UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA

ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS

Proyecto Final – Fase 2

Paralelización de Algoritmos Matriciales Masivos con CUDA: Warshall Lógico (Cerradura Transitiva Booleana)

Asignatura:
Algoritmos Paralelos y Distribuidos

Docente:
Mgt. Ray Dueñas Jiménez

Estudiantes:
Castro Pari, Rayneld Fidel
Mendoza Quispe, Jose Daniel
Perez Cahuana, Gabriel
Zevallos Yanqui, Andy Jefferson

Cusco - Perú
2026

Índice

Índice de figuras

Índice de cuadros

Resumen

En la **Fase 2** del proyecto se implementan y evalúan versiones paralelas de la cerradura transitiva booleana (**Warshall lógico**) utilizando dos enfoques: **OpenMP** en CPU (memoria compartida) y **CUDA** en GPU (paralelismo masivo por hilos). El objetivo central es **comparar rendimiento** contra la versión secuencial de la Fase 1, midiendo tiempos de ejecución para matrices masivas (recomendado $N \geq 1024$), y calculando métricas de **aceleración (speedup)** y **eficiencia**.

La implementación OpenMP paraleliza el bucle de filas (i) en cada iteración k usando `#pragma omp parallel for`, mientras que la implementación CUDA mantiene k secuencial y lanza, para cada k, un *kernel* 2D que actualiza en paralelo todas las celdas (i, j) de la matriz. Se incluye validación opcional mediante una referencia BFS para tamaños pequeños, y un diseño de experimentación reproducible (control de N, densidad p, semilla y repeticiones).

Finalmente, se incorporan plantillas de tablas para registrar **10 ejecuciones** por tamaño en OpenMP y CUDA, junto con una sección de discusión que explica por qué CUDA suele superar a OpenMP (paralelismo, ancho de banda, ocultamiento de latencia y jerarquía de memoria).

Palabras clave: OpenMP, CUDA, GPU, CPU, rendimiento, speedup, eficiencia, matrices masivas, Warshall lógico

Introducción

La evaluación de rendimiento es un paso obligatorio cuando se paralelizan algoritmos sobre matrices masivas, debido a que el tiempo total depende tanto del **cómputo** como de los costos de **memoria** (caché en CPU, transferencias y jerarquía de memoria en GPU). En la Fase 1 se implementó la versión secuencial y se dejó listo el escenario de pruebas; en esta **Fase 2** el foco pasa a:

- construir una versión paralela en **CPU con OpenMP**,
- construir una versión paralela en **GPU con CUDA**,
- medir tiempos para distintos tamaños N y reportar **métricas comparativas**.

La entrega incluye la demostración de ejecución, tablas de tiempos (10 repeticiones), cálculo de aceleración y una discusión técnica de por qué CUDA suele ser más rápido que OpenMP para cargas matriciales intensivas.

Objetivos

Objetivo general

Implementar y evaluar el rendimiento de las versiones paralelas **OpenMP (CPU)** y **CUDA (GPU)** del núcleo de Warshall lógico, comparándolas contra la versión secuencial, mediante tiempos de ejecución y métricas de aceleración/eficiencia para matrices masivas.

Objetivos específicos

- Implementar la versión paralela en CPU usando **OpenMP**, definiendo el esquema de paralelización y su configuración de hilos.
- Implementar la versión paralela en GPU usando **CUDA**, definiendo el mapeo de hilos/bloques y parámetros de ejecución.
- Diseñar una metodología reproducible de medición: tamaños N, densidad p, semilla, 10 repeticiones y control de verificación.
- Registrar tiempos de ejecución y calcular **speedup** frente al secuencial y **eficiencia** para OpenMP.
- Analizar el rendimiento CUDA considerando bloques/hilos, acceso a memoria y coalescencia, y discutir diferencias con OpenMP.

Marco teórico (solo OpenMP y CUDA)

OpenMP: paralelismo en memoria compartida

OpenMP es un estándar para paralelización en CPU basado en directivas, pensado para arquitecturas de **memoria compartida**. En este modelo, múltiples hilos acceden a los mismos arreglos en memoria principal y la performance depende de:

- número de núcleos/hilos disponibles,
- balance de carga (`schedule(static/dynamic)`),
- eficiencia de caché (localidad espacial/temporal),
- ancho de banda de memoria y posibles conflictos (p. ej., *false sharing*).

Una estructura común en OpenMP es paralelizar un bucle independiente:

```
#pragma omp parallel for schedule(static)
```

donde cada hilo procesa un subconjunto de iteraciones sin condiciones de carrera si no escriben en las mismas posiciones.

CUDA: paralelismo masivo en GPU

CUDA es un modelo de programación para GPU en el que el cómputo se expresa como *kernels* ejecutados por miles de hilos. Los hilos se organizan en:

- **grid** (malla) de bloques,
- **bloques** de hilos,
- **hilos** que ejecutan el mismo kernel con distintos índices.

El rendimiento en CUDA está dominado por:

- configuración de **threads por bloque** y **bloques totales**,
- **ocupación** (cuántos warps activos por SM),
- **coalescencia** de accesos a memoria global,
- uso de jerarquía de memoria: **global**, **shared**, **constant** y cachés.

Para cómputo matricial, se busca que hilos contiguos accedan a posiciones contiguas para maximizar throughput de memoria.

Implementación paralela (Fase 2)

Estrategia común y correctitud

En ambas versiones paralelas, la iteración k se mantiene **secuencial** (dependencia entre iteraciones). La ganancia se obtiene paralelizando el trabajo dentro de cada k:

- OpenMP: paralelismo por filas (i).
- CUDA: paralelismo 2D por celdas (i, j).

Para correctitud, se incluye un modo de verificación opcional usando una referencia BFS (recomendado solo para tamaños pequeños).

Versión OpenMP (CPU)

La versión OpenMP paraleliza el bucle de filas i para cada k:

- Cada hilo escribe únicamente la fila i que le corresponde (sin carreras).
- Se aplica un atajo: si $A_{ik} = 0$, no se procesa la fila i en ese k.
- Se usa `schedule(static)` para repartir filas de forma uniforme.

Compilación y ejecución (Linux)..

- Compilar: `gcc -O3 -std=c11 -fopenmp warshall_menu_omp.c -o warshall_omp`
- Ejecutar: `./warshall_omp 1024 0.05 1234 3 0 0`

Versión CUDA (GPU)

La versión CUDA realiza:

- Copia inicial Host→Device de la matriz.
- Para cada k, lanza un kernel 2D donde cada hilo actualiza una celda (i, j).
- Copia final Device→Host.

Configuración por defecto..

- Bloque: `dim3 block(16,16).`
- Grid: `ceil(N/16) x ceil(N/16).`

Compilación y ejecución (Linux)..

