

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  
Mayhuire Chacon, Brenda Lucia  
Mendoza Quispe, Jose Daniel  
Perez Cahuana, Gabriel  
Zevallos Yanqui, Andy Jefferson



## Índice

<b>Resumen</b>	<b>4</b>
<b>1. Introducción</b>	<b>5</b>
<b>2. Objetivos</b>	<b>5</b>
2.1. Objetivo general . . . . .	5
2.2. Objetivos específicos . . . . .	5
<b>3. Marco teórico (solo OpenMP y CUDA)</b>	<b>5</b>
3.1. OpenMP: paralelismo en memoria compartida . . . . .	5
3.2. CUDA: paralelismo masivo en GPU . . . . .	6
<b>4. Implementación paralela (Fase 2)</b>	<b>6</b>
4.1. Estrategia común y correctitud . . . . .	6
4.2. Versión OpenMP (CPU) . . . . .	7
4.3. Versión CUDA (GPU) . . . . .	7
4.4. Notas sobre medición de tiempo . . . . .	7
<b>5. Evaluación y métricas</b>	<b>7</b>
5.1. Diseño experimental . . . . .	7
5.2. Tablas de tiempos (llenado manual) . . . . .	8
5.3. Cálculo de métricas . . . . .	8
5.4. Análisis del rendimiento (CUDA) . . . . .	8
<b>6. Gráficos (plantillas)</b>	<b>9</b>
<b>7. Discusión: ¿por qué CUDA suele ser más rápido que OpenMP?</b>	<b>9</b>
7.1. Paralelismo disponible . . . . .	9
7.2. Ancho de banda y ocultamiento de latencia . . . . .	9
7.3. Accesos a memoria y coalescencia . . . . .	9
7.4. Overhead y casos donde OpenMP compite . . . . .	9
7.5. Oportunidades de mejora en HCUDA (memoria compartida) . . . . .	10
<b>8. Conclusiones</b>	<b>11</b>
<b>9. Trabajo futuro</b>	<b>11</b>
<b>A. Código OpenMP (warshall_menu_omp.c)</b>	<b>13</b>





## Índice de figuras



## Índice de cuadros

1.	Tiempos de ejecución (10 repeticiones) – OpenMP . . . . .	8
2.	Tiempos de ejecución (10 repeticiones) – CUDA . . . . .	8



## 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 paralleliza 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 parallelizar 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) – 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 2: 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

### Cálculo de métricas

Para cada tamaño del problema  $N$ , se utiliza el tiempo promedio  $\bar{t}$  obtenido en las Tablas 1 y 2 como tiempo paralelo  $T_{par}$ .

**Aceleración (Speedup)..** La aceleración se calcula como:

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

donde  $T_{seq}$  es el tiempo secuencial y  $T_{par}$  corresponde a OpenMP o CUDA.

**Eficiencia (OpenMP)..** Para OpenMP, la eficiencia se define como:

$$E(N) = \frac{S(N)}{p}$$

donde  $p$  es el número de hilos utilizados.

### Análisis del rendimiento (CUDA)

Para CUDA se analiza:



- configuración de bloques y threads por bloque (por defecto 16x16),
- accesos a memoria global y coalescencia (hilos contiguos en  $j$ ),
- sincronización por iteración  $k$  (en el código se fuerza `cudaDeviceSynchronize`).

### Gráficos (plantillas)

Una vez completadas las tablas, se recomienda graficar:

