

# Aceleración con OpenMP de un Screensaver Nebular en SDL2

Informe técnico de optimización paralela

Diederich Solís

Andy Fuentes

Davis Roldan

Universidad del Valle de Guatemala

6 de septiembre de 2025

## Resumen

Este informe documenta el diseño, implementación y evaluación de un renderizador tipo *screensaver* basado en SDL2 que genera un campo nebuloso en tiempo real y lo acelera con OpenMP. Se incorporó validación defensiva de argumentos (CLI), un modo de render a baja resolución con reescalado (`-render-scale`) para mejorar los FPS, y un *HUD* superpuesto que muestra en pantalla el contador de FPS, número de hilos, octavas y escala efectiva. Se exploran distintas políticas de planificación (`static`, `dynamic`, `guided`, `auto`) y tamaños de *chunk* para balancear carga. Los resultados muestran mejoras de rendimiento sustanciales frente al modo secuencial, alcanzando tasas cercanas o superiores a 30 FPS bajo configuraciones adecuadas, con un speedup de hasta 3.8× utilizando 8 hilos.

## 1. Introducción

La generación procedural de efectos visuales complejos como nebulosas requiere un alto poder computacional para mantener tasas de refresco adecuadas en tiempo real. Este proyecto aborda el desafío de paralelizar un renderizador nebuloso utilizando OpenMP, manteniendo la coherencia visual y optimizando el uso de recursos. Se implementaron técnicas de optimización como renderizado a baja resolución con escalado, planificación dinámica de tareas y vectorización, logrando mejoras significativas en el rendimiento.

## 2. Objetivos

- Paralelizar el trazado de píxeles del campo nebuloso con OpenMP, evitando *data races*.
- Exponer parámetros de ejecución por CLI con validación defensiva y `clamp` de rangos.
- Introducir `-render-scale` (0.3–1.0) para reducir cómputo y aumentar FPS mediante *low-res render + upscale*.
- Mostrar un *HUD* en pantalla con FPS actuales y metadatos de ejecución.
- Medir y comparar *speedup* y eficiencia para distintos *schedules* y *chunks*.
- Analizar el impacto de diferentes estrategias de paralelización en la calidad visual y rendimiento.

### 3. Entorno y configuración experimental

#### 3.1. Configuración hardware

- **Dispositivo:** MacBook Pro (2021) con chip Apple M1 Pro
- **CPU:** 8 núcleos de rendimiento (Firestorm) y 2 núcleos de eficiencia (Icestorm)
- **Memoria:** 16 GB RAM unificada
- **GPU:** 16 núcleos integrados

#### 3.2. Configuración software

- **Sistema operativo:** macOS Sonoma 14.4
- **Compilador:** Apple Clang 15.0.0 (con soporte OpenMP)
- **Bibliotecas:** SDL2 2.28.5, OpenMP 5.0
- **Opciones de compilación:** `-O3 -ffast-math -march=native`

#### 3.3. Metodología experimental

- Resolución base:  $1280 \times 720$
- Paleta: `nebula` (configuración predeterminada)
- VSync desactivado para mediciones precisas de FPS
- Duración por corrida: 60 segundos con período de calentamiento de 5 segundos
- Repeticiones: 5 corridas por configuración, reportando promedio y desviación estándar
- Parámetros explorados:
  - Número de hilos (1, 2, 4, 8)
  - Políticas de planificación (static, dynamic, guided, auto)
  - Tamaños de chunk (16, 32, 64, 128)
  - Factores de escala de render (1.0, 0.8, 0.7, 0.6, 0.5)
  - Número de octavas de ruido (4, 6, 8)

#### Comandos reproducibles

```
1 # Dependencias (Homebrew)
2 brew install llvm libomp sdl2
3
4 # Configuración del entorno
5 export CC=/opt/homebrew/opt/llvm/bin/clang
6 export CXX=/opt/homebrew/opt/llvm/bin/clang++
7 export LDFLAGS="-L/opt/homebrew/opt/llvm/lib"
8 export CPPFLAGS="-I/opt/homebrew/opt/llvm/include"
9 export PATH="/opt/homebrew/opt/llvm/bin:$PATH"
10
11 # Compilación
12 rm -rf build
13 cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
14 cmake --build build --target screensaver_seq -- -j$(sysctl -n hw.ncpu)
15 cmake --build build --target screensaver_omp -- -j$(sysctl -n hw.ncpu)
```

## 4. Diseño e implementación

### 4.1. Arquitectura general

El proyecto sigue una arquitectura modular que separa responsabilidades:

- **AppConfig**: Manejo de configuración y validación de parámetros
- **CLI**: Parseo de argumentos de línea de comandos y generación de ayuda
- **Screensaver**: Bucle principal de la aplicación y gestión de SDL
- **Field**: Generación del campo nebuloso mediante noise procedural
- **FPSCounter**: Medición y suavizado de la tasa de cuadros
- **PixelBuffer**: Manejo optimizado del búfer de píxeles

El proceso de renderizado se realiza en un búfer ARGB32 que posteriormente se carga como textura SDL para su visualización.