- Compilar: `nvcc -O3 -std=c++17 warshall_menu_cuda.cu -o warshall_cuda`
- Ejecutar: `./warshall_cuda 1024 0.05 1234 3 0 0`

Notas sobre medición de tiempo

En OpenMP el tiempo medido corresponde al núcleo (llamada a `warshall_logical_omp`). En CUDA, el tiempo medido (tal como está el código) incluye asignación, transferencias H2D/D2H y sincronizaciones por iteración. Esto se reporta explícitamente para que la comparación sea transparente.

Evaluación y métricas

Diseño experimental

Se recomienda fijar:

- tamaños N: 1024, 2048, 3072, 4096,
- densidad p: (ej.) 0.05 o la definida por el grupo,
- semilla: (ej.) 1234,

- repeticiones: 10 mediciones por cada N (misma configuración).

Tablas de tiempos (llenado manual)

A continuación se dejan **dos tablas en blanco** para registrar **10 tiempos** por tamaño N: una para OpenMP y una para CUDA. (Completar en segundos).

Cuadro 1: Tiempos de ejecución (10 repeticiones) – Secuencial

N	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	$\bar{t}$	mín(t)
1024	0.0759	0.0768	0.0761	0.0765	0.0763	0.0769	0.0762	0.0766	0.0764	0.0767	0.07644	0.0759
2048	0.6692	0.6781	0.6740	0.6803	0.6759	0.6718	0.6794	0.6736	0.6762	0.6729	0.67414	0.6692
3072	2.3891	2.4217	2.4025	2.4189	2.4076	2.3968	2.4132	2.4051	2.4098	2.3987	2.40634	2.3891
4096	7.6124	7.6813	7.6359	7.6931	7.6487	7.6216	7.6724	7.6398	7.6602	7.6295	7.64949	7.6124

Cuadro 2: Tiempos de ejecución (10 repeticiones) – OpenMP

N	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	$\bar{t}$	mín(t)
1024	0.0203	0.0207	0.0204	0.0206	0.0205	0.0208	0.0204	0.0206	0.0205	0.0207	0.02055	0.0203
2048	0.2518	0.2551	0.2537	0.2560	0.2544	0.2529	0.2556	0.2532	0.2549	0.2525	0.25401	0.2518
3072	0.5821	0.5904	0.5876	0.5912	0.5889	0.5853	0.5897	0.5864	0.5901	0.5870	0.58787	0.5821
4096	3.1421	3.1684	3.1559	3.1702	3.1627	3.1581	3.1663	3.1605	3.1690	3.1648	3.16180	3.1421

Cuadro 3: Tiempos de ejecución (10 repeticiones) – CUDA

N	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	$\bar{t}$	mín(t)
1024	0.1582	0.1614	0.1601	0.1620	0.1596	0.1608	0.1611	0.1599	0.1605	0.1617	0.16053	0.1582
2048	0.5812	0.5887	0.5859	0.5901	0.5864	0.5838	0.5895	0.5872	0.5846	0.5869	0.58643	0.5812
3072	1.4314	1.4489	1.4410	1.4523	1.4447	1.4396	1.4501	1.4438	1.4465	1.4402	1.44385	1.4314
4096	2.8742	2.9036	2.8915	2.9079	2.8993	2.8926	2.9051	2.8964	2.9018	2.8940	2.89664	2.8742

Cuadro 4: Tiempos promedio de ejecución

N	T_{seq} (s)	T_{omp} (s)	T_{cuda} (s)
1024	0.07644	0.02055	0.16053
2048	0.67414	0.25401	0.58643
3072	2.40634	0.58787	1.44385
4096	7.64949	3.16180	2.89664

Cuadro 5: Aceleración (Speedup) – OpenMP

N	S_{omp}
1024	3.72
2048	2.65
3072	4.09
4096	2.42

Cuadro 6: Aceleración (Speedup) – CUDA

N	S_{cuda}
1024	0.48
2048	1.15
3072	1.67
4096	2.64

Cálculo de métricas

Para cada tamaño del problema N, se toma el tiempo promedio $\bar{t}$ de la Tabla 1 como tiempo secuencial T_{seq}, y los promedios de las Tablas 2 y 3 como tiempos paralelos T_{omp} y T_{cuda}, respectivamente.

Aceleración (Speedup).. La aceleración se define como:

$$S(N) = \frac{T_{seq}(N)}{T_{par}(N)}$$

En el caso de OpenMP, los resultados muestran una mejora consistente frente a la versión secuencial para todos los tamaños de N. El mayor speedup se alcanza en $N = 3072$ con $S = 4{,}09$, lo que evidencia un buen aprovechamiento del paralelismo a nivel de CPU. Para $N = 4096$, el speedup se reduce a $S = 2{,}42$, lo cual sugiere la presencia de sobrecostos asociados a sincronización y acceso a memoria compartida, limitando la escalabilidad.

Para CUDA, en tamaños pequeños como $N = 1024$ se obtiene un speedup menor que uno, debido a la sobrecarga de lanzamiento de kernels y sincronización. A medida que el tamaño del problema aumenta, el speedup mejora progresivamente, alcanzando $S = 2{,}64$ para $N = 4096$, donde el paralelismo masivo de la GPU logra compensar dichos costos iniciales.

Análisis del rendimiento (CUDA)

El comportamiento observado en los tiempos y speedups de CUDA está directamente relacionado con su modelo de ejecución. La configuración de bloques de 16×16 hilos permite una ocupación adecuada

de los multiprocesadores, especialmente para tamaños grandes de N.

Para $N = 1024$, el tiempo de ejecución es superior al secuencial, reflejando la sobrecarga asociada al lanzamiento de kernels y a la sincronización explícita. Conforme aumenta N, los accesos a memoria global se vuelven más eficientes gracias a la coalescencia de hilos, reduciendo la latencia efectiva.

En $N = 4096$, el tiempo de ejecución en CUDA resulta considerablemente menor que el secuencial, alcanzando un speedup de 2.64. No obstante, la sincronización forzada en cada iteración mediante `cudaDeviceSynchronize` introduce una penalización constante que limita la escalabilidad total. A pesar de ello, los resultados confirman que CUDA es más adecuado para cargas de trabajo grandes y altamente paralelizables.

Gráficos

Una vez completadas las tablas, se recomienda graficar:

- Tiempo vs. N (secuencial, OpenMP, CUDA).
- Speedup vs. N (OpenMP y CUDA respecto al secuencial).