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

```
1 // warshall_menu_omp.c
2 // CPU Paralelo con OPENMP: Cerradura transitiva booleana (Warshall logico)
3 // + MENú interactivo:
4 //   (1) ingresar matriz manual + parametros
5 //   (2) ingresar parametros + grafo random
6 //   (3) grafo y parametros random
7 //
8 // Mantiene modo clasico por argumentos:
9 //   ./warshall_omp N p seed repeats verify [print]
10 //
11 // Compilar (Linux):
12 //   gcc -O3 -std=c11 -fopenmp warshall_menu_omp.c -o warshall_omp
13 // (si tu Linux requiere):
14 //   gcc -O3 -std=c11 -fopenmp warshall_menu_omp.c -o warshall_omp -lrt
15 //
16 // Ejecutar:
17 //   ./warshall_omp
18 //   ./warshall_omp 1024 0.05 1234 3 0 0
19 //
20 // Nota:
21 // - Paralelizamos el bucle de i (filas) para cada k.
22 // - Cada hilo escribe solo su fila row_i, no hay condiciones de carrera.
23 // - k queda secuencial (dependencia entre iteraciones).
24
25 #define _POSIX_C_SOURCE 200809L
26
27 #include <stdio.h>
28 #include <stdlib.h>
29 #include <stdint.h>
30 #include <string.h>
31 #include <time.h>
32 #include <errno.h>
33 #include <ctype.h>
34
35 #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                 );
59         exit(EXIT_FAILURE);
60     }
61     return A;
62 }
63
64 static void init_random(uint8_t* A, int N, double p, unsigned seed) {
65     srand(seed);
66     for (int i = 0; i < N * N; i++) {
67         double r = (double)rand() / (double)RAND_MAX;
68         A[i] = (r < p) ? 1 : 0;
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("ooooo");
75     for (int j = 0; j < N; j++) printf("%2d ", j);
76     printf("\n");
77
78     printf("ooooo");
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
151     for (int j = 0; j < N; j++) row_ref[j] = vis[j];
152
153     free(queue);
154
155 }
156
157 static int verify_against_ref(const uint8_t* Rref, const uint8_t* Aout, int N) {
158
159     for (int i = 0; i < N; i++) {
160
161         const uint8_t* rr = &Rref[i * N];
162
163         const uint8_t* ao = &Aout[i * N];
164
165         for (int j = 0; j < N; j++) {
166
167             if (rr[j] != ao[j]) {
168
169                 fprintf(stderr,
170                         "FALLO_verificacion:_fila_i=%d,_col_j=%d_|_esperado=%d,_"
171                         "obtenido=%d\n",
172                         i, j, (int)rr[j], (int)ao[j]);
173
174             }
175
176         }
177
178         return 0;
179
180     }
181
182
183     static void read_line(char* buf, size_t n) {
184
185         if (!fgets(buf, (int)n, stdin)) {
186
187             printf("\nEOF_detectado._Saliendo.\n");
188
189             exit(0);
190
191         }
192
193
194         static long read_long_prompt(const char* prompt, long minv, long maxv) {
195
196             char line[256];
197
198             for (;;) {
199
200                 if (fscanf(stdin, "%ld", &val) == 1) {
201
202                     if (val < minv || val > maxv) {
203
204                         printf("El valor debe estar entre %ld y %ld.\n", minv, maxv);
205
206                         continue;
207
208                     }
209
210                     break;
211
212                 }
213
214             }
215
216             return val;
217
218         }
219
220
221         static void print_usage() {
222
223             printf("Uso: ./matrixmult [filas A] [columnas B] [columnas C]\n");
224
225             exit(1);
226
227         }
228
229
230         if (argc != 4) {
231
232             print_usage();
233
234         }
235
236         else {
237
238             int s = atoi(argv[1]);
239
240             int t = atoi(argv[2]);
241
242             int u = atoi(argv[3]);
243
244             if (s < 1 || s > 1000000000) {
245
246                 print_usage();
247
248             }
249
250             if (t < 1 || t > 1000000000) {
251
252                 print_usage();
253
254             }
255
256             if (u < 1 || u > 1000000000) {
257
258                 print_usage();
259
260             }
261
262             if (s != t) {
263
264                 print_usage();
265
266             }
267
268             if (s * t != u) {
269
270                 print_usage();
271
272             }
273
274             if (s > 1000000000) {
275
276                 print_usage();
277
278             }
279
280             if (t > 1000000000) {
281
282                 print_usage();
283
284             }
285
286             if (u > 1000000000) {
287
288                 print_usage();
289
290             }
291
292             if (s > 1000000000) {
293
294                 print_usage();
295
296             }
297
298             if (t > 1000000000) {
299
300                 print_usage();
301
302             }
303
304             if (u > 1000000000) {
305
306                 print_usage();
307
308             }
309
310             if (s > 1000000000) {
311
312                 print_usage();
313
314             }
315
316             if (t > 1000000000) {
317
318                 print_usage();
319
320             }
321
322             if (u > 1000000000) {
323
324                 print_usage();
325
326             }
327
328             if (s > 1000000000) {
329
330                 print_usage();
331
332             }
333
334             if (t > 1000000000) {
335
336                 print_usage();
337
338             }
339
340             if (u > 1000000000) {
341
342                 print_usage();
343
344             }
345
346             if (s > 1000000000) {
347
348                 print_usage();
349
350             }
351
352             if (t > 1000000000) {
353
354                 print_usage();
355
356             }
357
358             if (u > 1000000000) {
359
360                 print_usage();
361
362             }
363
364             if (s > 1000000000) {
365
366                 print_usage();
367
368             }
369
370             if (t > 1000000000) {
371
372                 print_usage();
373
374             }
375
376             if (u > 1000000000) {
377
378                 print_usage();
379
380             }
381