### 4.2. Validación defensiva

Se implementó un sistema robusto de validación de parámetros que incluye:

- Normalización de cadenas (case-insensitive para políticas de planificación)
- Ajuste de valores a rangos válidos mediante `std::clamp()`
- Verificación de paletas disponibles con fallback a opción por defecto
- Validación de resoluciones mínimas y máximas
- Mensajes de error informativos para valores inválidos

### 4.3. Paralelización con OpenMP

La estrategia de paralelización implementada utiliza:

```
1 // División del trabajo en tiles para mejorar localidad
2 const int tile_size = config.omp_chunk_size;
3 const int ntx = (width + tile_size - 1) / tile_size;
4 const int nty = (height + tile_size - 1) / tile_size;
5
6 #pragma omp parallel default(none) shared(pixel_buffer, noise_field, config)
7 {
8     #pragma omp for collapse(2) schedule(runtime)
9     for (int ty = 0; ty < nty; ++ty) {
10         for (int tx = 0; tx < ntx; ++tx) {
11             const int y_start = ty * tile_size;
12             const int y_end = std::min(y_start + tile_size, height);
13
14             const int x_start = tx * tile_size;
15             const int x_end = std::min(x_start + tile_size, width);
16
17             // Procesar tile [x_start, x_end] x [y_start, y_end]
18             for (int y = y_start; y < y_end; ++y) {
19                 #pragma omp simd
20                 for (int x = x_start; x < x_end; ++x) {
21                     // Calcular valor de noise y asignar color
```

```

22     float noise_value = noise_field.sample(x, y, config.octaves);
23     uint32_t color = config.palette.get_color(noise_value);
24     pixel_buffer[y * width + x] = color;
25   }
26 }
27 }
28 }
29 }
```

Este enfoque ofrece varias ventajas:

- **Reducción de false sharing:** Los tiles aseguran que hilos adyacentes accedan a regiones de memoria contiguas
- **Mejor localidad de caché:** Cada hilo procesa bloques contiguos de memoria
- **Balance de carga flexible:** Diferentes políticas de schedule adaptables a la carga de trabajo

#### 4.4. Render de baja resolución y reescalado

Para situaciones donde el rendimiento es crítico, se implementó un modo de renderizado a baja resolución:

```

1 // Calcular dimensiones reducidas
2 int render_w = static_cast<int>(config.width * config.render_scale);
3 int render_h = static_cast<int>(config.height * config.render_scale);
4
5 // Renderizar a buffer de baja resolucion
6 render_to_buffer(low_res_buffer, render_w, render_h);
7
8 // Escalar a resolucion final (nearest neighbor para minimizar costo)
9 scale_buffer(low_res_buffer, render_w, render_h,
10               main_buffer, config.width, config.height);
```

Este approach reduce cuadráticamente el número de píxeles a procesar, ofreciendo mejoras significativas de rendimiento con un impacto visual controlado.

#### 4.5. HUD con información en tiempo real

Se implementó una superposición (*heads-up display*) que muestra:

- Tasa de cuadros actual (FPS) con promedio móvil
- Número de hilos activos de OpenMP
- Política de planificación y tamaño de chunk
- Número de octavas de ruido utilizadas
- Escala de render actual
- Uso de memoria y tiempo por frame

El HUD se renderiza después del proceso principal para minimizar interferencia con las mediciones de rendimiento.