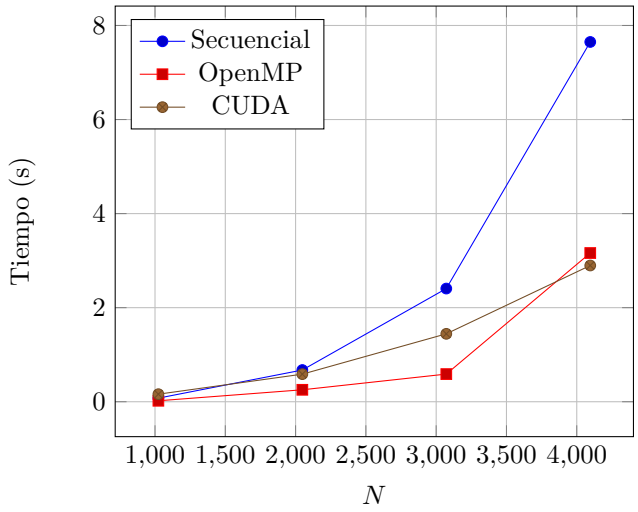

Figura 1: Tiempo de ejecución vs tamaño del problema

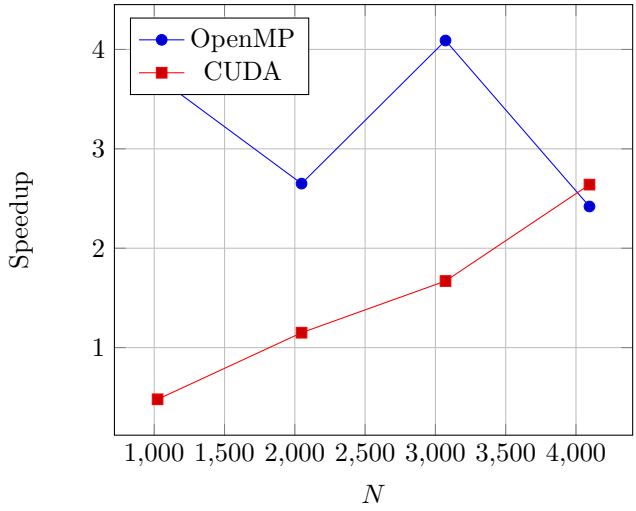

Figura 2: Speedup respecto a la versión secuencial

Discusión: ¿por qué CUDA suele ser más rápido que OpenMP?

En general, CUDA tiende a superar a OpenMP en este tipo de cargas matriciales por varias razones:

Paralelismo disponible

OpenMP está limitado por el número de núcleos de CPU (decenas de hilos como máximo en un equipo típico), mientras que CUDA explota **miles de hilos** concurrentes. En Warshall lógico, cada iteración k puede actualizar N^2 celdas, lo que se ajusta de forma natural al paralelismo masivo de GPU.

Ancho de banda y ocultamiento de latencia

Las GPU están diseñadas para alto throughput: cuando un grupo de hilos (warp) espera memoria, el SM puede ejecutar otros warps, ocultando latencias. En CPU, aunque existen cachés y vectorización, el rendimiento suele quedar limitado por el ancho de banda y la presión de memoria al recorrer matrices grandes.

Accesos a memoria y coalescencia

En CUDA, si los hilos contiguos acceden a posiciones contiguas (por ejemplo, celdas con j consecutivo), las lecturas/escrituras pueden coalescerse, aprovechando transacciones eficientes en memoria global. En OpenMP, cada hilo recorre secuencialmente j en su fila; aunque eso favorece localidad, el paralelismo global es menor y la ganancia se estanca al saturar caché/memoria.

Overhead y casos donde OpenMP compite

CUDA tiene overhead de lanzamiento de kernels y transferencias Host–Device (si se miden). Para tamaños moderados o si el cálculo no domina el costo total, OpenMP puede ser competitivo. Para

N grande (carga cúbica), el cómputo domina y CUDA suele despegar.

Oportunidades de mejora en HCUDA (memoria compartida)

La versión actual usa principalmente memoria global. Una optimización típica es *tiling* con **shared memory** para reutilizar segmentos de la fila k (y mejorar el acceso repetido), además de evaluar diferentes configuraciones de bloque para mejorar ocupación.

Conclusiones

- Se implementaron dos versiones paralelas del núcleo: OpenMP (CPU) y CUDA (GPU), manteniendo k secuencial.
- Se dejó un protocolo de medición reproducible con tablas para 10 ejecuciones por tamaño, base para speedup y eficiencia.
- La discusión técnica muestra por qué CUDA suele lograr mayor aceleración en cargas matriciales masivas: más paralelismo y throughput.

Trabajo futuro

- Medir el tiempo CUDA separando cómputo (kernel) de transferencias usando `cudaEvent`.
- Probar variantes de bloque (p. ej., 8x8, 16x16, 32x8) y reportar su impacto.
- Implementar *tiling* con shared memory para reutilización de datos en cada k.

Referencias

[1] OpenMP Architecture Review Board, *OpenMP Application Programming Interface (Specification)*.

[2] NVIDIA, *CUDA C++ Programming Guide*.

[3] NVIDIA, *CUDA C++ Best Practices Guide*.

Código OpenMP (warshall_menu_omp.c)

