Universidad
Rey Juan Carlos

Práctica 2
-
Computación de Altas Prestaciones

Ingeniería de Computadores

Alberto Alegre Burcio

Antón Rodríguez Seselle

Contenido

Tabla de ilustraciones

Introducción

En esta práctica se ha realizado un estudio de la mejora de rendimiento de un algoritmo de ray tracing en CPU y en GPU mediante las tecnologías OpenMP, MPI y CUDA.

El pipeline seguido ha sido la aplicación independiente de cada tecnología, OpenMP y MPI en CPU y CUDA en GPU; y posteriormente la combinación de MPI y OpenMP.

Para las tecnologías de CPU se ha implementado paralelización en columnas, filas y regiones rectangulares, además de la paralelización del renderizado de varios fotogramas a la vez.

Para la recogida de datos y su posterior análisis se han realizado diferentes ejecuciones automáticas con scripts de python, y con los datos recogidos se han creado gráficas para observar los tiempos y mejoras con el uso de las tecnologías.

Implementación de tecnologías

MPI

Para la versión de solo MPI se ha utilizado como base el proyecto de ray tracing en CPU proporcionado, con lectura de fichero para cargar datos de la escena. Se ha preferido utilizar la escena cargada de fichero por encima de la generada aleatoriamente por su determinismo en cuanto a los resultados.

En esta versión se ha implementado paralelización de fotogramas, y reparto de la imagen entre procesos por columnas, filas y regiones rectangulares. A continuación, se explica el código desarrollado.

Primero se han programado la toma de argumentos en la ejecución, pudiendo pasarse el número de fotogramas a renderizar, el ancho y el alto de la imagen, entre otras cosas; esto con el objetivo de facilitar la realización de experimentos mediante scripts de Python. El número de procesos se pasa al comando mpiexec.

```
int main(int argc, char** argv) {
    /////////// nFotogramas, width, height, ns, strategy (cols|rows|blocks)
    int nFotogramas = std::atoi(argv[1]);
    int w = std::atoi(argv[2]);
    int h = std::atoi(argv[3]);
    int ns = std::atoi(argv[4]);
    std::string strategy = argv[5];
```

Ilustración 1. Manejo de argumentos MPI

Inmediatamente después se han realizado la inicialización del entorno MPI y se han obtenido el número de procesos en marcha y el identificador global del nodo actual.

```
MPI_Init(&argc, &argv);
int worldRank, worldNP;
MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
MPI_Comm_size(MPI_COMM_WORLD, &worldNP);
```

Ilustración 2. Inicialización MPI

Para optimizar el reparto de nodos para el renderizado de diferentes fotogramas, se ha puesto como límite 1 fotograma por proceso/nodo, puesto que, si se lanzasen más frames que procesos, sería una ejecución lenta por razones obvias, habría una “cola” de fotogramas en espera hasta que quedasen procesos libres y no se haría de forma paralela.

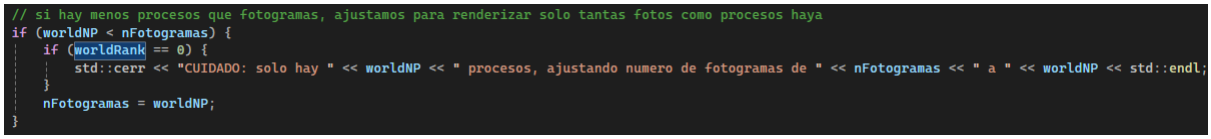

Ilustración 3. Exceso de frames MPI

Se ha realizado el reparto de los procesos de cada frame mediante un vector de cantidades de procesos y otro de offsets. En caso de no ser un reparto exacto y sobrar procesos, estos se incluyen en los primeros procesos de manera más o menos equitativa.

```
// repartir datos entre los fotogramas
std::vector<int> procsPerFrame(nFotogramas), frameStart(nFotogramas);
int base = worldNP / nFotogramas;
int rem = worldNP % nFotogramas;
for (int f = 0; f < nFotogramas; ++f) {
    procsPerFrame[f] = base + (f < rem ? 1 : 0);
    frameStart[f] = (f == 0 ? 0 : frameStart[f - 1] + procsPerFrame[f - 1]);
}

// a que fotograma pertenece
int frameIdx = 0;
for (int f = 0; f < nFotogramas; ++f) {
    if (worldRank >= frameStart[f] &&
        worldRank < frameStart[f] + procsPerFrame[f]) {
        frameIdx = f;
        break;
    }
}
```

Ilustración 4. Reparto de procesos entre fotogramas

Para la comunicación entre nodos del mismo frame, se ha creado un nuevo comunicador de MPI, obteniendo el identificador de nodo local y el tamaño del grupo.

```
// comunicador local por cada fotograma
MPI_Comm frameComm;
MPI_Comm_split(MPI_COMM_WORLD, frameIdx, worldRank, &frameComm);
int rank, np;
MPI_Comm_rank(frameComm, &rank);
MPI_Comm_size(frameComm, &np);
```

Ilustración 5. Comunicador de MPI

El nodo maestro de cada grupo realiza un print para debug en el que se muestran los datos del caso de ejecución. Después, calcula la región del frame que renderiza cada proceso y lo transmite mediante un broadcast. Este reparto se realiza mediante 3

funciones diferentes: una para reparto en columnas, otra para filas y otra para regiones rectangulares, las cuales se llaman en función de la estrategia elegida.

```
if (rank == 0) {
    std::cout << "Fotograma " << (frameIdx + 1) << "/" << nFotogramas
        << " -> imagen " << w << "x" << h
        << " con " << ns << " spp, strategy=" << strategy
        << std::endl;
}

// preparar los patches
std::vector<Patch> patches(np);
if (rank == 0) {
    for (int i = 0; i < np; ++i) {
        if (strategy == "cols")   patches[i] = divideByCols(w, h, np, i);
        else if (strategy == "rows")   patches[i] = divideByRows(w, h, np, i);
        else                            patches[i] = divideByBlocks(w, h, np, i);
    }
}
MPI_Bcast(patches.data(), sizeof(Patch) * np, MPI_BYTE, 0, frameComm);

Patch my = patches[rank];
```

