE

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

Objetivos del módulo

- ➔ Entender que **entrenar una red neuronal** es álgebra lineal, cálculo y un bucle: **no hay magia**.
- ➔ Programar **desde cero** una red completa: *forward*, *loss*, *backpropagation* y *gradient descent*.
- ➔ Identificar dónde se va el tiempo: casi todo son **GEMM** (multiplicaciones de matrices).
- ➔ Paralelizar el **mismo** entrenamiento con **OpenMP**, **MPI** (*data parallelism*) y **CUDA**.
- ➔ Verificar que todas las versiones producen **el mismo resultado** y medir su rendimiento.
- ➔ Conectar con la IA a gran escala: *data/model/pipeline parallelism*, memoria y comunicación.

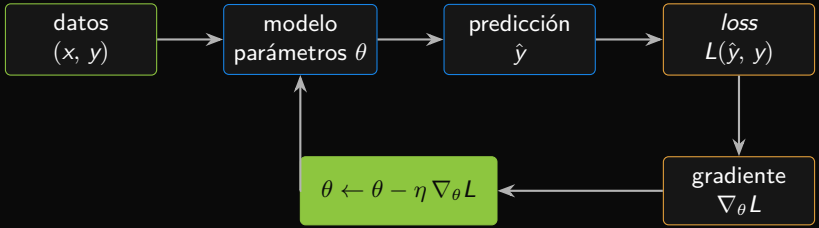
Agenda

1. Aprender es optimizar: *gradient descent*
2. De la neurona a la red: capas = matrices
3. *Backpropagation*: la regla de la cadena en código
4. ¿Dónde se va el tiempo? FLOPs y GEMM
5. OpenMP: paralelismo de hilos en el entrenamiento
6. MPI: *data parallelism* y *allreduce*
7. CUDA: el entrenamiento completo en la GPU
8. IA a gran escala y reto grupal

1

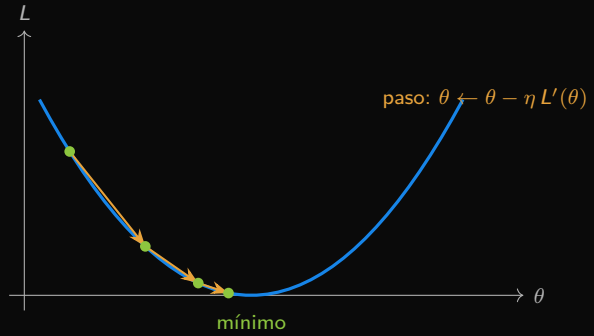
Aprender es optimizar: gradient descent

El ciclo de entrenamiento



- **Aprender** = encontrar los parámetros θ que minimizan una *loss* (el error medio sobre los datos).
- El gradiente dice hacia dónde crece el error; damos un paso **en contra**. η es el *learning rate*.
- Todo el *deep learning* es este bucle; cambia el tamaño del modelo y de los datos.

Gradient descent en una dimensión



- ➔ η **muy grande**: el paso se pasa del mínimo y el error **diverge**. η **muy pequeño**: converge, pero lento.
- ➔ Con millones de parámetros θ es un vector: el *gradient* es un vector del mismo tamaño.

Primer modelo: regresión lineal en C++

```cpp
double w = 0, b = 0, lr = 0.1;                    // parametros y learning rate
for (int it = 0; it <= 200; ++it) {
    double gw = 0, gb = 0, loss = 0;
    for (int i = 0; i < n; ++i) {
        double e = w * x[i] + b - y[i];           // error de la prediccion
        loss += e * e;  gw += 2 * e * x[i];  gb += 2 * e;   // grad. de e^2
    }
    w -= lr * gw / n;  b -= lr * gb / n;          // paso contra el gradiente
    if (it % 50 == 0) printf("%d %.5f %.3f %.3f\n", it, loss/n, w, b);
}
```

```
it   0  loss 3.22208  w 0.197  b 0.107        # verdad: w = 3, b = 0.5
it  50  loss 0.00431  w 2.906  b 0.501
it 200  loss 0.00085  w 3.002  b 0.500
```

Con datos $y = 3x + 0.5 +$ ruido, el programa **descubre** $w \approx 3$, $b \approx 0.5$ sin que se lo digamos.