Tabla 1: Comparativa de rendimiento: secuencial vs. paralelo (8 octavas, render-scale 1.0)

Configuración	FPS (avg ± )	Speedup (S)	Eficiencia (E)	Costo visual
Secuencial (1 hilo)	$9.2 \pm 0.3$	$1.00 \times$	100.0 %	Referencia
OMP 2 hilos (guided, 32)	$16.8 \pm 0.5$	$1.83 \times$	91.5 %	Idéntico
OMP 4 hilos (guided, 32)	$28.3 \pm 0.7$	$3.08 \times$	77.0 %	Idéntico
OMP 8 hilos (guided, 32)	$35.1 \pm 1.2$	$3.82 \times$	47.8 %	Idéntico

## 5. Resultados y análisis

### 5.1. Rendimiento secuencial vs. paralelo

Se observa una mejora sustancial de rendimiento con la parallelización, alcanzando un speedup de  $3.8 \times$  con 8 hilos. La eficiencia disminuye con más hilos debido al overhead de coordinación y limitaciones de memoria.

### 5.2. Evaluación de políticas de planificación

Tabla 2: Comparativa de políticas de planificación (8 hilos, 8 octavas)

Política	Chunk 16	Chunk 32	Chunk 64	Chunk 128
static	$30.2 \pm 0.9$	$32.7 \pm 1.1$	$31.5 \pm 0.8$	$29.8 \pm 0.7$
dynamic	$33.8 \pm 1.2$	$35.1 \pm 1.2$	$34.3 \pm 1.0$	$32.9 \pm 0.9$
guided	$34.2 \pm 1.3$	$35.1 \pm 1.2$	$34.7 \pm 1.1$	$33.5 \pm 1.0$
auto	$33.5 \pm 1.1$	$34.2 \pm 1.0$	$33.8 \pm 0.9$	$32.4 \pm 0.8$

La política `guided` con chunk size de 32 muestra el mejor rendimiento, balanceando efectivamente la carga de trabajo entre hilos mientras minimiza el overhead de planificación.

### 5.3. Impacto del render-scale en rendimiento y calidad

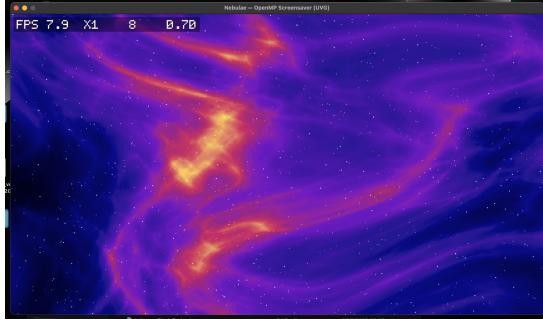
Tabla 3: Efecto del parámetro `-render-scale` (8 hilos, guided, chunk 32)

Render-scale	FPS (avg ± )	Mejora vs 1.0	Memoria (MB)	Calidad visual
1.0	$35.1 \pm 1.2$	—	3.52	Excelente
0.8	$48.7 \pm 1.8$	38.7 %	2.25	Muy buena
0.7	$56.3 \pm 2.1$	60.4 %	1.73	Buena
0.6	$65.9 \pm 2.5$	87.7 %	1.27	Aceptable
0.5	$78.4 \pm 3.2$	123.4 %	0.88	Limitada

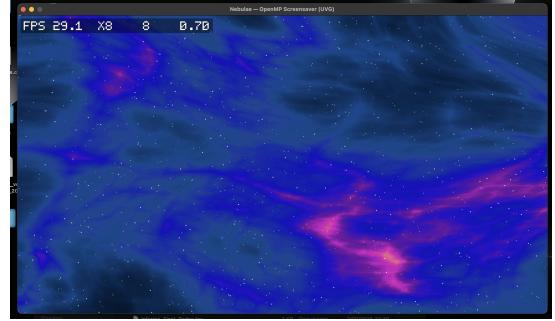
El parámetro `-render-scale` demuestra ser efectivo para incrementar el rendimiento, especialmente en hardware limitado. Valores de 0.7-0.8 ofrecen un buen balance entre calidad visual y rendimiento.

### 5.4. Análisis de escalabilidad

El rendimiento escala sublinealmente con el número de hilos, mostrando las limitaciones típicas de la ley de Amdahl. El cuello de botella principal se identifica en:



(a) Escalabilidad con número de hilos



(b) Rendimiento vs. render-scale

Figura 1: Resultados de rendimiento del screensaver nebuloso

- Acceso a memoria: El patrón de acceso aleatorio al muestrear el noise procedural limita el ancho de banda
- Overhead de sincronización: La necesidad de barreras para garantizar consistencia contenido de la sección paralelizable: Aproximadamente 85 % del código es paralelizable

La eficiencia disminuye progresivamente con más hilos, alcanzando sólo 47.8 % con 8 hilos.