Ilustración 6. Reparto de la imagen MPI

```
Patch divideByRows(int w, int h, int np, int rank) {
    int rows_per_proc = h / np;
    int extra_rows = h % np;

    int start_row = rank * rows_per_proc + std::min(rank, extra_rows);
    int num_rows = rows_per_proc + (rank < extra_rows ? 1 : 0);

    return { 0, start_row, w, start_row + num_rows };
}

Patch divideByCols(int w, int h, int np, int rank) {
    int cols_per_proc = w / np;
    int extra_cols = w % np;

    int start_col = rank * cols_per_proc + std::min(rank, extra_cols);
    int num_cols = cols_per_proc + (rank < extra_cols ? 1 : 0);

    return { start_col, 0, start_col + num_cols, h };
}
```

Ilustración 7. Paralelización en filas y en columnas

```cpp
Patch divideByBlocks(int width, int height, int num_processes, int rank) {
    // numero de filas
    int rows = static_cast<int>(std::floor(std::sqrt(num_processes)));
    if (num_processes >= 2) rows = std::max(rows, 2);
    rows = std::min(rows, num_processes);

    // numero de columnas
    int cols_base = (num_processes + rows - 1) / rows; // ceil(np/rows)

    // distribucion de procesos (columnas) por cada fila:
    std::vector<int> cols_in_row(rows);
    for (int r = 0; r < rows; ++r) {
        if (r < rows - 1)
            cols_in_row[r] = cols_base;
        else
            cols_in_row[r] = num_processes - cols_base * (rows - 1);
    }

    // para cada rank
    int row = rank / cols_base;
    int col = rank % cols_base;  // siempre col < cols_in_row[row] porque rank < np

    int baseH = height / rows;
    int remH = height % rows;
    int py = row * baseH + std::min(row, remH);
    int h_block = baseH + (row < remH ? 1 : 0);
    int ph = py + h_block;

    int thisRowCols = cols_in_row[row];
    int baseW = width / thisRowCols;
    int remW = width % thisRowCols;
    int px = col * baseW + std::min(col, remW);
    int w_block = baseW + (col < remW ? 1 : 0);
    int pw = px + w_block;

    return { px, py, pw, ph };
}
```

Ilustración 8. Paralelización en bloques/regiones rectangulares

Esto se maneja mediante un struct Patch.

```cpp
struct Patch {
    int px, py, pw, ph;
};
```

Ilustración 9. Struct Patch

Una vez se tiene esto, se crea el array de datos locales y se reserva la memoria del frame completo con valores iniciales 0 en cada componente RGB, esto para facilitar la reducción de datos al final después del ray tracing.

```cpp
unsigned char* local_data = (unsigned char*)calloc(w * h * 3, 1);
double init_time = 0.0, end_time = 0.0;
```

Ilustración 10. Datos locales y tiempo

El nodo maestro local inicia un contador de tiempo, y todos los nodos comienzan a realizar el ray tracing, pasando como argumentos el array local, el ancho y alto de la imagen, los muestreos, y las coordenadas globales de la región a renderizar.

```
if (rank == 0) init_time = omp_get_wtime();

rayTracingCPU(local_data, w, h, ns, my.px, my.py, my.pw, my.ph);
```

Ilustración 11. Inicio tiempo y llamada a ray tracing

En cuanto a la función de ray tracing en CPU, se han realizado algunos cambios de cálculo de coordenadas para que se adapte al uso de los parches generados. De esta forma, se utilizan coordenadas globales de la imagen.

Una vez se termina de realizar el trazado de rayos, se crea el array de datos globales para reducir los datos, reservando memoria solo en el maestro local. Mediante la función de reducción de MPI y la operación MPI_SUM, el maestro local obtiene los datos calculados de la imagen completa. Se ha decidido unificar los datos con MPI_Reduce debido a que cada píxel solo es modificado por un solo proceso y los píxeles no tocados por un nodo almacenan valores de 0, por lo que al sumar cada dato de los diferentes arrays locales, obtenemos el mismo resultado que se obtendría con un solo nodo renderizando.

```
unsigned char* global_data = nullptr;
if (rank == 0) global_data = (unsigned char*)calloc(w * h * 3, 1);
MPI_Reduce(local_data, global_data, w * h * 3, MPI_UNSIGNED_CHAR, MPI_SUM, 0, frameComm);
```

Ilustración 12. Reducción de datos

Finalmente, el nodo maestro local termina el contador de tiempo y vuelca los datos de la imagen en el archivo final, identificado con un id de imagen.

```
double frameTime = 0.0;
if (rank == 0) {
    end_time = omp_get_wtime();
    frameTime = end_time - init_time;
}

// crear la foto
if (rank == 0) {
    char filename[256];
    std::sprintf(filename, "../../../../MPI/Imagenes/imgCPUImg%d.bmp", frameIdx + 1);
    writeBMP(filename, global_data, w, h);
    std::cout << "Imagen creada en " << frameTime << " s" << std::endl;
    free(global_data);
}
```

Ilustración 13. Fin de tiempo local y escritura en fichero

Una vez se termina esto, se libera la memoria reservada y el comunicador. Los nodos maestros locales envían mediante mensaje síncrono el tiempo que ha tardado en

renderizarse su frame, recibiendo el maestro global cada tiempo mediante recepción síncrona, y calculando el tiempo total (tiempo máximo de entre los tiempos calculados).

```
free(local_data);

MPI_Comm_free(&frameComm);

// enviar tiempos de cada fotograma (solo si no es el proceso 0 global)
if (rank == 0 && worldRank != 0) {
    MPI_Send(&frameTime, 1, MPI_DOUBLE, 0, frameIdx, MPI_COMM_WORLD);
}

// el maestro recibe tiempos, calcula tiempo total y emite CSV
if (worldRank == 0) {
    std::vector<double> times(nFotogramas);
    if (rank == 0) times[0] = frameTime;
    for (int f = 1; f < nFotogramas; ++f) {
        MPI_Recv(&times[f], 1, MPI_DOUBLE,
            MPI_ANY_SOURCE, f, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    double totalTime = *std::max_element(times.begin(), times.end());
```