Reto individual 1: el learning rate

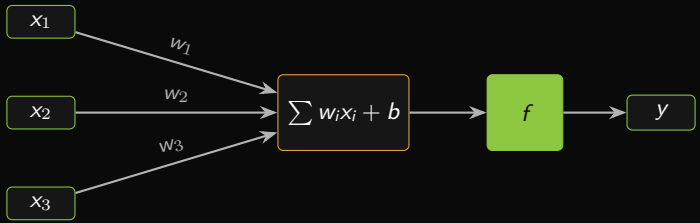
■ Reto 1 – Domando el learning rate

1. Compila y ejecuta `linreg.cpp`. ¿En cuántas iteraciones la *loss* baja de 0.01?
2. Prueba `lr` $= 0.001$, 0.1, 1.0 y 1.1. ¿Cuál converge más rápido? ¿Cuál diverge?
3. Añade una segunda característica: $y = 2x_1 - x_2 + 1$. Escribe los tres gradientes.
4. Cambia el *loss* a error absoluto $|e|$: ¿cómo cambia el gradiente?

2

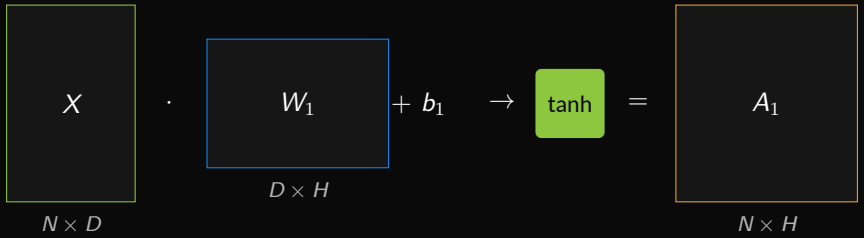
De la neurona a la red: capas = matrices

Una neurona



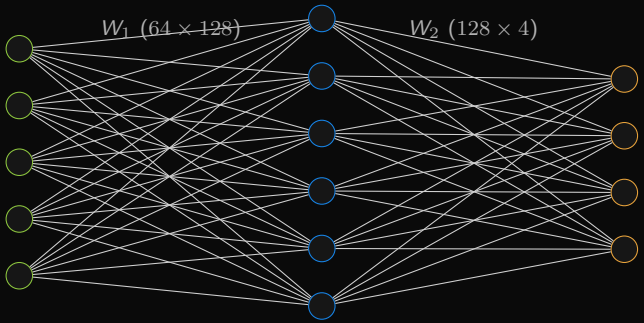
- Una neurona calcula una **suma ponderada** de sus entradas y le aplica una función no lineal f (*activation*).
- Sin f, apilar capas seguiría siendo una sola transformación lineal: la **no linealidad** es lo que da capacidad.
- Una neurona = un *dot product* más una función: operación simple, repetida millones de veces.

Una capa completa es una multiplicación de matrices



- N muestras a la vez (*batch*) $\times$ D características: una sola **GEMM** calcula la salida de H neuronas para las N muestras.
- **Esto es lo que hace rápida a una GPU**: millones de *dot products* independientes.
- Costo: $2\,N\,D\,H$ operaciones de punto flotante (FLOP).

La red que vamos a programar



8 836 parámetros $(64 \cdot 128 + 128 + 128 \cdot 4 + 4)$. Clasifica cada muestra en 1 de 4 clases. Los datos son sintéticos: la etiqueta la decide una red "maestra" aleatoria, así sabemos que el problema **es aprendible**.

Activación, softmax y entropía cruzada

Activaciones (derivada en la columna siguiente)

- ➔ $\tanh(z)$: salida en $(-1, 1)$.
- ➔ ReLU: $\max(0, z)$, una de las más usadas.
- ➔ Sigmoid: $1/(1 + e^{-z})$.

Salida: softmax

$$p_c = \frac{e^{z_c - \max z}}{\sum_k e^{z_k - \max z}}$$

Convierte puntajes en probabilidades; restar el máximo evita el *overflow*.

Loss: entropía cruzada

$$L = -\frac{1}{N} \sum_i \log p_{i,\, y_i}$$

Mide qué tan poca probabilidad dio el modelo a la clase correcta.

Derivadas útiles

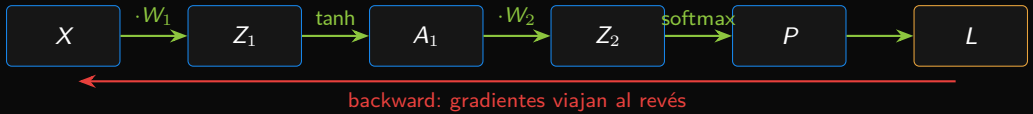
- ➔ $\tanh'(z) = 1 - \tanh^2(z)$
- ➔ Softmax + entropía cruzada: $\dfrac{\partial L}{\partial z} = \dfrac{p - \mathbb{1}_y}{N}$

Este último resultado tan simple es el **punto de partida** de *backpropagation*.

