

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

Resumen	4
1. Introducción	5
2. Objetivos	5
2.1. Objetivo general	5
2.2. Objetivos específicos	5
3. Marco teórico (solo OpenMP y CUDA)	5
3.1. OpenMP: paralelismo en memoria compartida	5
3.2. CUDA: paralelismo masivo en GPU	6
4. Implementación paralela (Fase 2)	6
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
5. Evaluación y métricas	7
5.1. Diseño experimental	7
5.2. Tablas de tiempos (llenado manual)	8
5.3. Métricas requeridas	8
5.4. Análisis del rendimiento (CUDA)	9
6. Gráficos (plantillas)	9
7. Discusión: ¿por qué CUDA suele ser más rápido que OpenMP?	9
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 CUDA (memoria compartida)	10
8. Conclusiones	10
9. Trabajo futuro	10
A. Código OpenMP (warshall_menu_omp.c)	10





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												
2048												
3072												
4096												

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												
2048												
3072												
4096												

Métricas requeridas

Sea T_{seq} el tiempo secuencial (de la Fase 1) y T_{par} el tiempo paralelo (OpenMP o CUDA), para un mismo N .

Aceleración (Speedup)..

$$S = \frac{T_{seq}}{T_{par}}$$

Eficiencia (Efficiency) para OpenMP..

$$E = \frac{S}{\#\text{núcleos (o hilos)}}$$



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 CUDA (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 // + MENU 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
72 static void print_matrix(const uint8_t* A, int N, const char* title) {
73     printf("\n==== %s (N=%d) ====\n", title, N);
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
86     for (int j = 0; j < N; j++) printf("%2d ", (int)row[j]);
87
88 }
89
90 // -----
91 // Nucleo: Warshall logico (OPENMP)
92 // -----
93 void warshall_logical_omp(uint8_t* A, int N) {
94
95     // A[i][j] = A[i][j] OR (A[i][k] AND A[k][j])
96     // k debe ser secuencial, pero paralelizamos i (cada hilo escribe su fila).
97
98     for (int k = 0; k < N; k++) {
99
100         const uint8_t* row_k = &A[k * N];
101
102
103         #pragma omp parallel for schedule(static)
104
105         for (int i = 0; i < N; i++) {
106
107             uint8_t aik = A[i * N + k];
108
109             if (!aik) continue;
110
111             uint8_t* row_i = &A[i * N];
112
113             // vectorizable
114
115             for (int j = 0; j < N; j++) {
116
117                 row_i[j] = (uint8_t)(row_i[j] | (aik & row_k[j]));
118
119             }
120
121         }
122
123     }
124
125
126     // -----
127
128     // Referencia: cerradura transitiva con BFS
129
130
131     static void bfs_closure_ref(const uint8_t* Ain, uint8_t* Rref, int N) {
132
133         int* queue = (int*)malloc((size_t)N * sizeof(int));
134
135         uint8_t* vis = (uint8_t*)malloc((size_t)N * sizeof(uint8_t));
136
137         if (!queue || !vis) {
138
139             fprintf(stderr, "ERROR: memoria insuficiente para BFS\n");
140
141             free(queue);
142
143         }
144
145     }
146
147 }
```



```
122         free(vis);
123
124         exit(EXIT_FAILURE);
125     }
126
127     for (int s = 0; s < N; s++) {
128
129         memset(vis, 0, (size_t)N);
130
131         int front = 0, back = 0;
132
133
134         const uint8_t* row_s = &Ain[s * N];
135
136         for (int v = 0; v < N; v++) {
137
138             if (row_s[v]) {
139
140                 vis[v] = 1;
141
142                 queue[back++] = v;
143
144             }
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     }
154
155     free(queue);
156
157     free(vis);
158
159 static int verify_against_ref(const uint8_t* Rref, const uint8_t* Aout, int N) {
160
161     for (int i = 0; i < N; i++) {
162
163         const uint8_t* rr = &Rref[i * N];
```