Ilustración 14. Liberación memoria, recogida de tiempos y tiempo global

Para finalizar, se printean los datos con un formato concreto para recoger los resultados en gráficas, y se finaliza el entorno de MPI.

```
        // para el CSV
        std::cout << nFotogramas << ","
            << w << ","
            << h << ","
            << ns << ","
            << worldNP << ","
            << totalTime;
        for (double t : times) {
            std::cout << "," << t;
        }
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Ilustración 15. Printeo de resultados y finalización MPI

OpenMP

Para la versión de solo OMP se ha partido del mismo proyecto del que se partió para MPI.

Se han implementado las mismas paralelizaciones que en MPI: renderizado de varios fotogramas en paralelo, y paralelización de trabajo en columnas, filas y regiones cuadradas.

Se ha planteado el reparto de trabajo entre hilos de forma análoga a MPI. Además de los argumentos que se pasaban en la ejecución de MPI, ahora se debe pasar el número de hilos OMP a utilizar (antes se pasaba el número de procesos de MPI en el comando mpiexec –n).

A continuación, se explica el código realizado.

Primero se ha manejado los argumentos que se pasan al ejecutable, en este orden: número de threads, número de frames a renderizar, ancho y alto de imagen, muestreos y estrategia de paralelización.

```
int main(int argc, char** argv) {
    // totalThreads, numFrames, w, h, ns, strategy (cols|rows|blocks)
    int totalThreads = std::atoi(argv[1]); // 8
    int numFrames = std::atoi(argv[2]); // 4;
    int w = std::atoi(argv[3]); // 1024;
    int h = std::atoi(argv[4]); // 1024;
    int ns = std::atoi(argv[5]); // 10;
    std::string strategy = argv[6];
```

Ilustración 16. Manejo de argumentos OMP

Se ha establecido el número de threads del programa mediante la función omp_set_num_threads y se han creado las variables para los tiempos, guardando el tiempo de inicio.

```
omp_set_num_threads(totalThreads);

double time_start, time_end;
time_start = omp_get_wtime();
```

Ilustración 17. Número de threads e inicio de tiempo

De manera parecida a la versión MPI, si hay más frames que threads, se ajusta el número de frames para que el reparto quede con un hilo por frame y no dejar ninguno en espera.

```
if (numFrames > totalThreads) {
    std::cerr << "CUIDADO: solo hay " << totalThreads << " procesos, ajustando numero de fotogramas de " << numFrames << " a " << totalThreads << std::endl;
    numFrames = totalThreads;
}
```

Ilustración 18. Exceso de fotogramas en OMP

A continuación, se ha creado un vector de frameBuffers reservando memoria para cada frame y se ha realizado el reparto de hilos entre los frames.

```
int bufferSize = sizeof(unsigned char) * w * h * 3;
std::vector<unsigned char*> frameBuffers(numFrames, nullptr);

for (int i = 0; i < numFrames; ++i) {
    frameBuffers[i] = (unsigned char*)calloc(bufferSize, 1);
}

std::vector<int> frameOffsets(numFrames);
std::vector<int> threadsPerFrame(numFrames);
int baseThreadPerFrame = totalThreads / numFrames;
int extra = totalThreads % numFrames;
int offset = 0;
for (int i = 0; i < numFrames; ++i) {
    threadsPerFrame[i] = baseThreadPerFrame + (i < extra ? 1 : 0);
    frameOffsets[i] = offset;
    offset += threadsPerFrame[i];
}
```

Ilustración 19. FrameBuffers y reparto de hilos entre frames

También se ha creado un vector de tiempos para medir el tiempo de cada frame.

```
std::vector<double> frameTimes(numFrames);
```

Ilustración 20. Vector de tiempos

Dentro de la región paralelizada con pragma omp parallel, se han creado variables de tiempo para el frame y las variables para identificar el hilo, su frame y su identificador dentro del frame. Se ha creado una función identifyThread para esto.

```
#pragma omp parallel
{
    double startLocal, endLocal;
    int tid = omp_get_thread_num();
    /*
    int threadsPerFrame = totalThreads / numFrames + (tid % numFrames < totalThreads % numFrames ? 1 : 0);
    int frameId = tid / threadsPerFrame; // id del frame sobre el que trabaja el hilo
    int threadInFrame = tid % threadsPerFrame; // id del thread dentro del grupo que trabaja en este hilo
    */
    int frameId = -1;
    int threadInFrame = -1;
    identifyThread(tid, threadInFrame, frameId, numFrames, frameOffsets, threadsPerFrame);
```

Ilustración 21. Inicio región paralela e identificación de thread

```
void identifyThread(int tid, int& threadInFrame, int& frameId, int numFrames, const std::vector<int>& frameOffsets, const std::vector<int>& threadsPerFrame) {
    for (int i = 0; i < numFrames; ++i) {
        if (tid >= frameOffsets[i] && tid < frameOffsets[i] + threadsPerFrame[i]) {
            frameId = i;
            break;
        }
    }
    threadInFrame = tid - frameOffsets[frameId];
}
```

Ilustración 22. Función de identificación de thread dentro de un fotograma

El hilo maestro local inicia el tiempo del frame. Después, cada hilo toma una referencia al frame sobre el que trabajan y se selecciona la parte del frame que le toca mediante funciones prácticamente iguales a las que se usan en la versión de MPI.

Tras ello, cada hilo realiza su ray tracing.