3

Backpropagation: la regla de la cadena en código

Forward y backward



- **Forward**: de los datos a la *loss*. Se guardan las activaciones intermedias.
- **Backward** (*backpropagation*): regla de la cadena desde L hasta los parámetros; **reutiliza** lo guardado.
- Costo de *backward* $\approx 2\times$ el de *forward* en general: por cada GEMM hacia adelante hay **dos** hacia atrás (gradiente de los pesos y de la entrada). En esta red la primera capa no calcula el gradiente de la entrada, así que aquí cuesta $\approx 1\times$.

Las ecuaciones: todo son GEMM

Paso	Forma	Operación
$dZ_2 = (P - \mathbb{1}_y)/N$	$N \times C$	elemento a elemento
$dW_2 = A_1^\top \, dZ_2$	$H \times C$	`GEMM` $(A^\top B)$
$db_2 = \sum_i dZ_2[i,:]$	C	suma por columna
$dA_1 = dZ_2 \, W_2^\top$	$N \times H$	`GEMM` $(AB^\top)$
$dZ_1 = dA_1 \odot (1 - A_1^2)$	$N \times H$	elemento a elemento
$dW_1 = X^\top \, dZ_1$	$D \times H$	`GEMM` $(A^\top B)$
$db_1 = \sum_i dZ_1[i,:]$	H	suma por columna

Con esto se **deriva a mano** la red completa. Los *frameworks* (PyTorch, TensorFlow) automatizan exactamente estas fórmulas: *autograd*. Ref.: Rumelhart, Hinton y Williams (1986).

Forward en C++

```
gemm(X.data(), W1.data(), A1.data(), N, H, D);          // Z1 = X W1
for (int i = 0; i < N; ++i) for (int j = 0; j < H; ++j)
    A1[i*H+j] = tanhf(A1[i*H+j] + b1[j]);               // A1 = tanh(Z1 + b1)
gemm(A1.data(), W2.data(), P.data(), N, C, H);          // Z2 = A1 W2

for (int i = 0; i < N; ++i) {                           // softmax + CE
    float mx = -1e30f, s = 0; int a = 0;
    for (int c = 0; c < C; ++c) {
        P[i*C+c] += b2[c];
        if (P[i*C+c] > mx) { mx = P[i*C+c]; a = c; }
    }
    for (int c = 0; c < C; ++c) { P[i*C+c] = expf(P[i*C+c] - mx); s += P[i*C+c];
        }
    for (int c = 0; c < C; ++c) P[i*C+c] /= s;
    loss -= logf(P[i*C+y[i]] + 1e-12f);  ok += (a == y[i]);
}
```

Matrices en **row-major** (`A[i*cols+j]`), como en el módulo 3. `gemm` calcula $C = AB$ con el orden de bucles i, p, j (acceso consecutivo en B y C).

La GEMM: tres bucles

```
// C[m x n] = A[m x k] * B[k x n]
void gemm(const float* A, const float* B, float* C, int m, int n, int k) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) C[i*n + j] = 0.0f;
        for (int p = 0; p < k; ++p) {
            float a = A[i*k + p];
            for (int j = 0; j < n; ++j) C[i*n + j] += a * B[p*n + j];  // j
                consecutivo
        }
    }
}
```

- $2mnk$ FLOP. El orden i, p, j recorre B y C por filas: es **cache-friendly** (módulo 3).
- Hay otras dos variantes: $A^\top B$ y $AB^\top$ (`gemm_tn`, `gemm_nt`) para el *backward*.
- Las bibliotecas (OpenBLAS, MKL, cuBLAS) hacen esto **mucho** mejor: *tiling*, SIMD, *Tensor Cores*.

Backward y actualización en C++

```
for (int i = 0; i < N; ++i) for (int c = 0; c < C; ++c)
    dZ2[i*C+c] = (P[i*C+c] - (c == y[i])) / N;          // dZ2 = (P - Y)/N
gemm_tn(A1.data(), dZ2.data(), dW2.data(), H, C, N);    // dW2 = A1^T dZ2
for (int c = 0; c < C; ++c) {                           // db2: suma por columna
    db2[c] = 0; for (int i = 0; i < N; ++i) db2[c] += dZ2[i*C+c]; }
gemm_nt(dZ2.data(), W2.data(), dA1.data(), N, H, C);    // dA1 = dZ2 W2^T
for (int i = 0; i < N*H; ++i) dA1[i] *= 1.0f - A1[i]*A1[i];   // derivada tanh
gemm_tn(X.data(), dA1.data(), dW1.data(), D, H, N);     // dW1 = X^T dA1
for (int j = 0; j < H; ++j) {                           // db1
    db1[j] = 0; for (int i = 0; i < N; ++i) db1[j] += dA1[i*H+j]; }
// actualizar: gradient descent
for (int i = 0; i < D*H; ++i) W1[i] -= lr * dW1[i];
for (int i = 0; i < H*C; ++i) W2[i] -= lr * dW2[i];
for (int j = 0; j < H; ++j) b1[j] -= lr * db1[j];
for (int c = 0; c < C; ++c) b2[c] -= lr * db2[c];
```

Siete líneas de cálculo y cuatro de actualización: **eso** es entrenar una red.