382             if (s > 1000000000) {
383
384                 print_usage();
385
386             }
387
388             if (t > 1000000000) {
389
390                 print_usage();
391
392             }
393
394             if (u > 1000000000) {
395
396                 print_usage();
397
398             }
399
400             if (s > 1000000000) {
401
402                 print_usage();
403
404             }
405
406             if (t > 1000000000) {
407
408                 print_usage();
409
410             }
411
412             if (u > 1000000000) {
413
414                 print_usage();
415
416             }
417
418             if (s > 1000000000) {
419
420                 print_usage();
421
422             }
423
424             if (t > 1000000000) {
425
426                 print_usage();
427
428             }
429
430             if (u > 1000000000) {
431
432                 print_usage();
433
434             }
435
436             if (s > 1000000000) {
437
438                 print_usage();
439
440             }
441
442             if (t > 1000000000) {
443
444                 print_usage();
445
446             }
447
448             if (u > 1000000000) {
449
450                 print_usage();
451
452             }
453
454             if (s > 1000000000) {
455
456                 print_usage();
457
458             }
459
460             if (t > 1000000000) {
461
462                 print_usage();
463
464             }
465
466             if (u > 1000000000) {
467
468                 print_usage();
469
470             }
471
472             if (s > 1000000000) {
473
474                 print_usage();
475
476             }
477
478             if (t > 1000000000) {
479
480                 print_usage();
481
482             }
483
484             if (u > 1000000000) {
485
486                 print_usage();
487
488             }
489
490             if (s > 1000000000) {
491
492                 print_usage();
493
494             }
495
496             if (t > 1000000000) {
497
498                 print_usage();
499
500             }
501
502             if (u > 1000000000) {
503
504                 print_usage();
505
506             }
507
508             if (s > 1000000000) {
509
510                 print_usage();
511
512             }
513
514             if (t > 1000000000) {
515
516                 print_usage();
517
518             }
519
520             if (u > 1000000000) {
521
522                 print_usage();
523
524             }
525
526             if (s > 1000000000) {
527
528                 print_usage();
529
530             }
531
532             if (t > 1000000000) {
533
534                 print_usage();
535
536             }
537
538             if (u > 1000000000) {
539
540                 print_usage();
541
542             }
543
544             if (s > 1000000000) {
545
546                 print_usage();
547
548             }
549
550             if (t > 1000000000) {
551
552                 print_usage();
553
554             }
555
556             if (u > 1000000000) {
557
558                 print_usage();
559
560             }
561
562             if (s > 1000000000) {
563
564                 print_usage();
565
566             }
567
568             if (t > 1000000000) {
569
570                 print_usage();
571
572             }
573
574             if (u > 1000000000) {
575
576                 print_usage();
577
578             }
579
580             if (s > 1000000000) {
581
582                 print_usage();
583
584             }
585
586             if (t > 1000000000) {
587
588                 print_usage();
589
590             }
591
592             if (u > 1000000000) {
593
594                 print_usage();
595
596             }
597
598             if (s > 1000000000) {
599
600                 print_usage();
601
602             }
603
604             if (t > 1000000000) {
605
606                 print_usage();
607
608             }
609
610             if (u > 1000000000) {
611
612                 print_usage();
613
614             }
615
616             if (s > 1000000000) {
617
618                 print_usage();
619
620             }
621
622             if (t > 1000000000) {
623
624                 print_usage();
625
626             }
627
628             if (u > 1000000000) {
629
630                 print_usage();
631
632             }
633
634             if (s > 1000000000) {
635
636                 print_usage();
637
638             }
639
640             if (t > 1000000000) {
641
642                 print_usage();
643
644             }
645
646             if (u > 1000000000) {
647
648                 print_usage();
649
650             }
651
652             if (s > 1000000000) {
653
654                 print_usage();
655
656             }
657
658             if (t > 1000000000) {
659
660                 print_usage();
661
662             }
663
664             if (u > 1000000000) {
665
666                 print_usage();
667
668             }
669
670             if (s > 1000000000) {
671
672                 print_usage();
673
674             }
675
676             if (t > 1000000000) {
677
678                 print_usage();
679
680             }
681
682             if (u > 1000000000) {
683
684                 print_usage();
685
686             }
687
688             if (s > 1000000000) {
689
690                 print_usage();
691
692             }
693
694             if (t > 1000000000) {
695
696                 print_usage();
697
698             }
699
700             if (u > 1000000000) {
701
702                 print_usage();
703
704             }
705
706             if (s > 1000000000) {
707
708                 print_usage();
709
710             }
711
712             if (t > 1000000000) {
713
714                 print_usage();
715
716             }
717
718             if (u > 1000000000) {
719
720                 print_usage();
721
722             }
723
724             if (s > 1000000000) {
725
726                 print_usage();
727
728             }
729
730             if (t > 1000000000) {
731
732                 print_usage();
733
734             }