```
if (threadInFrame == 0){
    startLocal = omp_get_wtime();
}

unsigned char* data = frameBuffers[frameId];

Patch myPatch;
if (strategy == "cols") myPatch = divideByCols(w, h, threadsPerFrame[frameId], threadInFrame);
else if (strategy == "rows") myPatch = divideByRows(w, h, threadsPerFrame[frameId], threadInFrame);
else myPatch = divideByBlocks(w, h, threadsPerFrame[frameId], threadInFrame);

rayTracingCPU(data, w, h, ns, myPatch.px, myPatch.py, myPatch.pw, myPatch.ph);
```

Ilustración 23. Inicio de tiempo local y selección de región del frame

Se hace una barrera para sincronizar los hilos y cada nodo maestro local guarda la imagen creada en el archivo correspondiente. Con una región crítica se guarda el tiempo del frame en el vector de tiempos.

```
#pragma omp barrier
if (threadInFrame == 0) {
    std::string filename = "../../../../OMP/Imagenes/imgCPUImg" + std::to_string(frameId + 1) + ".bmp";
    writeBMP(filename.c_str(), data, w, h);

    endLocal = omp_get_wtime();

    #pragma omp critical
    {
        //std::cout << "Fotograma " << frameId + 1 << " guardado por hilo " << tid << std::endl;
        //std::cout << "Tiempo local " << (endLocal - startLocal) << std::endl;
        frameTimes[frameId] = (endLocal - startLocal);
    }
}
```

Ilustración 24. Escritura en fichero y fin de tiempo local

Una vez fuera de la región paralelizada, se ha tomado el tiempo de finalización global y se han realizado los prints pertinentes para el volcado de datos.

```
time_end = omp_get_wtime();
//std::cout << "Imagenes creadas en " << (time_end - time_start) << std::endl;

// para el CSV
std::cout << numFrames << ","
    << w << ","
    << h << ","
    << ns << ","
    << totalThreads << ","
    << (time_end - time_start);
for (double t : frameTimes) {
    std::cout << "," << t;
}
std::cout << std::endl;
```

Ilustración 25. Fin de tiempo global y printeo de resultados

Por último, se libera la memoria reservada.

MPI + OpenMP

Se han combinado las tecnologías MPI y OpenMP en otra versión del proyecto. Esta versión está basada en la antes explicada de MPI, añadiendo paralelización mediante OpenMP dentro de cada nodo MPI. El enfoque de la paralelización es el mismo: se reparte la imagen a renderizar en columnas, filas o bloques rectangulares para cada proceso que trabaje en ella. Sin embargo, ahora cada proceso subdivide la región o parche de la imagen que le toca, también en columnas, filas o bloques rectangulares, para cada hilo OMP. De esta manera se pueden combinar diferentes paralelizaciones: por ejemplo, se divide cada frame en regiones rectangulares y cada región en columnas.

En cuanto al renderizado de varios frames, funciona exactamente igual que la versión de solo MPI.

Ahora se pasan dos argumentos más respecto a MPI: los threads por nodo/proceso MPI y la estrategia de reparto entre hilos.

A continuación, se explica el código añadido.

Primero, se han añadido los nuevos argumentos.

```
int main(int argc, char** argv) {
    ////////// threadsPorProceso, nFotogramas, width, height, ns, strategy (cols|rows|blocks), subStrategy (cols|rows|blocks)
    int threadsPorProceso = std::atoi(argv[1]);
    int nFotogramas = std::atoi(argv[2]);
    int w = std::atoi(argv[3]);
    int h = std::atoi(argv[4]);
    int ns = std::atoi(argv[5]);
    std::string strategy = argv[6];
    std::string subStrategy = argv[7];
```

Ilustración 26. Manejo de argumentos MPI+OMP

En la línea de código donde se llamaba a rayTracingCPU, se ha establecido el número de hilos y se ha incluido la región paralela de OMP. Dentro de esta región cada hilo toma su id y la cantidad de threads del nodo. Cada hilo selecciona la subregión del frame sobre la que trabaja llamando a una función de reparto según la variable subStrategy. Estas 3 funciones de subdivisión son diferentes a las que se siguen utilizando por los procesos al comienzo del programa, puesto que aplica offsets que en las originales no se aplican. Después, cada hilo realiza el ray tracing.

```
omp_set_num_threads(threadsPorProceso);

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nt = omp_get_num_threads();

    Patch subpatch;
    if (subStrategy == "cols") {
        subpatch = subdivideByCols(my.pw - my.px, my.ph - my.py, nt, tid, my.px, my.py);
    }
    else if (subStrategy == "rows") {
        subpatch = subdivideByRows(my.pw - my.px, my.ph - my.py, nt, tid, my.px, my.py);
    }
    else {
        subpatch = subdivideByBlocks(my.pw - my.px, my.ph - my.py, nt, tid, my.px, my.py);
    }

    /*
    #pragma omp critical
    {
        std::cout << "Proceso " << worldRank << " thread " << tid << ": " << "maneja columnas
            << "y filas [" << subpatch.py << "," << subpatch.ph << "]\n";
    }
    */

    rayTracingCPU(local_data, w, h, ns, subpatch.px, subpatch.py, subpatch.pw, subpatch.ph);
}
```

Ilustración 27. Subdivisión entre threads y llamada a ray tracing

```
Patch subdivideByRows(int w, int h, int nt, int tid, int px, int py) {
    int rows_per_thread = h / nt;
    int extra_rows = h % nt;

    int start_row = tid * rows_per_thread + std::min(tid, extra_rows) + py;
    int num_rows = rows_per_thread + (tid < extra_rows ? 1 : 0);

    return { px, start_row, px+w, start_row + num_rows };
}

Patch subdivideByCols(int w, int h, int nt, int tid, int px, int py) {
    int cols_per_proc = w / nt;
    int extra_cols = w % nt;

    int start_col = tid * cols_per_proc + std::min(tid, extra_cols) + px;
    int num_cols = cols_per_proc + (tid < extra_cols ? 1 : 0);

    return { start_col, py, start_col + num_cols, py+h };
}
```

Ilustración 28. Funciones de subdivisión en filas y columnas