Entrenando: la red aprende

```
$ g++ -O2 -o mlp mlp.cpp && ./mlp
it   0  loss 1.4002  acc 0.248        # al azar: ~ 1/4
it  25  loss 0.5009  acc 0.919
it  50  loss 0.2783  acc 0.957
it  75  loss 0.2083  acc 0.966
it 100  loss 0.1730  acc 0.970
```

- ➔ Empieza en $\approx \ln 4 = 1.39$ (adivinar al azar entre 4 clases) y la *loss* baja de forma monótona.
- ➔ La precisión pasa de **25 %** a **97 %** sin que escribamos una sola regla de clasificación.
- ➔ Es **full-batch** *gradient descent*: cada iteración usa las $N = 4096$ muestras.

Los datos usan `rand()` con semilla fija: con otra `libc` las primeras cifras pueden variar, pero converge igual. El tiempo depende de la máquina.

Reto individual 2: gradient check

■ Reto 2 – Verifica tu backward

1. Implementa una función que calcule solo la *loss* (forward).
2. Para unos pocos pesos w de W_1 calcula la derivada numérica $\dfrac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon}$ con $\epsilon = 10^{-3}$.
3. Compárala con el valor de `dW1` de *backpropagation*: el error relativo debe ser menor que 10^{-2} en `float`.
4. Rompe a propósito la fórmula de la derivada de tanh y observa cómo falla el *gradient check*.

Reto individual 3: cambia la red

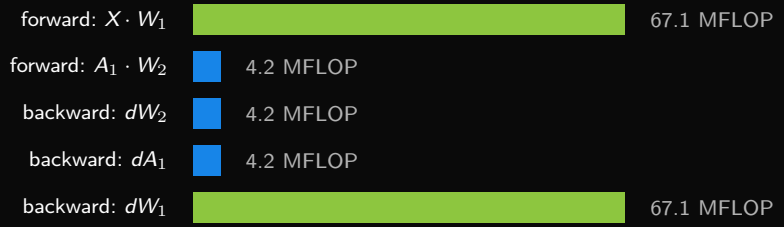
■ Reto 3 – Tu propia arquitectura

1. Cambia tanh por **ReLU** (derivada: 1 si $z > 0$, 0 si no). ¿Cambia la velocidad de aprendizaje?
2. Varía H (16, 64, 128, 512) y η (0.05, 0.5, 2): haz una tabla de *loss* y precisión tras 100 iteraciones.
3. Pasa de *full-batch* a **mini-batch** (por ejemplo 256 muestras por paso). ¿Qué ocurre con el costo por iteración?
4. Añade una **segunda capa oculta**: ¿cuántas GEMM nuevas aparecen en forward y backward?

4

¿Dónde se va el tiempo? FLOPs y GEMM

Contando FLOP por iteración



- Con $N = 4096$, $D = 64$, $H = 128$, $C = 4$: ≈**147 MFLOP** por iteración, repartidos en **5 GEMM**. Las operaciones elemento a elemento (tanh, softmax, actualización) suman menos del 1 % de los FLOP.
- Las GEMM grandes son las que tienen la dimensión H y D: **ahí** hay que paralelizar.
- Para 100 iteraciones: ≈14.7 GFLOP. FLOP/s = FLOP / tiempo: mídelo tú.

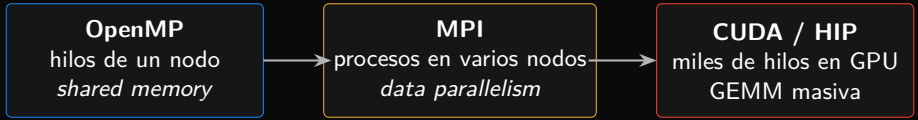
Compute bound vs memory bound

- **Intensidad aritmética** = FLOP / byte movido.
- GEMM de $n \times n$: $2n^3$ FLOP sobre $\approx 3n^2$ valores $\Rightarrow$ crece con n: **compute bound**.
- Activaciones, *softmax*, actualización de pesos: pocas operaciones por byte $\Rightarrow$ **memory bound**.
- Por eso *batch* grande y capas anchas **usan mejor** la GPU: más FLOP por byte.

★ El modelo roofline

Es el modelo *roofline* (Williams et al., 2009): el rendimiento está limitado por el menor entre **pico de cómputo** y **ancho de banda** $\times$ **intensidad aritmética**.