735
736             if (u > 1000000000) {
737
738                 print_usage();
739
740             }
741
742             if (s > 1000000000) {
743
744                 print_usage();
745
746             }
747
748             if (t > 1000000000) {
749
750                 print_usage();
751
752             }
753
754             if (u > 1000000000) {
755
756                 print_usage();
757
758             }
759
760             if (s > 1000000000) {
761
762                 print_usage();
763
764             }
765
766             if (t > 1000000000) {
767
768                 print_usage();
769
770             }
771
772             if (u > 1000000000) {
773
774                 print_usage();
775
776             }
777
778             if (s > 1000000000) {
779
780                 print_usage();
781
782             }
783
784             if (t > 1000000000) {
785
786                 print_usage();
787
788             }
789
790             if (u > 1000000000) {
791
792                 print_usage();
793
794             }
795
796             if (s > 1000000000) {
797
798                 print_usage();
799
800             }
801
802             if (t > 1000000000) {
803
804                 print_usage();
805
806             }
807
808             if (u > 1000000000) {
809
810                 print_usage();
811
812             }
813
814             if (s > 1000000000) {
815
816                 print_usage();
817
818             }
819
820             if (t > 1000000000) {
821
822                 print_usage();
823
824             }
825
826             if (u > 1000000000) {
827
828                 print_usage();
829
830             }
831
832             if (s > 1000000000) {
833
834                 print_usage();
835
836             }
837
838             if (t > 1000000000) {
839
840                 print_usage();
841
842             }
843
844             if (u > 1000000000) {
845
846                 print_usage();
847
848             }
849
850             if (s > 1000000000) {
851
852                 print_usage();
853
854             }
855
856             if (t > 1000000000) {
857
858                 print_usage();
859
860             }
861
862             if (u > 1000000000) {
863
864                 print_usage();
865
866             }
867
868             if (s > 1000000000) {
869
870                 print_usage();
871
872             }
873
874             if (t > 1000000000) {
875
876                 print_usage();
877
878             }
879
880             if (u > 1000000000) {
881
882                 print_usage();
883
884             }
885
886             if (s > 1000000000) {
887
888                 print_usage();
889
890             }
891
892             if (t > 1000000000) {
893
894                 print_usage();
895
896             }
897
898             if (u > 1000000000) {
899
900                 print_usage();
901
902             }
903
904             if (s > 1000000000) {
905
906                 print_usage();
907
908             }
909
910             if (t > 1000000000) {
911
912                 print_usage();
913
914             }
915
916             if (u > 1000000000) {
917
918                 print_usage();
919
920             }
921
922             if (s > 1000000000) {
923
924                 print_usage();
925
926             }
927
928             if (t > 1000000000) {
929
930                 print_usage();
931
932             }
933
934             if (u > 1000000000) {
935
936                 print_usage();
937
938             }
939
940             if (s > 1000000000) {
941
942                 print_usage();
943
944             }
945
946             if (t > 1000000000) {
947
948                 print_usage();
949
950             }
951
952             if (u > 1000000000) {
953
954                 print_usage();
955
956             }
957
958             if (s > 1000000000) {
959
960                 print_usage();
961
962             }
963
964             if (t > 1000000000) {
965
966                 print_usage();
967
968             }
969
970             if (u > 1000000000) {
971
972                 print_usage();
973
974             }
975
976             if (s > 1000000000) {
977
978                 print_usage();
979
980             }
981
982             if (t > 1000000000) {
983
984                 print_usage();
985
986             }
987
988             if (u > 1000000000) {
989
990                 print_usage();
991
992             }
993
994             if (s > 1000000000) {
995
996                 print_usage();
997
998             }
999
1000            if (t > 1000000000) {
1001                print_usage();
1002
1003            }
1004
1005            if (u > 1000000000) {
1006
1007                print_usage();
1008
1009            }
1010
1011            if (s > 1000000000) {
1012
1013                print_usage();
1014
1015            }
1016
1017            if (t > 1000000000) {
1018
1019                print_usage();
1020
1021            }
1022
1023            if (u > 1000000000) {
1024
1025                print_usage();
1026
1027            }
1028
1029            if (s > 1000000000) {
1030
1031                print_usage();
1032
1033            }
1034
1035            if (t > 1000000000) {
1036
1037                print_usage();
1038
1039            }
1040
1041            if (u > 1000000000) {
1042
1043                print_usage();
1044
1045            }
1046
1047            if (s > 1000000000) {
1048
1049                print_usage();
1050
1051            }
1052
1053            if (t > 1000000000) {
1054
1055                print_usage();
1056
1057            }
1058
1059            if (u > 1000000000) {
1060
1061                print_usage();
1062
1063            }
1064
1065            if (s > 1000000000) {
1066
1067                print_usage();
1068
1069            }
1070
1071            if (t > 1000000000) {
1072
1073                print_usage();
1074
1075            }
1076
1077            if (u > 1000000000) {
1078
1079                print_usage();
1080
1081            }
1082
1083            if (s > 1000000000) {