```
160     const uint8_t* ao = &Aout[i * N];
161
162     for (int j = 0; j < N; j++) {
163
164         if (rr[j] != ao[j]) {
165
166             fprintf(stderr,
167                     "FALLO_verificacion:_fila_i=%d,_col_j=%d_|_esperado=%d,_
168                     obtenido=%d\n",
169                     i, j, (int)rr[j], (int)ao[j]);
170
171         }
172
173     }
174
175     return 1;
176 }
177
178 // =====
179 // Helpers de input robusto (sin scanf)
180 // =====
181
182 static void read_line(char* buf, size_t n) {
183
184     if (!fgets(buf, (int)n, stdin)) {
185
186         printf("\nEOF_detectado._Saliendo.\n");
187
188         exit(0);
189     }
190 }
191
192 static long read_long_prompt(const char* prompt, long minv, long maxv) {
193
194     char line[256];
195
196     for (;;) {
197
198         printf("%s", prompt);
199
200         read_line(line, sizeof(line));
201
202
203         char* end = NULL;
204
205         errno = 0;
206
207         long v = strtol(line, &end, 10);
208
209         if (errno == 0) {
210
211             while (end && *end && isspace((unsigned char)*end)) end++;
212
213             if (end && (*end == '\0' || *end == '\n')) {
214
215                 if (v >= minv && v <= maxv) return v;
216
217             }
218
219         }
220
221     }
222 }
```



```
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=sí, 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; }
261     if (repeats <= 0) repeats = 1;
262
263     uint8_t* A = alloc_matrix(N);
264
265     if (verify) {
266         if (N > 128) {
267             printf("Aviso: verificacion activada con N=%d; se recomienda N<=128.\n"
268                 , N);
269         }
270     }
271     memcpy(A, Ain, (size_t)N * (size_t)N);
```





```
307     printf("OPENMP_Warshall_logico_|_N=%d_|_p=%.3f_|_seed=%u_|_repeats=%d_|_
308             best_kernel_time=%.6f_|_s_|_threads=%d\n",
309             N, p, seed, repeats, best, omp_get_max_threads());
310
311     free(A);
312
313     return EXIT_SUCCESS;
314 }
315
316 // =====
317 // Menu
318 // =====
319
320 static void menu_loop(void) {
321
322     for (;;) {
323
324         printf("\n=====|\n");
325         printf("||MENU||-||Warshall||logico||OPENMP||\n");
326         printf("=====|\n");
327         printf("1) Ingresar||MATRIZ||manual||+||parametros||\n");
328         printf("2) Ingresar||parametros||+||GRAFO||random||\n");
329         printf("3) Grafo||y||parametros||RANDOM||\n");
330         printf("0) Salir\n");
331
332
333         int opt = read_int_prompt("Opcion:", 0, 3);
334
335         if (opt == 0) break;
336
337
338         int N = 256;
339         double p = 0.05;
340         unsigned seed = 1234;
341
342         int repeats = 3;
343         int verify = 0;
344
345         int print = 0;
346
347
348         uint8_t* Ain = NULL;
349
350
351         if (opt == 1) {
352
353             N = read_int_prompt("Ingrese||N|| (1..2048||recomendado) : ", 1, 4096);
354
355             Ain = alloc_matrix(N);
356
357
358             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);
371     verify = read_yesno_prompt("verify");
372     print = read_yesno_prompt("print");
373
374     Ain = alloc_matrix(N);
375     init_random(Ain, N, p, seed);
376 }
377 else if (opt == 3) {
378     unsigned s = (unsigned)time(NULL);
379     srand(s);
380
381     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
407     if (!read_yesno_prompt("\nDeseas_ejecutar_otra_vez")) break;
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
434     const int PRINT_LIMIT = 16;
435
436     if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
437
438     uint8_t* Ain = alloc_matrix(N);
439     init_random(Ain, N, p, seed);
440
441     int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
442     free(Ain);
443
444     return rc;
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
24 // paralelo.
25 //
26 // IMPORTANTE (correctitud):
27 // - Este kernel lee A[i,k] y A[k,j] del MISMO buffer A que tambien se escribe.
28 // Eso replica tu version CUDA didactica anterior, y suele funcionar para
29 // Marshall booleano,
30 // pero si quieres "paso k limpio" (lectura desde snapshot), usa doble buffer (
31 // Ain->Aout)
32 // por cada k (mas lento por copias). Aqui dejo la version simple y rapida (in-
33 // place).
34
35 #define _POSIX_C_SOURCE 200809L
36
37 #include <cuda_runtime.h>
38 #include <cstdio>
39 #include <cstdlib>
40 #include <cstdint>
41 #include <cstring>
42 #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 #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     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
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: Marshall 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         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
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
198         for (int j = 0; j < N; j++) {
199             if (rr[j] != ao[j]) {
200                 std::fprintf(stderr,
201                             "FALLO_verificacion:_fila_i=%d,_col_j=%d_|_esperado=%d,_"
202                             "obtenido=%d\n",
203                             i, j, (int)rr[j], (int)ao[j]);
204             }
205         }
206     }
207     return 1;
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
228     long v = std::strtol(line, &end, 10);
229
230     if (errno == 0) {
231
232         while (end && *end && std::isspace((unsigned char)*end)) end++;
233
234         if (end && (*end == '\0' || *end == '\n')) {
235
236             if (v >= minv && v <= maxv) return v;
237
238         }
239
240         std::printf("Entrada invalida. Rango permitido: [%ld..%ld]\n", minv, maxv);
241     }
242
243 }
244
245 static double read_double_prompt(const char* prompt, double minv, double maxv) {
246
247     char line[256];
248
249     for (;;) {
250
251         std::printf("%s", prompt);
252
253         read_line(line, sizeof(line));
254
255         char* end = NULL;
256
257         errno = 0;
258
259         double v = std::strtod(line, &end);
260
261         if (errno == 0) {
262
263             while (end && *end && std::isspace((unsigned char)*end)) end++;
264
265             if (end && (*end == '\0' || *end == '\n')) {
266
267                 if (v >= minv && v <= maxv) return v;
268
269             }
270
271         }
272
273         std::printf("Entrada invalida. Rango permitido: [.3f..%.3f]\n", minv, maxv
274
275     );
276
277 }
278
279 static int read_int_prompt(const char* prompt, int minv, int maxv) {
280
281     return (int)read_long_prompt(prompt, (long)minv, (long)maxv);
282
283 }
284
285 static int read_yesno_prompt(const char* prompt) {
286
287     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,
290     , int verify, int print) {
291     const int PRINT_LIMIT = 16;
292
293     if (N <= PRINT_LIMIT) { verify = 1; print = 1; }
294     if (repeats <= 0) repeats = 1;
295
296     uint8_t* A = alloc_matrix(N);
297
298     if (verify) {
299         if (N > 128) {
```