El mismo entrenamiento, tres niveles de paralelismo



- Es **el mismo algoritmo** y el mismo resultado numérico; cambia dónde se ejecutan los bucles.
- Los sistemas reales los **combinan**: varios nodos (MPI) $\times$ varias GPU por nodo $\times$ miles de hilos por GPU.
- A continuación los programamos uno por uno sobre `mlp.cpp`.

5

OpenMP: paralelismo de hilos en el entrenamiento

Paralelizar la GEMM por filas

```
void gemm(const float* A, const float* B, float* C,
          int m, int n, int k) {
    #pragma omp parallel for
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) C[i*n+j] = 0.0f;
        for (int p = 0; p < k; ++p) {
            float a = A[i*k + p];
            for (int j = 0; j < n; ++j)
                C[i*n+j] += a * B[p*n+j];
        }
    }
}
```



- ➔ Cada fila de C es **independiente**: no hay *race condition*. Una línea basta.
- ➔ Se compila con `g++ -O2 -fopenmp mlp.cpp` y se controla con `OMP_NUM_THREADS`.

Resultado: mismo resultado, menos tiempo

```
$ for t in 1 2; do OMP_NUM_THREADS=$t ./mlp_omp | tail -2; done
it 100  loss 0.1730  acc 0.970
tiempo: 5.339 s            # 1 hilo
it 100  loss 0.1730  acc 0.970
tiempo: 3.733 s            # 2 hilos
```

- La *loss* es **idéntica** con 1 y 2 hilos: verificación de corrección.
- Ejemplo medido en una máquina de **2 núcleos**: *speedup* ≈ 1.4. **Mide en guane** con 1, 2, 4, 8, 16 hilos.
- Ley de Amdahl: lo que no paraleliza (actualización de pesos, sumas de sesgos, *softmax*) limita el *speedup*.

★ Mide tú

El resultado numérico en `float` puede variar en el último dígito si cambia el orden de las sumas (por ejemplo en una *reduction*): compara con tolerancia, no con `==`.

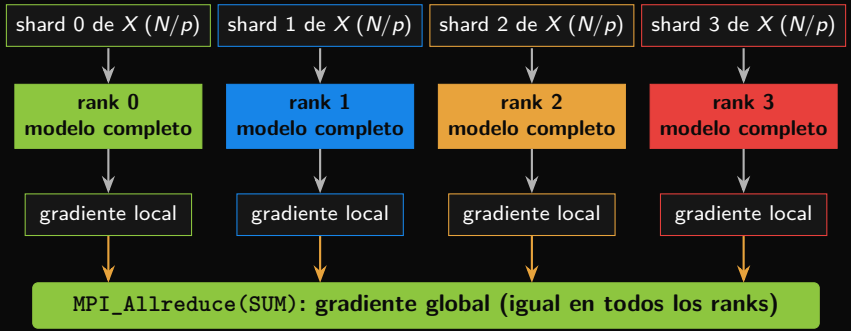
Reto individual 4: paraleliza el resto

■ Reto 4 – OpenMP completo

1. Mide con `omp_get_wtime` cuánto tarda cada etapa (GEMM, *softmax*, sumas de sesgos, actualización).
2. Paraleliza las etapas que faltan: bucles elemento a elemento y la suma de `db1`/`db2`. ¿Cuál necesita `reduction` o paralelizar por columnas?
3. Haz la gráfica *speedup* y eficiencia con 1 a 16 hilos en un nodo `legacy`. Compara con la ley de Amdahl.
4. Mejora `gemm`: bloques (*tiling*) y `#pragma omp simd`. ¿Cuántos GFLOP/s logras? Compara con OpenBLAS.

6
MPI: data parallelism y allreduce

Data parallelism



- Cada *rank* tiene una **copia completa** del modelo y procesa **una parte de los datos**.
- La *loss* es una **suma sobre muestras**, así que el gradiente global = suma de los gradientes locales.
- Tras el *allreduce* todos aplican la **misma** actualización: los modelos siguen idénticos.

El código MPI: dos cambios

```
const int Nl = N / size, off = rank * Nl;      // cada rank: un bloque de filas
const float* Xl = X.data() + off * D;
const int*   yl = y.data() + off;
...
gemm(Xl, W1.data(), A1.data(), Nl, H, D);      // forward: mis Nl muestras
...
dZ2[i*C+c] = (P[i*C+c] - (c == yl[i])) / N;    // OJO: se divide entre N global
...
// despues del backward local: sumar los gradientes de todos los ranks
MPI_Allreduce(MPI_IN_PLACE, dW1.data(), D*H, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, dW2.data(), H*C, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, db1.data(), H,   MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, db2.data(), C,   MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
// ... y todos aplican la misma actualizacion de pesos
```

La *loss* y los aciertos también se suman con `MPI_Allreduce` para imprimirlos. Para simplificar, cada *rank* genera todo el conjunto de datos y se queda con su parte; en producción cada uno **lee solo su shard**.