1084
1085                print_usage();
1086
1087            }
1088
1089            if (t > 1000000000) {
1090
1091                print_usage();
1092
1093            }
1094
1095            if (u > 1000000000) {
1096
1097                print_usage();
1098
1099            }
1100
1101            if (s > 1000000000) {
1102
1103                print_usage();
1104
1105            }
1106
1107            if (t > 1000000000) {
1108
1109                print_usage();
1110
1111            }
1112
1113            if (u > 1000000000) {
1114
1115                print_usage();
1116
1117            }
1118
1119            if (s > 1000000000) {
1120
1121                print_usage();
1122
1123            }
1124
1125            if (t > 1000000000) {
1126
1127                print_usage();
1128
1129            }
1130
1131            if (u > 1000000000) {
1132
1133                print_usage();
1134
1135            }
1136
1137            if (s > 1000000000) {
1138
1139                print_usage();
1140
1141            }
1142
1143            if (t > 1000000000) {
1144
1145                print_usage();
1146
1147            }
1148
1149            if (u > 1000000000) {
1150
1151                print_usage();
1152
1153            }
1154
1155            if (s > 1000000000) {
1156
1157                print_usage();
1158
1159            }
1160
1161            if (t > 1000000000) {
1162
1163                print_usage();
1164
1165            }
1166
1167            if (u > 1000000000) {
1168
1169                print_usage();
1170
1171            }
1172
1173            if (s > 1000000000) {
1174
1175                print_usage();
1176
1177            }
1178
1179            if (t > 1000000000) {
1180
1181                print_usage();
1182
1183            }
1184
1185            if (u > 1000000000) {
1186
1187                print_usage();
1188
1189            }
1190
1191            if (s > 1000000000) {
1192
1193                print_usage();
1194
1195            }
1196
1197            if (t > 1000000000) {
1198
1199                print_usage();
1200
1201            }
1202
1203            if (u > 1000000000) {
1204
1205                print_usage();
1206
1207            }
1208
1209            if (s > 1000000000) {
1210
1211                print_usage();
1212
1213            }
1214
1215            if (t > 1000000000) {
1216
1217                print_usage();
1218
1219            }
1220
1221            if (u > 1000000000) {
1222
1223                print_usage();
1224
1225            }
1226
1227            if (s > 1000000000) {
1228
1229                print_usage();
1230
1231            }
1232
1233            if (t > 1000000000) {
1234
1235                print_usage();
1236
1237            }
1238
1239            if (u > 1000000000) {
1240
1241                print_usage();
1242
1243            }
1244
1245            if (s > 1000000000) {
1246
1247                print_usage();
1248
1249            }
1250
1251            if (t > 1000000000) {
1252
1253                print_usage();
1254
1255            }
1256
1257            if (u > 1000000000) {
1258
1259                print_usage();
1260
1261            }
1262
1263            if (s > 1000000000) {
1264
1265                print_usage();
1266
1267            }
1268
1269            if (t > 1000000000) {
1270
1271                print_usage();
1272
1273            }
1274
1275            if (u > 1000000000) {
1276
1277                print_usage();
1278
1279            }
1280
1281            if (s > 1000000000) {
1282
1283                print_usage();
1284
1285            }
1286
1287            if (t > 1000000000) {
1288
1289                print_usage();
1290
1291            }
1292
1293            if (u > 1000000000) {
1294
1295                print_usage();
1296
1297            }
1298
1299            if (s > 1000000000) {
1300
1301                print_usage();
1302
1303            }
1304
1305            if (t > 1000000000) {
1306
1307                print_usage();
1308
1309            }
1310
1311            if (u > 1000000000) {
1312
1313                print_usage();
1314
1315            }
1316
1317            if (s > 1000000000) {
1318
1319                print_usage();
1320
1321            }
1322
1323            if (t > 1000000000) {
1324
1325                print_usage();
1326
1327            }
1328
1329            if (u > 1000000000) {
1330
1331                print_usage();
1332
1333            }
1334
1335            if (s > 1000000000) {
1336
1337                print_usage();
1338
1339            }
1340
1341            if (t > 1000000000) {
1342
1343                print_usage();
1344
1345            }
1346
1347            if (u > 1000000000) {
1348
1349                print_usage();
1350
1351            }
1352
1353            if (s > 1000000000) {
1354
1355                print_usage();
1356
1357            }
1358
1359            if (t > 1000000000) {
1360
1361                print_usage();
1362
1363            }
1364
1365            if (u > 1000000000) {
1366
1367                print_usage();
1368
1369            }
1370
1371            if (s > 1000000000) {
1372
1373                print_usage();
1374
1375            }
1376
1377            if (t > 1000000000) {
1378
1379                print_usage();
1380
1381            }
1382
1383            if (u > 1000000000) {
1384
1385                print_usage();
1386
1387            }
1388
1389            if (s > 1000000000) {
1390
1391                print_usage();
1392
1393            }
1394
1395            if (t > 1000000000) {
1396
1397                print_usage();
1398
1399            }
1400
1401            if (u > 1000000000) {
1402
1403                print_usage();
1404
1405            }
1406
1407            if (s > 1000000000) {
1408
1409                print_usage();
1410
1411            }
1412
1413            if (t > 1000000000) {
1414
1415                print_usage();
1416
1417            }
1418
1419            if (u > 1000000000) {
1420
1421                print_usage();
1422
1423            }
1424
1425            if (s > 1000000000) {
1426