Listing 1: `warshall_menu_omp.c` – Warshall lógico con OpenMP + menú + medición

```c
// warshall_menu_omp.c
// CPU Paralelo con OPENMP: Cerradura transitiva booleana (Warshall logico)
// + MENu interactivo:
//   (1) ingresar matriz manual + parametros
//   (2) ingresar parametros + grafo random
//   (3) grafo y parametros random
//
// Mantiene modo clasico por argumentos:
//   ./warshall_omp N p seed repeats verify [print]
//
// Compilar (Linux):
//   gcc -O3 -std=c11 -fopenmp warshall_menu_omp.c -o warshall_omp
// (si tu Linux requiere):
//   gcc -O3 -std=c11 -fopenmp warshall_menu_omp.c -o warshall_omp -lrt
//
// Ejecutar:
//   ./warshall_omp
//   ./warshall_omp 1024 0.05 1234 3 0 0
//
// Nota:
// - Paralelizamos el bucle de i (filas) para cada k.
// - Cada hilo escribe solo su fila row_i, no hay condiciones de carrera.
// - k queda secuencial (dependencia entre iteraciones).

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <ctype.h>

#include <omp.h>
```

```
36
37 static inline double seconds_now(void) {
38     struct timespec ts;
39 #if defined(CLOCK_MONOTONIC)
40     if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0) {
41         perror("clock_gettime");
42         exit(EXIT_FAILURE);
43     }
44 #else
45     if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
46         perror("clock_gettime");
47         exit(EXIT_FAILURE);
48     }
49 #endif
50     return (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9;
51 }
52
53 static uint8_t* alloc_matrix(int N) {
54     size_t bytes = (size_t)N * (size_t)N * sizeof(uint8_t);
55     uint8_t* A = (uint8_t*)malloc(bytes);
56     if (!A) {
57         fprintf(stderr, "ERROR: no se pudo asignar %zu bytes para N=%d\n", bytes, N
            );
58         exit(EXIT_FAILURE);
59     }
60     return A;
61 }
62
63 static void init_random(uint8_t* A, int N, double p, unsigned seed) {
64     srand(seed);
65     for (int i = 0; i < N * N; i++) {
66         double r = (double)rand() / (double)RAND_MAX;
67         A[i] = (r < p) ? 1 : 0;
68     }
69 }
70
71 static void print_matrix(const uint8_t* A, int N, const char* title) {
72     printf("\n=== %s (N=%d) ===\n", title, N);
```

```
73
74      printf("      ");
75      for (int j = 0; j < N; j++) printf("%2d ", j);
76      printf("\n");
77
78      printf("      ");
79      for (int j = 0; j < N; j++) printf("---");
80      printf("\n");
81
82      for (int i = 0; i < N; i++) {
83          printf("%2d | ", i);
84          const uint8_t* row = &A[i * N];
85          for (int j = 0; j < N; j++) printf("%2d ", (int)row[j]);
86          printf("\n");
87      }
88  }
89
90  // -------------------------
91  // Nucleo: Warshall logico (OPENMP)
92  // -------------------------
93  void warshall_logical_omp(uint8_t* A, int N) {
94      // A[i][j] = A[i][j] OR (A[i][k] AND A[k][j])
95      // k debe ser secuencial, pero paralelizamos i (cada hilo escribe su fila).
96      for (int k = 0; k < N; k++) {
97          const uint8_t* row_k = &A[k * N];
98
99          #pragma omp parallel for schedule(static)
100         for (int i = 0; i < N; i++) {
101             uint8_t aik = A[i * N + k];
102             if (!aik) continue;
103
104             uint8_t* row_i = &A[i * N];
105             // vectorizable
106             for (int j = 0; j < N; j++) {
107                 row_i[j] = (uint8_t)(row_i[j] | (aik & row_k[j]));
108             }
109         }
110     }
```

```
111 }
112
113 // ------------------------------------
114 // Referencia: cerradura transitiva con BFS
115 // ------------------------------------
116 static void bfs_closure_ref(const uint8_t* Ain, uint8_t* Rref, int N) {
117     int* queue = (int*)malloc((size_t)N * sizeof(int));
118     uint8_t* vis = (uint8_t*)malloc((size_t)N * sizeof(uint8_t));
119     if (!queue || !vis) {
120         fprintf(stderr, "ERROR: memoria insuficiente para BFS\n");
121         free(queue);
122         free(vis);
123         exit(EXIT_FAILURE);
124     }
125
126     for (int s = 0; s < N; s++) {
127         memset(vis, 0, (size_t)N);
128         int front = 0, back = 0;
129
130         const uint8_t* row_s = &Ain[s * N];
131         for (int v = 0; v < N; v++) {
132             if (row_s[v]) {
133                 vis[v] = 1;
134                 queue[back++] = v;
135             }
136         }
137
138         while (front < back) {
139             int u = queue[front++];
140             const uint8_t* row_u = &Ain[u * N];
141             for (int v = 0; v < N; v++) {
142                 if (row_u[v] && !vis[v]) {
143                     vis[v] = 1;
144                     queue[back++] = v;
145                 }
146             }
147         }
148
```

```
149         uint8_t* row_ref = &Rref[s * N];
150         for (int j = 0; j < N; j++) row_ref[j] = vis[j];
151     }
152
153     free(queue);
154     free(vis);
155 }
156
157 static int verify_against_ref(const uint8_t* Rref, const uint8_t* Aout, int N) {
158     for (int i = 0; i < N; i++) {
159         const uint8_t* rr = &Rref[i * N];
160         const uint8_t* ao = &Aout[i * N];
161         for (int j = 0; j < N; j++) {
162             if (rr[j] != ao[j]) {
163                 fprintf(stderr,
164                         "FALLO verificacion: fila i=%d, col j=%d | esperado=%d,
                            obtenido=%d\n",
165                         i, j, (int)rr[j], (int)ao[j]);
166                 return 0;
167             }
168         }
169     }
170     return 1;
171 }
172
173 // ====================================================
174 // Helpers de input robusto (sin scanf)
175 // ====================================================
176 static void read_line(char* buf, size_t n) {
177     if (!fgets(buf, (int)n, stdin)) {
178         printf("\nEOF detectado. Saliendo.\n");
179         exit(0);
180     }
181 }
182
183 static long read_long_prompt(const char* prompt, long minv, long maxv) {
184     char line[256];
185     for (;;) {
```

```
186         printf("%s", prompt);
187         read_line(line, sizeof(line));
188 
189         char* end = NULL;
190         errno = 0;
191         long v = strtol(line, &end, 10);
192         if (errno == 0) {
193             while (end && *end && isspace((unsigned char)*end)) end++;
194             if (end && (*end == '\0' || *end == '\n')) {
195                 if (v >= minv && v <= maxv) return v;
196             }
197         }
198         printf("Entrada␣invalida.␣Rango␣permitido:␣[%ld..%ld]\n", minv, maxv);
199     }
200 }
201 
202 static double read_double_prompt(const char* prompt, double minv, double maxv) {
203     char line[256];
204     for (;;) {
205         printf("%s", prompt);
206         read_line(line, sizeof(line));
207 
208         char* end = NULL;
209         errno = 0;
210         double v = strtod(line, &end);
211         if (errno == 0) {
212             while (end && *end && isspace((unsigned char)*end)) end++;
213             if (end && (*end == '\0' || *end == '\n')) {
214                 if (v >= minv && v <= maxv) return v;
215             }
216         }
217         printf("Entrada␣invalida.␣Rango␣permitido:␣[%.3f..%.3f]\n", minv, maxv);
218     }
219 }
220 
221 static int read_int_prompt(const char* prompt, int minv, int maxv) {
222     return (int)read_long_prompt(prompt, (long)minv, (long)maxv);
223 }
```

```
224
225 static int read_yesno_prompt(const char* prompt) {
226     char line[64];
227     for (;;) {
228         printf("%s (1=si, 0=no): ", prompt);
229         read_line(line, sizeof(line));
230         if (line[0] == '1') return 1;
231         if (line[0] == '0') return 0;
232         printf("Entrada invalida. Escribe 1 o 0.\n");
233     }
234 }
235
236 static int parse_row_01(const char* line, uint8_t* row, int N) {
237     // Acepta: "0 1 0 1" o "0101..." (con o sin espacios)
238     int count = 0;
239     for (const char* p = line; *p && count < N; p++) {