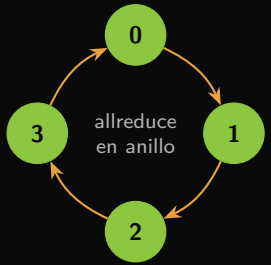
Resultado y verificación

```
$ mpicxx -O2 -o mlp_mpi mlp_mpi.cpp
$ mpirun -np 1 ./mlp_mpi | tail -2     # it 100  loss 0.1730  acc 0.970
$ mpirun -np 2 ./mlp_mpi | tail -2     # it 100  loss 0.1730  acc 0.970
$ mpirun -np 4 ./mlp_mpi | tail -2     # it 100  loss 0.1730  acc 0.970
```

- Con 1, 2 y 4 *ranks* la *loss* es **idéntica** a la versión secuencial: *data parallelism* **no cambia el algoritmo**, solo cómo se calcula.
- Ejemplo en 2 núcleos: 1 rank 6.5 s, 2 ranks 3.3 s (4 ranks *oversubscribed*: 3.5 s). **Mide en varios nodos del cluster.**
- Se usa `mpirun --oversubscribe` solo para pruebas; en SLURM, `srun` o `mpirun` con `--ntasks`.

El costo de comunicación

- Por iteración se comunican **solo los gradientes**: $8\,836 \times 4 \approx 35$ KB. Para un modelo de P parámetros: $4P$ bytes (en `float`).
- El cómputo escala con N/p; la comunicación **no depende de** N.
- Regla: el *data parallelism* rinde cuando $\frac{\text{cómputo local}}{\text{comunicación}}$ es grande: ***batch* por *rank*** grande, modelo pequeño o red rápida.
- Con modelos enormes, el *allreduce* de gradientes domina: se hace *overlap* con el *backward* y se usan NCCL (NVIDIA) o RCCL (AMD).



Cada *rank* envía a su vecino partes del vector: el ancho de banda usado por *rank* es $\approx 2\frac{p-1}{p}\cdot$ tamaño, casi independiente de p (Patarasuk y Yuan, 2009).

Reto individual 5: MPI

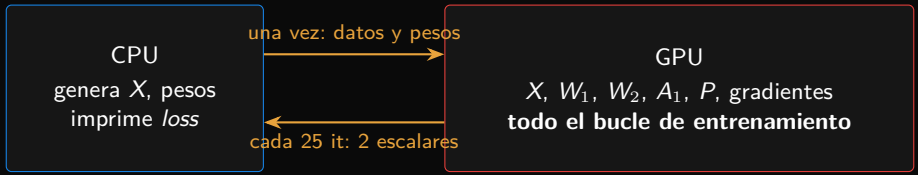
■ Reto 5 – Escalamiento con MPI

1. Compila `mlp_mpi.cpp` y ejecuta con 1, 2, 4 y 8 *ranks* en varios nodos; verifica que la *loss* coincide.
2. Mide por separado el tiempo de cómputo y el del *allreduce* con `MPI_Wtime`. ¿Cuánto pesa la comunicación?
3. Aumenta H a 1024 y 4096. ¿Cómo cambia la relación cómputo/comunicación?
4. Cambia a **mini-batch**: un *allreduce* por *mini-batch*. ¿Por qué hacer pasos más pequeños aumenta el costo relativo de comunicar?

7

CUDA: el entrenamiento completo en la GPU

Qué se mueve a la GPU



- ➔ Regla del módulo 6: **evita las transferencias**. Aquí los datos **viven en la GPU** durante todo el entrenamiento.
- ➔ Por iteración solo se lanzan *kernels*; a la CPU vuelven la *loss* y los aciertos de vez en cuando.
- ➔ Se necesitan: `gemm` (con variantes transpuestas), `bias_tanh`, `softmax_ce`, `dtanh`, `colsum`, `sgd`.

La GEMM en CUDA: un hilo por elemento de C

```cpp
// C[m x n] = op(A) * op(B); ta/tb: A/B transpuestas
__global__ void gemm(const float* A, const float* B, float* C,
                     int m, int n, int k, bool ta, bool tb) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= m || j >= n) return;
    float s = 0.0f;
    for (int p = 0; p < k; ++p)
        s += (ta ? A[p*m+i] : A[i*k+p]) * (tb ? B[j*k+p] : B[p*n+j]);
    C[i*n + j] = s;
}
// lanzamiento: bloques 16x16 que cubren C
dim3 bl(16, 16), gr((n + 15) / 16, (m + 15) / 16);
gemm<<<gr, bl>>>(A, B, C, m, n, k, false, false);
```

Es el **matmul ingenuo** del módulo 6: correcto, pero sin *tiling*. Con una sola variante cubrimos las tres GEMM del entrenamiento.