1427                print_usage();
1428
1429            }
1430
1431            if (t > 1000000000) {
1432
1433                print_usage();
1434
1435            }
1436
1437            if (u > 1000000000) {
1438
1439                print_usage();
1440
1441            }
1442
1443            if (s > 1000000000) {
1444
1445                print_usage();
1446
1447            }
1448
1449            if (t > 1000000000) {
1450
1451                print_usage();
1452
1453            }
1454
1455            if (u > 1000000000) {
1456
1457                print_usage();
1458
1459            }
1460
1461            if (s > 1000000000) {
1462
1463                print_usage();
1464
1465            }
1466
1467            if (t > 1000000000) {
1468
1469                print_usage();
1470
1471            }
1472
1473            if (u > 1000000000) {
1474
1475                print_usage();
1476
1477            }
1478
1479            if (s > 1000000000) {
1480
1481                print_usage();
1482
1483            }
1484
1485            if (t > 1000000000) {
1486
1487                print_usage();
1488
1489            }
1490
1491            if (u > 1000000000) {
1492
1493                print_usage();
1494
1495            }
1496
1497            if (s > 1000000000) {
1498
1499                print_usage();
1500
1501            }
1502
1503            if (t > 1000000000) {
1504
1505                print_usage();
1506
1507            }
1508
1509            if (u > 1000000000) {
1510
1511                print_usage();
1512
1513            }
1514
1515            if (s > 1000000000) {
1516
1517                print_usage();
1518
1519            }
1520
1521            if (t > 1000000000) {
1522
1523                print_usage();
1524
1525            }
1526
1527            if (u > 1000000000) {
1528
1529                print_usage();
1530
1531            }
1532
1533            if (s > 1000000000) {
1534
1535                print_usage();
1536
1537            }
1538
1539            if (t > 1000000000) {
1540
1541                print_usage();
1542
1543            }
1544
1545            if (u > 1000000000) {
1546
1547                print_usage();
1548
1549            }
1550
1551            if (s > 1000000000) {
1552
1553                print_usage();
1554
1555            }
1556
1557            if (t > 1000000000) {
1558
1559                print_usage();
1560
1561            }
1562
1563            if (u > 1000000000) {
1564
1565                print_usage();
1566
1567            }
1568
1569            if (s > 1000000000) {
1570
1571                print_usage();
1572
1573            }
1574
1575            if (t > 1000000000) {
1576
1577                print_usage();
1578
1579            }
1580
1581            if (u > 1000000000) {
1582
1583                print_usage();
1584
1585            }
1586
1587            if (s > 1000000000) {
1588
1589                print_usage();
1590
1591            }
1592
1593            if (t > 1000000000) {
1594
1595                print_usage();
1596
1597            }
1598
1599            if (u > 1000000000) {
1600
1601                print_usage();
1602
1603            }
1604
1605            if (s > 1000000000) {
1606
1607                print_usage();
1608
1609            }
1610
1611            if (t > 1000000000) {
1612
1613                print_usage();
1614
1615            }
1616
1617            if (u > 1000000000) {
1618
1619                print_usage();
1620
1621            }
1622
1623            if (s > 1000000000) {
1624
1625                print_usage();
1626
1627            }
1628
1629            if (t > 1000000000) {
1630
1631                print_usage();
1632
1633            }
1634
1635            if (u > 1000000000) {
16
```



```
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 :
250         0;
251     return (double)ones / (double)((long long)N * (long long)N);
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
257 , int verify, int print) {
258     const int PRINT_LIMIT = 16;
259
260     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"
267                   , N);
268         }
269
270         memcpy(A, Ain, (size_t)N * (size_t)N);
271
272         double t0 = seconds_now();
273         warshall_logical_omp(A, N);
274         double t1 = seconds_now();
275         double kernel_time = t1 - t0;
276
277         uint8_t* Rref = alloc_matrix(N);
278         bfs_closure_ref(Ain, Rref, N);
279
280         int ok = verify_against_ref(Rref, A, N);
281
282         if (print) {
283             print_matrix(Ain, N, "MATRIZ_DE_ENTRADA_(Grafo/_Adyacencia)");
284             print_matrix(Rref, N, "MATRIZ_DE_VERIFICACION_(Referencia_BFS)");
285             print_matrix(A, N, "MATRIZ_DE_SALIDA_(Warshall_logico)[OPENMP]");
286         }
287
288         printf("\nVALIDACION_(BFS)_para_N=%d:%s\n", N, ok ? "OK" : "FALLIDA");
289         printf("Tiempo del nucleo_(warshall_logical_omp):%.6f s\n", kernel_time);
290         printf("Resumen_params_|_N=%d_|_p=% .3f_|_seed=%u_|_repeats=%d_|_verify=%d_|_
291             _print=%d\n",
292             N, p, seed, repeats, verify, print);
293
294         free(Rref);
295         free(A);
296
297         return ok ? EXIT_SUCCESS : EXIT_FAILURE;
298     }
299 }
```



```
296
297     double best = 1e100;
298
299     for (int r = 0; r < repeats; r++) {
300
301         memcpy(A, Ain, (size_t)N * (size_t)N);
302
303         double t0 = seconds_now();
304
305         warshall_logical_omp(A, N);
306
307         double t1 = seconds_now();
308
309         double dt = t1 - t0;
310
311         if (dt < best) best = dt;
312     }
313
314 // =====
315 // Menu
316 // =====
317 static void menu_loop(void) {
318     for (;;) {
319
320         printf("\n=====\\n");
321         printf("___MENU___Warshall_logico_OPENMP\\n");
322         printf("=====\\n");
323         printf("1) Ingresar_MATRIZ_manual+parametros\\n");
324         printf("2) Ingresar_parametros+_GRAFO_random\\n");
325         printf("3) Grafo_y_parametros_RANDOM\\n");
326         printf("0) Salir\\n");
327
328         int opt = read_int_prompt("Opcion:", 0, 3);