240         if (*p == '0' || *p == '1') {
241             row[count++] = (uint8_t)(*p - '0');
242         }
243     }
244     return (count == N);
245 }
246
247 static double density_ones(const uint8_t* A, int N) {
248     long long ones = 0;
249     for (long long i = 0; i < (long long)N * (long long)N; i++) ones += A[i] ? 1 :
        0;
250     return (double)ones / (double)((long long)N * (long long)N);
251 }
252
253 // ====================================================
254 // Ejecutar en modo verify/timing
255 // ====================================================
256 static int run_experiment(uint8_t* Ain, int N, double p, unsigned seed, int repeats
    , int verify, int print) {
257     const int PRINT_LIMIT = 16;
258
259     if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
```

```
260     if (repeats <= 0) repeats = 1;
261
262     uint8_t* A = alloc_matrix(N);
263
264     if (verify) {
265         if (N > 128) {
266             printf("Aviso: verificacion activada con N=%d; se recomienda N<=128.\n"
                    , N);
267         }
268
269         memcpy(A, Ain, (size_t)N * (size_t)N);
270
271         double t0 = seconds_now();
272         warshall_logical_omp(A, N);
273         double t1 = seconds_now();
274         double kernel_time = t1 - t0;
275
276         uint8_t* Rref = alloc_matrix(N);
277         bfs_closure_ref(Ain, Rref, N);
278
279         int ok = verify_against_ref(Rref, A, N);
280
281         if (print) {
282             print_matrix(Ain,  N, "MATRIZ DE ENTRADA (Grafo / Adyacencia)");
283             print_matrix(Rref, N, "MATRIZ DE VERIFICACIoN (Referencia BFS)");
284             print_matrix(A,    N, "MATRIZ DE SALIDA (Warshall logico) [OPENMP]");
285         }
286
287         printf("\nVALIDACIoN (BFS) para N=%d: %s\n", N, ok ? "OK" : "FALLIDA");
288         printf("Tiempo del nucleo (warshall_logical_omp): %.6f s\n", kernel_time);
289         printf("Resumen params | N=%d | p=%.3f | seed=%u | repeats=%d | verify=%d |
                print=%d\n",
290                N, p, seed, repeats, verify, print);
291
292         free(Rref);
293         free(A);
294         return ok ? EXIT_SUCCESS : EXIT_FAILURE;
295     }
```

```
296
297     double best = 1e100;
298     for (int r = 0; r < repeats; r++) {
299         memcpy(A, Ain, (size_t)N * (size_t)N);
300         double t0 = seconds_now();
301         warshall_logical_omp(A, N);
302         double t1 = seconds_now();
303         double dt = t1 - t0;
304         if (dt < best) best = dt;
305     }
306
307     printf("OPENMP Warshall logico | N=%d | p=%.3f | seed=%u | repeats=%d |
        best_kernel_time=%.6f s | threads=%d\n",
308            N, p, seed, repeats, best, omp_get_max_threads());
309
310     free(A);
311     return EXIT_SUCCESS;
312 }
313
314 // ========================================================
315 // Menu
316 // ========================================================
317 static void menu_loop(void) {
318     for (;;) {
319         printf("\n==============================\n");
320         printf("   MENu - Warshall logico OPENMP\n");
321         printf("==============================\n");
322         printf("1) Ingresar MATRIZ manual + parametros\n");
323         printf("2) Ingresar parametros + GRAFO random\n");
324         printf("3) Grafo y parametros RANDOM\n");
325         printf("0) Salir\n");
326
327         int opt = read_int_prompt("Opcion: ", 0, 3);
328         if (opt == 0) break;
329
330         int N = 256;
331         double p = 0.05;
332         unsigned seed = 1234;
```

```
333         int repeats = 3;
334         int verify = 0;
335         int print = 0;
336 
337         uint8_t* Ain = NULL;
338 
339         if (opt == 1) {
340             N = read_int_prompt("Ingrese N (1..2048 recomendado): ", 1, 4096);
341             Ain = alloc_matrix(N);
342 
343             printf("\nIngrese la matriz de adyacencia (%dx%d) con 0/1.\n", N, N);
344             printf("Formato permitido por fila: '0 1 0 1' o '0101...'\n\n");
345 
346             char line[8192];
347             for (int i = 0; i < N; i++) {
348                 for (;;) {
349                     printf("Fila %d: ", i);
350                     read_line(line, sizeof(line));
351                     if (parse_row_01(line, &Ain[i * N], N)) break;
352                     printf("Fila invalida. Debe contener %d valores 0/1.\n", N);
353                 }
354             }
355 
356             p = density_ones(Ain, N);
357             seed = 0;
358 
359             repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
360             verify  = read_yesno_prompt("verify");
361             print   = read_yesno_prompt("print");
362 
363             printf("\nDensidad p calculada desde la matriz: %.3f\n", p);
364         }
365         else if (opt == 2) {
366             N = read_int_prompt("N (1..4096): ", 1, 4096);
367             p = read_double_prompt("p (0..1): ", 0.0, 1.0);
368             seed = (unsigned)read_long_prompt("seed (0..2^31-1): ", 0, 2147483647L)
                ;
369             repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
```

```
370             verify  = read_yesno_prompt("verify");
371             print   = read_yesno_prompt("print");
372
373             Ain = alloc_matrix(N);
374             init_random(Ain, N, p, seed);
375         }
376         else if (opt == 3) {
377             unsigned s = (unsigned)time(NULL);
378             srand(s);
379
380             int choices[] = {8,16,32,64,128,256,512,1024};
381             int idx = rand() % (int)(sizeof(choices)/sizeof(choices[0]));
382             N = choices[idx];
383
384             p = 0.01 + ((double)rand() / (double)RAND_MAX) * 0.19; // [0.01..0.20]
385             seed = (unsigned)rand();
386             repeats = 1 + (rand() % 7);
387
388             verify = (N <= 128) ? 1 : 0;
389             print  = (N <= 16) ? 1 : 0;
390
391             Ain = alloc_matrix(N);
392             init_random(Ain, N, p, seed);
393
394             printf("\nParametros␣random␣generados:\n");
395             printf("N=%d␣|␣p=%.3f␣|␣seed=%u␣|␣repeats=%d␣|␣verify=%d␣|␣print=%d\n",
396                    N, p, seed, repeats, verify, print);
397         }
398
399         int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
400         free(Ain);
401
402         if (rc != EXIT_SUCCESS) {
403             printf("Ejecucion␣termino␣con␣error␣(validacion␣fallida␣o␣problema).\n"
                );
404         }
405
406         if (!read_yesno_prompt("\nDeseas␣ejecutar␣otra␣vez")) break;
```

```
407     }
408 }
409
410 int main(int argc, char** argv) {
411     // --- Modo por argumentos ---
412     if (argc >= 2) {
413         int N = 256;
414         double p = 0.05;
415         unsigned seed = 1234;
416         int repeats = 3;
417         int verify = 0;
418         int print = 0;
419
420         if (argc >= 2) N = atoi(argv[1]);
421         if (argc >= 3) p = atof(argv[2]);
422         if (argc >= 4) seed = (unsigned)atoi(argv[3]);
423         if (argc >= 5) repeats = atoi(argv[4]);
424         if (argc >= 6) verify = atoi(argv[5]);
425         if (argc >= 7) print = atoi(argv[6]);
426
427         if (N <= 0) { fprintf(stderr, "ERROR: N debe ser > 0\n"); return
                EXIT_FAILURE; }
428         if (p < 0.0 || p > 1.0) { fprintf(stderr, "ERROR: p debe estar en [0,1]\n")
                ; return EXIT_FAILURE; }
429         if (repeats <= 0) repeats = 1;
430
431         const int PRINT_LIMIT = 16;
432         if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
433
434         uint8_t* Ain = alloc_matrix(N);
435         init_random(Ain, N, p, seed);
436
437         int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
438         free(Ain);
439         return rc;
440     }
441
442     // --- Modo menu ---
```

```
443     menu_loop();
444     return 0;
445 }
```