Kernels elemento a elemento y *reductions*

```cuda
__global__ void bias_tanh(float* A, const float* b, int rows, int cols) {
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    if (t < rows * cols) A[t] = tanhf(A[t] + b[t % cols]);   // 1 hilo/elemento
}
__global__ void softmax_ce(float* Z, const int* y, const float* b2, float* dZ,
                           float* loss, int* ok, int N, int C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;       // 1 hilo por muestra
    if (i >= N) return;
    /* softmax de la fila i; dZ[i,:] = (P - Y) / N */
    atomicAdd(loss, -logf(Z[i*C + y[i]] + 1e-12f));             // loss global
    atomicAdd(ok, (int)(a == y[i]));                            // aciertos
}
```

`atomicAdd` sobre un solo escalar **serializa**: mejor una *reduction* por bloque en `__shared__` (módulo 6). `colsum` usa un hilo por columna: pocos hilos, mejorable.

El bucle de entrenamiento en la GPU

```
for (int it = 0; it <= iters; ++it) {
    // forward
    mm(X, W1, A1, N, H, D, false, false);   bias_tanh<<<nb(N*H), 256>>>(A1, b1, N, H);
    mm(A1, W2, P, N, C, H, false, false);
    cudaMemset(loss, 0, sizeof(float));     cudaMemset(ok, 0, sizeof(int));
    softmax_ce<<<nb(N), 256>>>(P, y, b2, dZ2, loss, ok, N, C);
    // backward
    mm(A1, dZ2, dW2, H, C, N, true, false);                 // dW2 = A1^T dZ2
    colsum<<<nb(C), 256>>>(dZ2, db2, N, C);
    mm(dZ2, W2, dA1, N, H, C, false, true);                 // dA1 = dZ2 W2^T
    dtanh<<<nb(N*H), 256>>>(dA1, A1, N*H);
    mm(X, dA1, dW1, D, H, N, true, false);                  // dW1 = X^T dA1
    colsum<<<nb(H), 256>>>(dA1, db1, N, H);
    // actualizar
    sgd<<<nb(D*H), 256>>>(W1, dW1, lr, D*H);   sgd<<<nb(H*C), 256>>>(W2, dW2, lr, H*C);
    sgd<<<nb(H), 256>>>(b1, db1, lr, H);       sgd<<<nb(C), 256>>>(b2, db2, lr, C);
}
```

Los *kernels* van en el mismo *stream*: se ejecutan en orden; no hace falta sincronizar entre ellos.

Compilar y ejecutar en gpu_titan

```
module load cuda/11.8                    # y un GCC <= 11 como host (modulo 6)
nvcc -O2 -arch=sm_61 -ccbin g++ mlp_cuda.cu -o mlp_cuda   # sm_61: ejemplo
srun -p gpu_titan --gres=gpu:1 ./mlp_cuda
# esperado: la loss de la version CPU (salvo redondeo)
```

- ➔ Verifica la *loss*: debe parecerse a la de CPU; pequeñas diferencias vienen del orden de las sumas (`atomicAdd`) y de tanh/`exp` en GPU.
- ➔ Usa `cudaEvent` (como en el módulo 6) para medir el tiempo de la parte GPU y compáralo con OpenMP y MPI.
- ➔ **HIP**: con `hipify-perl` el mismo programa corre en la partición `amd` (módulo 7).

★ Código sin ejecutar aquí

`mlp_cuda.cu` compila sin errores, pero se escribió sin GPU a mano: ejecútalo en `gpu_titan` y confirma *loss* y tiempos antes de presentar resultados.

De aquí a una biblioteca real

- **cuBLAS** (`cublasSgemm`): mismo cálculo, optimizado por años.
- **Tensor Cores**: unidades que multiplican bloques de matrices pequeñas en **FP16/BF16/TF32**; el *mixed precision* acumula en FP32.
- **cuDNN**: capas completas (convolución, normalización).
- **Fusión de kernels**: unir `bias+tanh` evita ir y volver a memoria global.

★ Mensaje

PyTorch o TensorFlow **no hacen nada distinto** a lo que programaste: llaman a bibliotecas como cuBLAS/rocBLAS y a kernels propios, y derivan el *backward* automáticamente.

Reto individual 6: CUDA

■ Reto 6 – La red en la GPU

1. Compila y ejecuta `mlp_cuda.cu` en `gpu_titan`; compara la *loss* con la versión CPU.
2. Mide el tiempo con `cudaEvent`. Calcula GFLOP/s con los 147 MFLOP por iteración y compáralos con OpenMP.
3. Reemplaza el `gemm` por una versión con *tiles* en `__shared__` (módulo 6). ¿Cuánto mejora?
4. *Extra*: usa `cublasSgemm` y compara. ¿Qué fracción del rendimiento de la biblioteca alcanzas?