```cpp
Patch subdivideByBlocks(int w, int h, int nt, int tid, int offsetX, int offsetY) {
    // numero de filas
    int rows = static_cast<int>(std::floor(std::sqrt(nt)));
    if (nt >= 2) rows = std::max(rows, 2);
    rows = std::min(rows, nt);

    // numero de columnas
    int cols_base = (nt + rows - 1) / rows; // ceil(np/rows)

    // distribucion de procesos (columnas) por cada fila:
    std::vector<int> cols_in_row(rows);
    for (int r = 0; r < rows; ++r) {
        if (r < rows - 1)
            cols_in_row[r] = cols_base;
        else
            cols_in_row[r] = nt - cols_base * (rows - 1);
    }

    // para cada rank
    int row = tid / cols_base;
    int col = tid % cols_base;  // siempre col < cols_in_row[row] porque rank < np

    int baseH = h / rows;
    int remH = h % rows;
    int py = row * baseH + std::min(row, remH);
    int h_block = baseH + (row < remH ? 1 : 0);
    int ph = py + h_block;

    int thisRowCols = cols_in_row[row];
    int baseW = w / thisRowCols;
    int remW = w % thisRowCols;
    int px = col * baseW + std::min(col, remW);
    int w_block = baseW + (col < remW ? 1 : 0);
    int pw = px + w_block;

    return { px + offsetX, py + offsetY, pw + offsetX, ph + offsetY };
}
```

Ilustración 29. Función de subdivisión en bloques/regiones rectangulares

CUDA

Para la versión de CUDA se ha partido del código de ray tracing en GPU del aula virtual. Se ha modificado la generación de la escena para que tenga los mismos datos que se cargaban de fichero en las otras versiones en vez de generarse aleatoriamente. Esto se hace para que las pruebas sean deterministas y puedan compararse mejor con las pruebas de las otras tecnologías.

Al igual que en las otras versiones se pasa la configuración de ejecución por argumentos: ancho y alto de imagen, muestreos, threads en x y threads en y.

```
int main(int argc, char** argv) {
    ////////// width, height, ns, threadsX, threadsY
    int w = std::atoi(argv[1]);
    int h = std::atoi(argv[2]);
    int ns = std::atoi(argv[3]);
    int tx = std::atoi(argv[4]);
    int ty = std::atoi(argv[5]);
```

Ilustración 30. Manejo de argumentos CUDA

Estas cantidades de threads se pasan a la función rayTracingGPU para que calcule el número de bloques necesarios y lance el kernel con ellos.

```
rayTracingGPU(img, w, h, ns, tx, ty);
```

Ilustración 31. Llamada a ray tracing en GPU

También se ha añadido el printeo de datos para el volcado a tablas.

```
// para el CSV
std::cout << tx << ","
    << ty << ","
    << w << ","
    << h << ","
    << ns << ","
    << timer_seconds;
std::cout << std::endl;
```

Ilustración 32. Printeo de resultados

Instrucciones de compilación y ejecución

Se ha trabajado el código en un proyecto de CMake en visual studio con 4 subproyectos, uno para cada versión. Para ello se ha incluido un CMakeLists.txt en la carpeta raíz para configurar el proyecto, y otro CMakeLists.txt en cada carpeta de subproyecto para estos.

Se deben tener instaladas las herramientas de compilación de CMake en visual studio. Se debe abrir la carpeta raíz (Proyectos) con el archivo de configuración principal mediante "Abrir una carpeta local". Visual detectará los archivos de configuración y generará los archivos correspondientes a cada proyecto. Para compilar, se debe seleccionar el modo de compilación (Debug o Release, para las pruebas se ha usado Release) y en la parte superior de visual seleccionar Compilar > Compilar todo. Esto compilará todos los proyectos y generará sus ejecutables en la carpeta out, situada en la carpeta raíz, con ruta: out/build/<modoDeCompilacion>/<nombreDelSubproyecto>. Por ejemplo, el ejecutable de la versión de OpenMP + MPI compilado con x64-Release estará en out/build/x64-Release/MPIOMP/mpi_omp_version.exe. Es esencial mantener la estructura de la carpeta y cada archivo en la carpeta en la que está para que se acceda correctamente a los archivos que hacen falta: archivo Scene1.txt y carpeta Imagenes de cada subproyecto.

Ejecución de OpenMP y CUDA

Para la ejecución de las versiones de solo OMP y de CUDA, se debe abrir la terminal de Windows y situarse en la carpeta del ejecutable correspondiente. Se debe ejecutar escribiendo ".\<nombre_del_ejecutable>", seguido de los argumentos separados por un espacio.

Para la versión de OpenMP, los argumentos se deben escribir en este orden:

1- Número total de hilos omp
2- Número de frames a renderizar
3- Ancho (width) de imagen
4- Alto (height) de imagen
5- Muestreos
6- Estrategia de paralelización: "cols"/"rows"/"blocks"

Por ejemplo:

.\omp_version.exe 8 4 256 256 5 cols

Para la versión de CUDA:

1- Ancho (width) de la imagen
2- Alto (height) de la imagen

3- Muestreos
4- Threads en x
5- Threads en y

Por ejemplo:

.\cuda_version.exe 256 256 5 8 8

Ejecución de MPI + OpenMP y MPI

Es necesario al igual que en los dos casos anteriores abrir la terminal de Windows y situarse en la carpeta del ejecutable correspondiente.

Para estos es necesario ejecutar utilizando el comando de Windows “mpiexec” con opción “-n”, indicando el número de procesos/nodos MPI y con el nombre del ejecutable. Por ejemplo:

mpiexec –n 4 .\ejecutable

Esto se acompaña de los parámetros separados por espacios.

Para el caso de MPI solo, los parámetros en orden son:

1- Número de fotogramas
2- Ancho de imagen
3- Alto de imagen
4- Muestreos
5- Estrategia de paralelización: “cols”/”rows”/”blocks”

Por ejemplo:

mpiexec –n 4 .\mpi_version.exe 2 512 512 10 blocks

Para el caso de MPI + OpenMP, los parámetros en orden son:

1- Número de hilos OMP por proceso MPI
2- Número de frames
3- Ancho de imagen
4- Altura de imagen
5- Muestreos
6- Estrategia de paralelización entre procesos: “cols”/”rows”/”blocks”
7- Estrategia de paralelización entre hilos de un proceso: “cols”/”rows”/”blocks”

Por ejemplo:

mpiexec –n 8 .\mpi_omp_version.exe 4 2 1024 1024 10 blocks rows

Experimentación

Para automatizar las ejecuciones de las diversas versiones implementando tecnologías, se han creado scripts de python que lanzan el programa variando los parámetros de entrada. Los scripts lanzan ejecuciones del programa y capturan su salida, volcándola en un archivo excel "experimentos.csv" que contiene también los resultados de los tiempos obtenidos de ejecución total y parcial por fotograma en caso de que haya paralelización de distintos fotogramas.

Para MPI, los parámetros que se modifican y sus distintos posibles valores son:

```
fotogramas  = [1, 2, 5]
widths      = [32, 64, 256, 512, 1024]
heights     = [32, 64, 256, 512, 1024]
nss         = [1, 2, 5, 10]
nprocs_list = [1, 2, 4, 8, 16]
strategies  = ["cols", "rows", "blocks"]
```

Ilustración 33. Parámetros de prueba MPI

Para OMP, los parámetros son muy similares, pero en lugar de procesos, se usan hilos en memoria compartida.

```
fotogramas    = [1, 2, 5]
widths        = [32, 64, 256, 512, 1024]
heights       = [32, 64, 256, 512, 1024]
nss           = [1, 2, 5, 10]
nthreads_list = [1, 2, 4, 8, 16]
strategies    = ["cols", "rows", "blocks"]
```

Ilustración 34. Parámetros de prueba OMP

En la implementación que combina OMP y MPI se modifica el número de procesos, el número de hilos por proceso y tanto la estrategia de MPI como la de OMP.

```
fotogramas      = [1, 2, 3]
widths          = [32, 256, 512, 1024]
heights         = [32, 256, 512, 1024]
nss             = [1, 3, 10]
nprocs_list     = [1, 4, 8, 16]
threads_list    = [4, 8, 12]
strategies      = ["cols", "rows", "blocks"]
substrategies   = ["cols", "rows", "blocks"]
```

Ilustración 35. Parámetros de prueba MPI+OMP

En la implementación de CUDA los parámetros a cambiar incluyen el tamaño de la imagen como antes, así como el número de muestras y las dimensiones de hilos dentro de cada bloque. Con estos datos, dado el ancho y alto de hilos, calcula el programa cuantos bloques debe haber.