```
299         std::printf("Aviso: verificacion_activada con N=%d; se recomienda N\n"
300                     <=128.\n", N);
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_VALIDACION_(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);
324     std::printf("Resumen_params_|_N=%d_|_p=%.3f_|_seed=%u_|_repeats=%d_|_verify\n"
325                 "=%d_|_print=%d\n",
326                 N, p, seed, repeats, verify, print);
327
328     std::free(Rref);
329     std::free(A);
330
331     return ok ? EXIT_SUCCESS : EXIT_FAILURE;
332 }
```



```
333     double t0 = seconds_now();
334
335     warshall_logical_cuda(A, N);
336
337     double t1 = seconds_now();
338
339     double dt = t1 - t0;
340
341     if (dt < best) best = dt;
342
343     }
344
345     std::printf("CUDA_Warshall_logico_|_N=%d_|_p=%.3f_|_seed=%u_|_repeats=%d_|_
346
347     best_kernel_time=%.6f_s\n",
348
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
363         std::printf("\n=====\\n");
364
365         std::printf("___MENU___Warshall_logico_CUDA\\n");
366
367         std::printf("=====\\n");
368
369         std::printf("1) Ingresar_MATRIZ_manual+_parametros\\n");
370
371         std::printf("2) Ingresar_parametros+_GRAFO_random\\n");
372
373         std::printf("3) Grafo_y_parametros_RANDOM\\n");
374
375         std::printf("0) Salir\\n");
376
377
378         int opt = read_int_prompt("Opcion:", 0, 3);
379
380         if (opt == 0) break;
381
382
383         int N = 256;
384
385         double p = 0.05;
386
387         unsigned seed = 1234;
388
389         int repeats = 3;
390
391         int verify = 0;
392
393         int print = 0;
```



```
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                             );
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         ;
403         repeats = read_int_prompt("repeats_(>=1):", 1, 1000);
404         verify = read_yesno_prompt("verify");
405         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
437             ).\n");
438     }
439
440     if (!read_yesno_prompt("\nDeseas_ejecutar_otra_vez")) break;
441 }
```



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



```
477     return 0;  
478 }
```