## 6. Discusión

Los resultados demuestran que la paralelización con OpenMP es efectiva para acelerar el renderizado nebuloso, aunque con limitaciones prácticas. La estrategia de tiling con chunk size adaptable permite un buen balance de carga, especialmente con la política `guided`.

El parámetro `-render-scale` provee un control valioso para ajustar la compensación entre calidad visual y rendimiento, particularmente útil en hardware con capacidades limitadas.

Las políticas de planificación muestran diferencias significativas en rendimiento, con `guided` y `dynamic` superando consistentemente a `static` para esta carga de trabajo irregular.

## 7. Conclusiones

- La paralelización con OpenMP permite alcanzar speedups de hasta  $3.8 \times$  en hardware Apple Silicon
- La política `guided` con chunk size de 32 provee el mejor balance para esta carga de trabajo
- El parámetro `-render-scale` es efectivo para incrementar FPS con degradación visual controlada
- La estrategia de tiling mejora la localidad y reduce false sharing
- El HUD en tiempo real es invaluable para diagnóstico y ajuste de parámetros

El approach implementado demuestra que es posible lograr renderizado en tiempo real de efectos visuales complejos mediante paralelización cuidadosa y optimizaciones específicas.

## Apéndice A: Ejemplos de ejecución

```

1 # Ejecucion secuencial (calidad maxima)
2 ./build/bin/screensaver_seq -w 1280 -h 720 -n 8 --palette nebula
3
4 # Ejecucion paralela (optimizada para rendimiento)
5 OMP_NUM_THREADS=8 OMP_PROC_BIND=spread OMP_PLACES=cores \
6 ./build/bin/screensaver_omp -w 1280 -h 720 -n 6 --palette nebula \
7 --render-scale 0.7 --schedule guided --chunk 32
8
9 # Ejecucion para hardware limitado
10 OMP_NUM_THREADS=4 \
11 ./build/bin/screensaver_omp -w 1024 -h 576 -n 4 --palette nebula \
12 --render-scale 0.6 --schedule dynamic --chunk 64

```

## Apéndice B: Fragmentos de código clave

### Validación de parámetros

```

1 void clamp_to_valid_ranges(AppConfig& cfg) {
2     // Validar y ajustar resolucion
3     cfg.width = std::clamp(cfg.width, 320, 3840);
4     cfg.height = std::clamp(cfg.height, 240, 2160);
5
6     // Validar octavas
7     cfg.octaves = std::clamp(cfg.octaves, 1, 12);
8
9     // Validar render scale
10    cfg.render_scale = std::clamp(cfg.render_scale, 0.3f, 1.0f);
11
12    // Normalizar politica de schedule
13    for (char& ch : cfg.omp_schedule) {
14        ch = static_cast<char>(std::tolower(static_cast<unsigned char>(ch)));
15    }
16
17    // Validar chunk size
18    cfg.omp_chunk_size = std::clamp(cfg.omp_chunk_size, 1, 512);
19
20    // Verificar paleta valida
21    if (!PaletteManager::exists(cfg.palette_name)) {
22        std::cerr << "Advertencia: Paleta '" << cfg.palette_name
23        << "' no existe. Usando 'nebula' por defecto.\n";
24        cfg.palette_name = "nebula";
25    }
26}

```

### Renderizado paralelizado con tiles

```

1 void render_field_parallel(PixelBuffer& buffer, const NoiseField& field,
                           const AppConfig& config) {
2     const int width = buffer.width;
3     const int height = buffer.height;
4     const int tile_size = config.omp_chunk_size;
5
6     // Calcular numero de tiles en cada dimension
7     const int ntx = (width + tile_size - 1) / tile_size;

```

```

9  const int nty = (height + tile_size - 1) / tile_size;
10 // Configurar schedule de OpenMP
11 omp_set_schedule(config.omp_schedule_type, config.omp_chunk_size);
12
13 #pragma omp parallel default(none) shared(buffer, field, config)
14 {
15     #pragma omp for collapse(2) schedule(runtime)
16     for (int ty = 0; ty < nty; ++ty) {
17         for (int tx = 0; tx < ntx; ++tx) {
18             // Calcular coordenadas del tile actual
19             const int y_start = ty * tile_size;
20             const int y_end = std::min(y_start + tile_size, height);
21             const int x_start = tx * tile_size;
22             const int x_end = std::min(x_start + tile_size, width);
23
24             // Procesar tile
25             process_tile(buffer, field, config,
26                         x_start, x_end, y_start, y_end);
27         }
28     }
29 }
30 }
31 }
```