Código CUDA (warshall_menu_cuda.cu)

Listing 2: `warshall_menu_cuda.cu` – Warshall lógico con CUDA + menú + medición

```
1  // warshall_menu_cuda.cu
2  // CUDA: Cerradura transitiva booleana (Warshall logico) + MENU interactivo
3  //
4  // Opciones de menu:
5  //   (1) ingresar matriz manual + parametros
6  //   (2) ingresar parametros + grafo random
7  //   (3) grafo y parametros random
8  //
9  // Modo por argumentos (como antes):
10 //   ./warshall_cuda N p seed repeats verify [print]
11 //
12 // Compilar (Linux):
13 //   nvcc -O3 -std=c++17 warshall_menu_cuda.cu -o warshall_cuda
14 // (si tu Linux requiere -lrt):
15 //   nvcc -O3 -std=c++17 warshall_menu_cuda.cu -o warshall_cuda -lrt
16 //
17 // Ejecutar:
18 //   ./warshall_cuda
19 //   ./warshall_cuda 1024 0.05 1234 3 0 0
20 //
21 // Nota de paralelizacion:
22 // - k se mantiene SECUENCIAL (dependencia entre iteraciones).
23 // - Para cada k, se lanza un kernel 2D que actualiza TODAS las celdas (i,j) en
       paralelo.
24 //
25 // IMPORTANTE (correctitud):
26 // - Este kernel lee A[i,k] y A[k,j] del MISMO buffer A que tambien se escribe.
27 //   Eso replica tu version CUDA didactica anterior, y suele funcionar para
       Warshall booleano,
28 //   pero si quieres "paso k limpio" (lectura desde snapshot), usa doble buffer (
       Ain->Aout)
```

```
29 //   por cada k (mas lento por copias). Aqui dejo la version simple y rapida (in-
       place).
30
31 #define _POSIX_C_SOURCE 200809L
32
33 #include <cuda_runtime.h>
34 #include <cstdio>
35 #include <cstdlib>
36 #include <cstdint>
37 #include <cstring>
38 #include <ctime>
39 #include <cerrno>
40 #include <cctype>
41 #include <iostream>
42
43 static inline void CUDA_CHECK(cudaError_t e, const char* file, int line) {
44     if (e != cudaSuccess) {
45         std::fprintf(stderr, "CUDA error %s:%d: %s\n", file, line,
              cudaGetErrorString(e));
46         std::exit(EXIT_FAILURE);
47     }
48 }
49 #define CUDA_CALL(x) CUDA_CHECK((x), __FILE__, __LINE__)
50
51 // ================================================
52 // Timing (CPU wall time) para medir total del "nucleo"
53 // ================================================
54 static inline double seconds_now(void) {
55     struct timespec ts;
56 #if defined(CLOCK_MONOTONIC)
57     if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0) {
58         perror("clock_gettime");
59         std::exit(EXIT_FAILURE);
60     }
61 #else
62     if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
63         perror("clock_gettime");
64         std::exit(EXIT_FAILURE);
```

```
65      }
66  #endif
67      return (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9;
68  }
69
70  // ====================================================
71  // Memoria / utilidades
72  // ====================================================
73  static uint8_t* alloc_matrix(int N) {
74      size_t bytes = (size_t)N * (size_t)N * sizeof(uint8_t);
75      uint8_t* A = (uint8_t*)std::malloc(bytes);
76      if (!A) {
77          std::fprintf(stderr, "ERROR: no se pudo asignar %zu bytes para N=%d\n",
                bytes, N);
78          std::exit(EXIT_FAILURE);
79      }
80      return A;
81  }
82
83  static void init_random(uint8_t* A, int N, double p, unsigned seed) {
84      std::srand(seed);
85      for (int i = 0; i < N * N; i++) {
86          double r = (double)std::rand() / (double)RAND_MAX;
87          A[i] = (r < p) ? 1 : 0;
88      }
89  }
90
91  static void print_matrix(const uint8_t* A, int N, const char* title) {
92      std::printf("\n=== %s (N=%d) ===\n", title, N);
93
94      std::printf("     ");
95      for (int j = 0; j < N; j++) std::printf("%2d ", j);
96      std::printf("\n");
97
98      std::printf("     ");
99      for (int j = 0; j < N; j++) std::printf("---");
100     std::printf("\n");
101
```

```
    for (int i = 0; i < N; i++) {
        std::printf("%2d | ", i);
        const uint8_t* row = &A[i * N];
        for (int j = 0; j < N; j++) std::printf("%2d ", (int)row[j]);
        std::printf("\n");
    }
}

// ====================================================
// CUDA kernel: 1 paso k (in-place)
// ====================================================
__global__ void warshall_step_u8(uint8_t* A, int N, int k) {
    int j = blockIdx.x * blockDim.x + threadIdx.x; // col
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    if (i < N && j < N) {
        uint8_t aik = A[i * N + k];
        if (!aik) return; // pequeno atajo
        uint8_t akj = A[k * N + j];
        uint8_t aij = A[i * N + j];
        A[i * N + j] = (uint8_t)(aij | (aik & akj));
    }
}

// ====================================================
// Nucleo: Warshall logico (CUDA)
// ====================================================
static void warshall_logical_cuda(uint8_t* A_host, int N) {
    size_t bytes = (size_t)N * (size_t)N * sizeof(uint8_t);

    uint8_t* d_A = nullptr;
    CUDA_CALL(cudaMalloc((void**)&d_A, bytes));
    CUDA_CALL(cudaMemcpy(d_A, A_host, bytes, cudaMemcpyHostToDevice));

    dim3 block(16, 16);
    dim3 grid((N + block.x - 1) / block.x,
              (N + block.y - 1) / block.y);

    for (int k = 0; k < N; k++) {
```

```
140         warshall_step_u8<<<grid, block>>>(d_A, N, k);
141         CUDA_CALL(cudaGetLastError());
142         CUDA_CALL(cudaDeviceSynchronize()); // claridad/didactica
143     }
144
145     CUDA_CALL(cudaMemcpy(A_host, d_A, bytes, cudaMemcpyDeviceToHost));
146     CUDA_CALL(cudaFree(d_A));
147 }
148
149 // ------------------------------------
150 // Referencia: cerradura transitiva con BFS (CPU)
151 // ------------------------------------
152 static void bfs_closure_ref(const uint8_t* Ain, uint8_t* Rref, int N) {
153     int* queue = (int*)std::malloc((size_t)N * sizeof(int));
154     uint8_t* vis = (uint8_t*)std::malloc((size_t)N * sizeof(uint8_t));
155     if (!queue || !vis) {
156         std::fprintf(stderr, "ERROR: memoria insuficiente para BFS\n");
157         std::free(queue);
158         std::free(vis);
159         std::exit(EXIT_FAILURE);