329
330         if (opt == 0) break;
331
332         int N = 256;
333         double p = 0.05;
334         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         ;
370         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
428             EXIT_FAILURE; }
429         if (p < 0.0 || p > 1.0) { fprintf(stderr, "ERROR: p debe estar en [0,1]\n")
430             ; return EXIT_FAILURE; }
431         if (repeats <= 0) repeats = 1;
432
433         const int PRINT_LIMIT = 16;
434         if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
435
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,
46                     cudaGetErrorString(e));
47         std::exit(EXIT_FAILURE);
48    }
49 }
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 }
```



```
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",
78                     bytes, N);
79         std::exit(EXIT_FAILURE);
80     }
81     return A;
82 }
83
84 static void init_random(uint8_t* A, int N, double p, unsigned seed) {
85     std::srand(seed);
86     for (int i = 0; i < N * N; i++) {
87         double r = (double)std::rand() / (double)RAND_MAX;
88         A[i] = (r < p) ? 1 : 0;
89     }
90 }
91
92 static void print_matrix(const uint8_t* A, int N, const char* title) {
93     std::printf("\n==== %s (N=%d) ====\n", title, N);
94
95     std::printf("----");
96     for (int j = 0; j < N; j++) std::printf(" %2d ", j);
97     std::printf("\n");
98
99     std::printf("----");
100    for (int j = 0; j < N; j++) std::printf(" ---");
101    std::printf("\n");
```



```
102     for (int i = 0; i < N; i++) {
103         std::printf("%2d|", i);
104         const uint8_t* row = &A[i * N];
105         for (int j = 0; j < N; j++) std::printf("%2d|", (int)row[j]);
106         std::printf("\n");
107     }
108 }
109
110 // =====
111 // CUDA kernel: 1 paso k (in-place)
112 // =====
113 __global__ void warshall_step_u8(uint8_t* A, int N, int k) {
114     int j = blockIdx.x * blockDim.x + threadIdx.x; // col
115     int i = blockIdx.y * blockDim.y + threadIdx.y; // row
116     if (i < N && j < N) {
117         uint8_t aik = A[i * N + k];
118         if (!aik) return; // pequeno atajo
119         uint8_t akj = A[k * N + j];
120         uint8_t aij = A[i * N + j];
121         A[i * N + j] = (uint8_t)(aij | (aik & akj));
122     }
123 }
124
125 // =====
126 // Nucleo: Warshall logico (CUDA)
127 // =====
128 static void warshall_logical_cuda(uint8_t* A_host, int N) {
129     size_t bytes = (size_t)N * (size_t)N * sizeof(uint8_t);
130
131     uint8_t* d_A = nullptr;
132     CUDA_CALL(cudaMalloc(&d_A, bytes));
133     CUDA_CALL(cudaMemcpy(d_A, A_host, bytes, cudaMemcpyHostToDevice));
134
135     dim3 block(16, 16);
136     dim3 grid((N + block.x - 1) / block.x,
137               (N + block.y - 1) / block.y);
138
139     for (int k = 0; k < N; k++) {
```



```
140         warshall_step_u8<<<grid, block>>>(d_A, N, k);
141
142         CUDA_CALL(cudaGetLastError());
143
144         CUDA_CALL(cudaDeviceSynchronize()); // claridad/didactica
145
146     CUDA_CALL(cudaMemcpy(A_host, d_A, bytes, cudaMemcpyDeviceToHost));
147     CUDA_CALL(cudaFree(d_A));
148 }
149
150 // -----
151 // Referencia: cerradura transitiva con BFS (CPU)
152 // -----
153 static void bfs_closure_ref(const uint8_t* Ain, uint8_t* Rref, int N) {
154     int* queue = (int*)std::malloc((size_t)N * sizeof(int));
155     uint8_t* vis = (uint8_t*)std::malloc((size_t)N * sizeof(uint8_t));
156
157     if (!queue || !vis) {
158         std::fprintf(stderr, "ERROR: memoria insuficiente para BFS\n");
159         std::free(queue);
160         std::free(vis);
161         std::exit(EXIT_FAILURE);
162     }
163
164     for (int s = 0; s < N; s++) {
165         std::memset(vis, 0, (size_t)N);
166         int front = 0, back = 0;
167
168         const uint8_t* row_s = &Ain[s * N];
169         for (int v = 0; v < N; v++) {
170             if (row_s[v]) {
171                 vis[v] = 1;
172                 queue[back++] = v;
173             }
174         }
175         while (front < back) {
176             int u = queue[front++];
177             const uint8_t* row_u = &Ain[u * N];
178             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,_"
201                             "obtenido=%d\n",
202                             i, j, (int)rr[j], (int)ao[j]);
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     }
254 }
```



```
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 :
283         0;
284     return (double)ones / (double)((long long)N * (long long)N);
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    ;
322    std::printf("Tiempo del nucleo (warshall_logical_cuda): %.6f s\n",
323               kernel_time);
```



```

322     std::printf("Resumen_params_|_N=%d_|_p=%.3f_|_seed=%u_|_repeats=%d_|_verify
323             =%d_|_print=%d\n",
324             N, p, seed, repeats, verify, print);
325
326     std::free(Rref);
327     std::free(A);
328
329     return ok ? EXIT_SUCCESS : EXIT_FAILURE;
330 }
331
332 double best = 1e100;
333
334 for (int r = 0; r < repeats; r++) {
335     std::memcpy(A, Ain, (size_t)N * (size_t)N);
336
337     double t0 = seconds_now();
338     warshall_logical_cuda(A, N);
339
340     double t1 = seconds_now();
341
342     double dt = t1 - t0;
343
344     if (dt < best) best = dt;
345 }
346
347 std::printf("CUDA_Warshall_logico_|_N=%d_|_p=%.3f_|_seed=%u_|_repeats=%d_|_
348             best_kernel_time=%.6f_s\n",
349             N, p, seed, repeats, best);
350
351     std::free(A);
352
353     return EXIT_SUCCESS;
354 }
355
356 // =====
357 // Menu
358 // =====
359
360 static void menu_loop(void) {
361     for (;;) {
362         std::printf("\n=====MENU - Warshall_logico_CUDA\n");
363         std::printf("=====MENU - Warshall_logico_CUDA\n");
364         std::printf("=====\\n");
365         std::printf("1)_Ingresar_MATRIZ_manual+_parametros\\n");
366         std::printf("2)_Ingresar_parametros+_GRAFO_random\\n");
367         std::printf("3)_Grafo_y_parametros_RANDOM\\n");
368     }
369 }