```
widths        = [32, 64, 256, 512, 1024]
heights       = [32, 64, 256, 512, 1024]
ns_list       = [1, 2, 5, 10]
threadsX_list = [1, 4, 8, 16, 32]
threadsY_list = [1, 4, 8, 16, 32]
```

Ilustración 36. Parámetros de prueba CUDA

Una vez obtenidos los resultados temporales de cada ejecución, mediante otro script de python se generan gráficas, tablas y medias para expresar de una mejor forma los resultados obtenidos (Se encuentran en la carpeta “Resultados” de la entrega del proyecto). A continuación, se van a explicar los resultados obtenidos en cada una de las tecnologías implementadas.

Resultados y análisis de rendimiento

MPI

Se implementan 3 estrategias de partición: por columnas, filas y bloques rectangulares. Cada estrategia reparte los píxeles entre los procesos asignados en forma de columnas, filas o bloques según la estrategia elegida. En pruebas de escala grande (1024 x 1024, 10 muestras) los 3 métodos tienen tiempos similares para un proceso (alrededor de 21 segundos). Sin embargo, la estrategia por columnas es la más escalable: al aumentar procesos de 1 a 16 el tiempo pasa de 21 segundos a 3,13 segundos, mientras que filas y bloques solo mejoran a 3,68-3,83 (véase la tabla inferior). Esto indica una mayor mejora con columnas.

Número de procesos	Columnas	Filas	Bloques
1	21,323	21,013	21,379
2	11,526	11,420	11,572
4	6,144	7,234	7,205
8	3,555	4,126	4,012
16	3,130	3,681	3,832

Cabe destacar que se observa un cierto punto en el que se reduce la aceleración: de 8 a 16 procesos la ganancia es mucho menor. Por ejemplo, la estrategia de columnas mejora de 3,55 a 3,13 segundos, indicando costes de comunicación y finalización. No obstante, hasta 8 procesos la mejora es casi ideal. El procesamiento por filas y bloques escala peor que por columnas, debido probablemente a la localidad de los datos accedidos por cada proceso.

Dichas conclusiones se pueden apreciar en las gráficas generadas para la media de tiempo por frame y tiempo total:

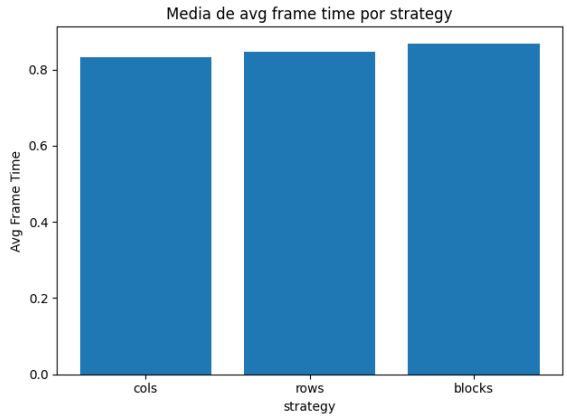

Ilustración 38. Tiempo medio por frame por estrategia MPI

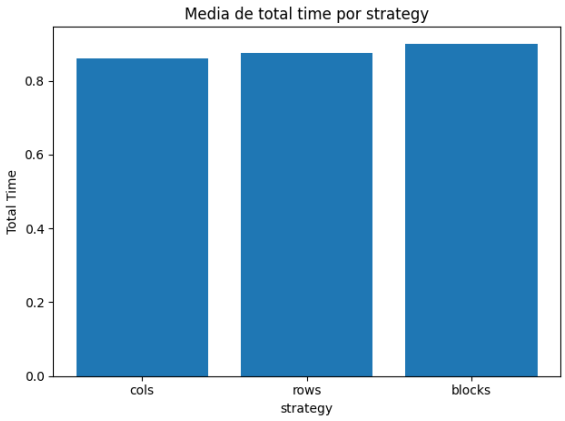

Ilustración 37. Tiempo medio de ejecución por estrategia MPI

Manteniendo fijo el ancho de la imagen, doblar la altura duplica aproximadamente el número de píxeles y, por tanto, el tiempo medio por frame crece también casi el doble:

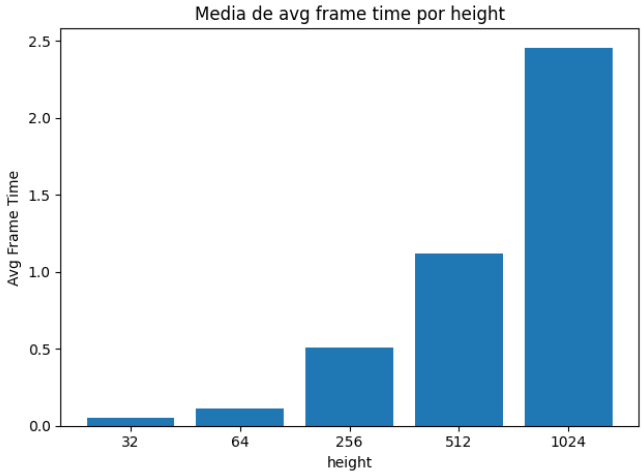

Ilustración 39. Tiempo medio por frame en función de altura MPI

- Altura 32: 0,04 segundos de media
- Altura 64: 0,09 segundos de media
- ...
- Altura 1024: 2,1 segundos de media

Se aprecia un crecimiento lineal en los tiempos a medida que aumenta el alto.

En cuanto al número de muestras por píxel, cuantas más haya, hay más trabajo y por tanto el tiempo de render crece en proporción directa a las muestras:

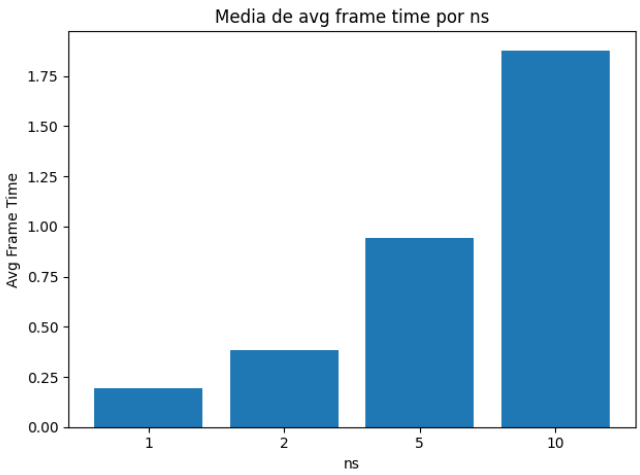

Ilustración 40. Tiempo medio por frame en función de muestreos MPI

Sucede de manera similar a con el tamaño de la imagen, el tiempo crece de manera lineal con respecto al número de muestras.

Sin embargo, esa linealidad observada en las características anteriormente descritas no se aprecia en la gráfica de número de fotogramas.

Al haber implementado paralelización de fotogramas, en los casos de prueba ejecutados se observa que el incremento temporal de 2 fotogramas con respecto a 1 no supone el doble de tiempo como supondría en caso de ejecución secuencial de fotogramas, sino alrededor del +50% de tiempo. En caso de paralelización de 5 fotogramas, en lugar de un incremento del +400% en el caso de la secuencialidad, supone alrededor de un 140%.