160     }
161
162     for (int s = 0; s < N; s++) {
163         std::memset(vis, 0, (size_t)N);
164         int front = 0, back = 0;
165
166         const uint8_t* row_s = &Ain[s * N];
167         for (int v = 0; v < N; v++) {
168             if (row_s[v]) {
169                 vis[v] = 1;
170                 queue[back++] = v;
171             }
172         }
173
174         while (front < back) {
175             int u = queue[front++];
176             const uint8_t* row_u = &Ain[u * N];
177             for (int v = 0; v < N; v++) {
```

```
178                 if (row_u[v] && !vis[v]) {
179                     vis[v] = 1;
180                     queue[back++] = v;
181                 }
182             }
183         }
184 
185         uint8_t* row_ref = &Rref[s * N];
186         for (int j = 0; j < N; j++) row_ref[j] = vis[j];
187     }
188 
189     std::free(queue);
190     std::free(vis);
191 }
192 
193 static int verify_against_ref(const uint8_t* Rref, const uint8_t* Aout, int N) {
194     for (int i = 0; i < N; i++) {
195         const uint8_t* rr = &Rref[i * N];
196         const uint8_t* ao = &Aout[i * N];
197         for (int j = 0; j < N; j++) {
198             if (rr[j] != ao[j]) {
199                 std::fprintf(stderr,
200                         "FALLO verificacion: fila i=%d, col j=%d | esperado=%d,
                            obtenido=%d\n",
201                         i, j, (int)rr[j], (int)ao[j]);
202                 return 0;
203             }
204         }
205     }
206     return 1;
207 }
208 
209 // ====================================================
210 // Helpers de input robusto (sin scanf)
211 // ====================================================
212 static void read_line(char* buf, size_t n) {
213     if (!std::fgets(buf, (int)n, stdin)) {
214         std::printf("\nEOF detectado. Saliendo.\n");
```

```
215         std::exit(0);
216     }
217 }
218
219 static long read_long_prompt(const char* prompt, long minv, long maxv) {
220     char line[256];
221     for (;;) {
222         std::printf("%s", prompt);
223         read_line(line, sizeof(line));
224
225         char* end = NULL;
226         errno = 0;
227         long v = std::strtol(line, &end, 10);
228         if (errno == 0) {
229             while (end && *end && std::isspace((unsigned char)*end)) end++;
230             if (end && (*end == '\0' || *end == '\n')) {
231                 if (v >= minv && v <= maxv) return v;
232             }
233         }
234         std::printf("Entrada␣invalida.␣Rango␣permitido:␣[%ld..%ld]\n", minv, maxv);
235     }
236 }
237
238 static double read_double_prompt(const char* prompt, double minv, double maxv) {
239     char line[256];
240     for (;;) {
241         std::printf("%s", prompt);
242         read_line(line, sizeof(line));
243
244         char* end = NULL;
245         errno = 0;
246         double v = std::strtod(line, &end);
247         if (errno == 0) {
248             while (end && *end && std::isspace((unsigned char)*end)) end++;
249             if (end && (*end == '\0' || *end == '\n')) {
250                 if (v >= minv && v <= maxv) return v;
251             }
252         }
```

```
253         std::printf("Entrada␣invalida.␣Rango␣permitido:␣[%.3f..%.3f]\n", minv, maxv
                );
254     }
255 }
256
257 static int read_int_prompt(const char* prompt, int minv, int maxv) {
258     return (int)read_long_prompt(prompt, (long)minv, (long)maxv);
259 }
260
261 static int read_yesno_prompt(const char* prompt) {
262     char line[64];
263     for (;;) {
264         std::printf("%s␣(1=si,␣0=no):␣", prompt);
265         read_line(line, sizeof(line));
266         if (line[0] == '1') return 1;
267         if (line[0] == '0') return 0;
268         std::printf("Entrada␣invalida.␣Escribe␣1␣o␣0.\n");
269     }
270 }
271
272 static int parse_row_01(const char* line, uint8_t* row, int N) {
273     int count = 0;
274     for (const char* p = line; *p && count < N; p++) {
275         if (*p == '0' || *p == '1') row[count++] = (uint8_t)(*p - '0');
276     }
277     return (count == N);
278 }
279
280 static double density_ones(const uint8_t* A, int N) {
281     long long ones = 0;
282     for (long long i = 0; i < (long long)N * (long long)N; i++) ones += A[i] ? 1 :
            0;
283     return (double)ones / (double)((long long)N * (long long)N);
284 }
285
286 // ====================================================
287 // Ejecutar en modo verify/timing (CUDA)
288 // ====================================================
```

```
289 static int run_experiment(uint8_t* Ain, int N, double p, unsigned seed, int repeats
        , int verify, int print) {
290     const int PRINT_LIMIT = 16;
291 
292     if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
293     if (repeats <= 0) repeats = 1;
294 
295     uint8_t* A = alloc_matrix(N);
296 
297     if (verify) {
298         if (N > 128) {
299             std::printf("Aviso: verificacion activada con N=%d; se recomienda N
                <=128.\n", N);
300         }
301 
302         std::memcpy(A, Ain, (size_t)N * (size_t)N);
303 
304         double t0 = seconds_now();
305         warshall_logical_cuda(A, N);
306         double t1 = seconds_now();
307         double kernel_time = t1 - t0;
308 
309         uint8_t* Rref = alloc_matrix(N);
310         bfs_closure_ref(Ain, Rref, N);
311 
312         int ok = verify_against_ref(Rref, A, N);
313 
314         if (print) {
315             print_matrix(Ain,  N, "MATRIZ DE ENTRADA (Grafo / Adyacencia)");
316             print_matrix(Rref, N, "MATRIZ DE VERIFICACIoN (Referencia BFS)");
317             print_matrix(A,    N, "MATRIZ DE SALIDA (Warshall logico) [CUDA]");
318         }
319 
320         std::printf("\nVALIDACIoN (BFS) para N=%d: %s\n", N, ok ? "OK" : "FALLIDA")
            ;
321         std::printf("Tiempo del nucleo (warshall_logical_cuda): %.6f s\n",
            kernel_time);
```

```
322         std::printf("Resumen params | N=%d | p=%.3f | seed=%u | repeats=%d | verify
                =%d | print=%d\n",