```



```
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,
377                     N);
378         std::printf("Formato_permitido_por_fila:_'0_1_0_1'_o_'0101...'\\n\\n");
379
380         char line[8192];
381         for (int i = 0; i < N; i++) {
382             for (;;) {
383                 std::printf("Fila_%d:", i);
384                 read_line(line, sizeof(line));
385                 if (parse_row_01(line, &Ain[i * N], N)) break;
386                 std::printf("Fila_invalida._Debe_contener_%d_valores_0/1.\n",
387                             N);
388             }
389         }
390         p = density_ones(Ain, N);
391         seed = 0;
392
393         repeats = read_int_prompt("repeats_(>=1):", 1, 1000);
394         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
399 else if (opt == 2) {
400
401     N = read_int_prompt("N_(1..4096):", 1, 4096);
402     p = read_double_prompt("p_(0..1):", 0.0, 1.0);
403     seed = (unsigned)read_long_prompt("seed_(0..2^31-1):", 0, 2147483647L)
404
405     ;
406
407     repeats = read_int_prompt("repeats_(>=1):", 1, 1000);
408     verify   = read_yesno_prompt("verify");
409     print    = read_yesno_prompt("print");
410
411
412     Ain = alloc_matrix(N);
413     init_random(Ain, N, p, seed);
414 }
415
416
417 else if (opt == 3) {
418
419     unsigned s = (unsigned)time(NULL);
420     srand(s);
421
422     int choices[] = {8,16,32,64,128,256,512,1024};
423     int idx = rand() % (int)(sizeof(choices)/sizeof(choices[0]));
424     N = choices[idx];
425
426
427     p = 0.01 + ((double)rand() / (double)RAND_MAX) * 0.19; // [0.01..0.20]
428     seed = (unsigned)rand();
429     repeats = 1 + (rand() % 7);
430
431
432     verify = (N <= 128) ? 1 : 0;
433     print  = (N <= 16) ? 1 : 0;
434
435
436     Ain = alloc_matrix(N);
437     init_random(Ain, N, p, seed);
438
439
440     std::printf("\nParametros_random_generados:\n");
441     std::printf("N=%d | p=%.3f | seed=%u | repeats=%d | verify=%d | print=%d\n",
442
443             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             .c_str());
438     }
439
440     if (!read_yesno_prompt("\nDeseas ejecutar otra vez")) break;
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
461             EXIT_FAILURE; }
462         if (p < 0.0 || p > 1.0) { std::fprintf(stderr, "ERROR: p debe estar en "
463             "[0,1]\n"); return EXIT_FAILURE; }
464         if (repeats <= 0) repeats = 1;
465
466         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
473     return rc;
474
475     // --- Modo menu ---
476     menu_loop();
477
478     return 0;
}
```