8
IA a gran escala

Tres formas de paralelizar un modelo

Data parallelism



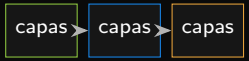
copia completa por GPU
datos repartidos

Model (tensor) parallelism



cada capa W se parte
en columnas entre GPU

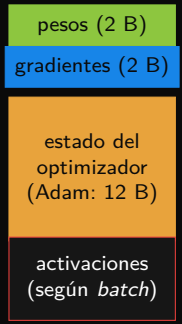
Pipeline parallelism



grupos de capas por GPU
micro-batches en *pipeline*

- ➔ Lo que programaste con MPI es **data parallelism**. Cuando el modelo **no cabe** en una GPU hace falta partirlo (*model/pipeline*).
- ➔ Los sistemas grandes combinan los tres y dependen de la **red** entre GPU (NVLink, InfiniBand) y de NCCL/RCCL.

La memoria también es un problema



- ➔ Con *mixed precision* y Adam se necesitan ≈**16 bytes por parámetro** sin contar activaciones (Rajbhandari et al., ZeRO, 2020).
- ➔ Un modelo de 7 000 millones de parámetros: ≈ 112 GB solo en estados: **no cabe** en una GPU de 80 GB.
- ➔ Soluciones: repartir estados (ZeRO/FSDP), recalcular activaciones en vez de guardarlas, cuantizar.

Lo que programaste vs. lo que ves en las noticias

En este módulo	En la IA a gran escala
MLP de 8 836 parámetros	Modelos de miles de millones de parámetros
full-batch gradient descent	*mini-batch* y optimizadores (Adam)
tanh + softmax	GELU/ReLU, atención, normalización
GEMM propia	cuBLAS/rocBLAS, Tensor Cores, *fused kernels*
allreduce con MPI	NCCL/RCCL, *overlap* con el *backward*
backward escrito a mano	*autograd*

Idea clave: los bloques de construcción son los mismos: GEMM, funciones elemento a elemento, *reductions* y comunicación colectiva. Lo que cambia es la **escala**, y por eso es un problema de HPC.

9
Reto grupal

Reto grupal: la red en cuatro versiones

★ Reto grupal – Equipos de 3 o 4 personas

Entrenen **la misma red** con las cuatro versiones y expliquen el resultado:

1. **C++** secuencial como línea base y OpenMP completo (módulo 4).
2. **MPI** (módulo 5) en 1 a 4 nodos.
3. **CUDA** en `gpu_titan` y, si pueden, **HIP** en `amd` con `hipify`.
4. Misma semilla y misma *loss* final (con tolerancia) en todas.
5. Escalen el problema: H y N grandes. ¿Cuándo gana cada versión y por qué?
6. *Extra*: Adam, *mini-batch*, segunda capa oculta, o **MPI + CUDA** (una GPU por *rank*, *allreduce* de gradientes).

Entregables y criterios

Entregables

- ➔ Repositorio con las versiones y un `Makefile`.
- ➔ Scripts `sbatch` por partición.
- ➔ Gráficas: *loss* vs. iteración, *speedup*, GFLOP/s por versión.
- ➔ Informe corto: dónde está el cuello de botella.
- ➔ Exposición del equipo: **“la IA no es magia”**.

Criterios

- ➔ Todas las versiones entrenan y coinciden.
- ➔ *Gradient check* documentado.
- ➔ Mediciones justas (repeticiones, nodos exclusivos).
- ➔ Análisis con Amdahl, *roofline* y costo de comunicación.
- ➔ Claridad y reproducibilidad.

Referencias

- D. Rumelhart, G. Hinton, R. Williams. *Learning representations by back-propagating errors*. Nature, 1986.
- I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016 (deeplearningbook.org).
- P. Patarasuk, X. Yuan. *Bandwidth optimal all-reduce algorithms for clusters of workstations*. JPDC, 2009.
- S. Rajbhandari et al. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. SC20, 2020.
- S. Williams, A. Waterman, D. Patterson. *Roofline: an insightful visual performance model for multicore architectures*. CACM 52(4), 2009.
- NVIDIA, *cuBLAS* y *CUDA C++ Programming Guide*; AMD, *rocBLAS* y *HIP Programming Guide*.
- Módulos 3 a 7 del semillero (C/C++, OpenMP, MPI, CUDA y HIP).

Cierre

★ La IA no es magia

Entrenar una red neuronal es: **multiplicar matrices**, **derivar con la regla de la cadena**, **dar pasos contra el gradiente** y **repetir**. Lo que la hace posible a gran escala es el trabajo de HPC: paralelismo, jerarquía de memoria, comunicación y medición rigurosa.

¡Gracias!

sc3.operaciones@uis.edu.co