323                 N, p, seed, repeats, verify, print);
324
325         std::free(Rref);
326         std::free(A);
327         return ok ? EXIT_SUCCESS : EXIT_FAILURE;
328     }
329
330     double best = 1e100;
331     for (int r = 0; r < repeats; r++) {
332         std::memcpy(A, Ain, (size_t)N * (size_t)N);
333         double t0 = seconds_now();
334         warshall_logical_cuda(A, N);
335         double t1 = seconds_now();
336         double dt = t1 - t0;
337         if (dt < best) best = dt;
338     }
339
340     std::printf("CUDA Warshall logico | N=%d | p=%.3f | seed=%u | repeats=%d |
            best_kernel_time=%.6f s\n",
341                 N, p, seed, repeats, best);
342
343     std::free(A);
344     return EXIT_SUCCESS;
345 }
346
347 // ====================================================
348 // Menu
349 // ====================================================
350 static void menu_loop(void) {
351     for (;;) {
352         std::printf("\n==============================\n");
353         std::printf("   MENu - Warshall logico CUDA\n");
354         std::printf("==============================\n");
355         std::printf("1) Ingresar MATRIZ manual + parametros\n");
356         std::printf("2) Ingresar parametros + GRAFO random\n");
357         std::printf("3) Grafo y parametros RANDOM\n");
```

```
358         std::printf("0) Salir\n");
359
360         int opt = read_int_prompt("Opcion: ", 0, 3);
361         if (opt == 0) break;
362
363         int N = 256;
364         double p = 0.05;
365         unsigned seed = 1234;
366         int repeats = 3;
367         int verify = 0;
368         int print = 0;
369
370         uint8_t* Ain = NULL;
371
372         if (opt == 1) {
373             N = read_int_prompt("Ingrese N (1..2048 recomendado): ", 1, 4096);
374             Ain = alloc_matrix(N);
375
376             std::printf("\nIngrese la matriz de adyacencia (%dx%d) con 0/1.\n", N,
                    N);
377             std::printf("Formato permitido por fila: '0 1 0 1' o '0101...'\n\n");
378
379             char line[8192];
380             for (int i = 0; i < N; i++) {
381                 for (;;) {
382                     std::printf("Fila %d: ", i);
383                     read_line(line, sizeof(line));
384                     if (parse_row_01(line, &Ain[i * N], N)) break;
385                     std::printf("Fila invalida. Debe contener %d valores 0/1.\n", N
                            );
386                 }
387             }
388
389             p = density_ones(Ain, N);
390             seed = 0;
391
392             repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
393             verify  = read_yesno_prompt("verify");
```

```
394             print   = read_yesno_prompt("print");
395
396             std::printf("\nDensidad p calculada desde la matriz: %.3f\n", p);
397         }
398         else if (opt == 2) {
399             N = read_int_prompt("N (1..4096): ", 1, 4096);
400             p = read_double_prompt("p (0..1): ", 0.0, 1.0);
401             seed = (unsigned)read_long_prompt("seed (0..2^31-1): ", 0, 2147483647L)
                    ;
402             repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
403             verify  = read_yesno_prompt("verify");
404             print   = read_yesno_prompt("print");
405
406             Ain = alloc_matrix(N);
407             init_random(Ain, N, p, seed);
408         }
409         else if (opt == 3) {
410             unsigned s = (unsigned)time(NULL);
411             srand(s);
412
413             int choices[] = {8,16,32,64,128,256,512,1024};
414             int idx = rand() % (int)(sizeof(choices)/sizeof(choices[0]));
415             N = choices[idx];
416
417             p = 0.01 + ((double)rand() / (double)RAND_MAX) * 0.19; // [0.01..0.20]
418             seed = (unsigned)rand();
419             repeats = 1 + (rand() % 7);
420
421             verify = (N <= 128) ? 1 : 0;
422             print  = (N <= 16) ? 1 : 0;
423
424             Ain = alloc_matrix(N);
425             init_random(Ain, N, p, seed);
426
427             std::printf("\nParametros random generados:\n");
428             std::printf("N= %d | p= %.3f | seed= %u | repeats= %d | verify= %d | print= %
                    d\n",
429                         N, p, seed, repeats, verify, print);
```

```
430         }
431
432         int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
433         std::free(Ain);
434
435         if (rc != EXIT_SUCCESS) {
436             std::printf("Ejecucion termino con error (validacion fallida o problema
                ).\n");
437         }
438
439         if (!read_yesno_prompt("\nDeseas ejecutar otra vez")) break;
440     }
441 }
442
443 int main(int argc, char** argv) {
444     // --- Modo por argumentos ---
445     if (argc >= 2) {
446         int N = 256;
447         double p = 0.05;
448         unsigned seed = 1234;
449         int repeats = 3;
450         int verify = 0;
451         int print = 0;
452
453         if (argc >= 2) N = std::atoi(argv[1]);
454         if (argc >= 3) p = std::atof(argv[2]);
455         if (argc >= 4) seed = (unsigned)std::atoi(argv[3]);
456         if (argc >= 5) repeats = std::atoi(argv[4]);
457         if (argc >= 6) verify = std::atoi(argv[5]);
458         if (argc >= 7) print = std::atoi(argv[6]);
459
460         if (N <= 0) { std::fprintf(stderr, "ERROR: N debe ser > 0\n"); return
                EXIT_FAILURE; }
461         if (p < 0.0 || p > 1.0) { std::fprintf(stderr, "ERROR: p debe estar en
                [0,1]\n"); return EXIT_FAILURE; }
462         if (repeats <= 0) repeats = 1;
463
464         const int PRINT_LIMIT = 16;
```

```
465         if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
466
467         uint8_t* Ain = alloc_matrix(N);
468         init_random(Ain, N, p, seed);
469
470         int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
471         std::free(Ain);
472         return rc;
473     }
474
475     // --- Modo menu ---
476     menu_loop();
477     return 0;
478 }
```