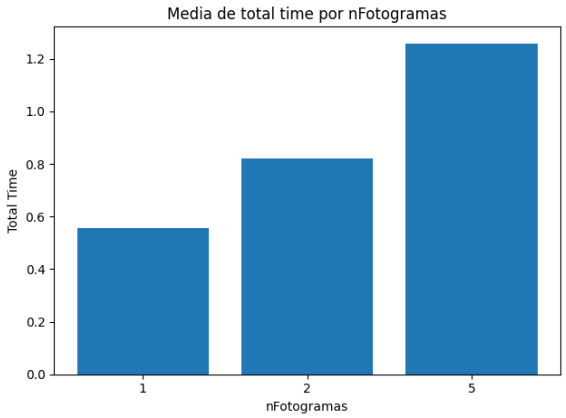

Ilustración 42. Tiempo medio de ejecución en función del número de fotogramas MPI

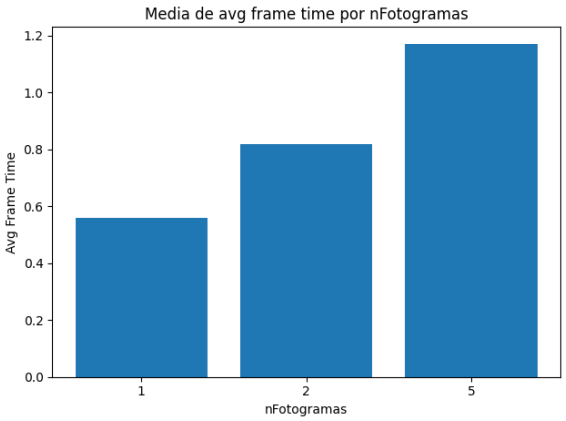

Ilustración 41. Tiempo medio por frame en función del número de fotogramas MPI

Cabe destacar que en tamaños muy pequeños de imagen, el overhead de arranque/sincronización de MPI es más visible que en grandes tamaños de imagen y más número de muestras.

Es razonable apreciar como el tiempo promedio de renderizado cae a medida que se aumenta el número de procesos de MPI:

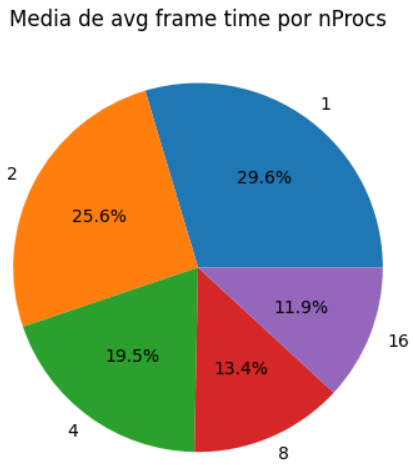

Ilustración 43. Tiempo medio por frame en función del número de procesos MPI

La disminución del tamaño de cada porción con respecto al incremento de procesos supone el speed-up de repartir el trabajo paralelamente. Cuantos más procesos, menor promedio de tiempo por frame. Además, con el crecimiento de la velocidad con el número de procesos, se compensa el tiempo usado para sincronización y comunicación de procesos.

Se aprecia también que los retornos no son lineales: El mayor salto se produce al doblar de 1 a 2 procesos y de 2 a 4. A partir de 8 la mejora es pequeña, ya que el trozo solo pasa de 13,4% a 11,9%, lo que indica que el overhead de sincronización/comunicación de MPI empieza a llevarse una buena parte de la ganancia del paralelismo.

OpenMP

En OpenMP con hilos en memoria compartida se lanzan ejecuciones como las anteriores pero cambiando procesos por hilos.

Los resultados de OpenMP son bastante parecidos a los de MPI. En el caso de imágenes grandes como se explicaba en el caso de MPI se obtiene un resultado bastante similar: Una imagen de 1024 x 1024 y 10 muestras para un solo hilo se tardan alrededor de 20,9 segundos con las 3 estrategias. Al aumentar de 1 a 16 hilos la estrategia de columnas baja a 3,14 segundos, mientras que filas y bloques bajan a 3,68 y 3,65 respectivamente, dándonos el resultado esperado y verosímil a los resultados obtenidos con MPI.

Pese a ser tiempos muy similares a MPI, se observa una ligerísima mejora, probablemente por la ausencia de mensajes, puesto que OMP tiene comunicación implícita, y el uso de la memoria compartida, mejorando las latencias de acceso a datos cercanos en memoria. No obstante, la mejora no es tan grande como cabría esperar debido a que la memoria compartida para nuestro reparto de trabajo es un arma de doble filo: los aciertos de caché reducen mucho la latencia, pero los fallos la aumentan.

A continuación se muestra una tabla con los resultados obtenidos en los casos que se comentan:

Hilos	Columnas	Filas	Bloques
1	20,878	20,958	20,903
2	11,542	11,222	11,303
4	6,138	7,210	6,995
8	3,500	4,113	3,901
16	3,142	3,680	3,647

A pesar de tener modelos diferentes, MPI y OMP con memoria distribuida y memoria compartida, respectivamente, los tiempos son muy cercanos, lo que demuestra que el código escala de forma equivalente. En OMP, al igual que en MPI, la estrategia de columnas es la más eficiente y escalable. Se puede apreciar, nuevamente aunque de forma muy sutil, en las gráficas de tiempo total y tiempo medio por frame:

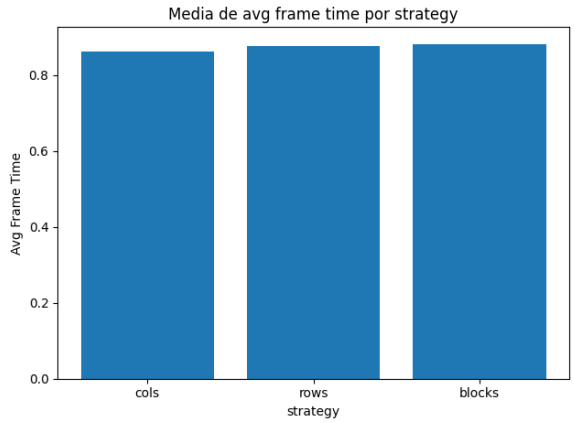

Ilustración 44. Tiempo medio por frame en función de la estrategia de paralelización OMP

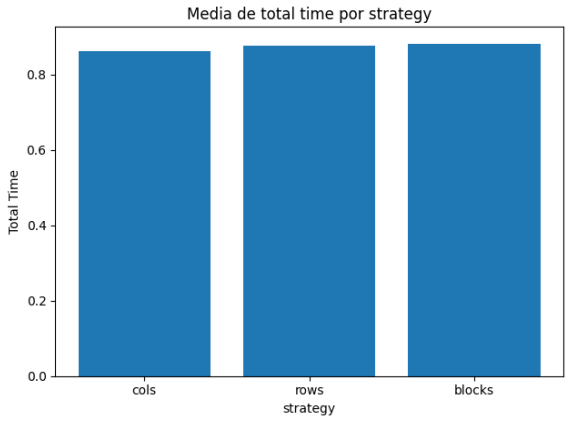

Ilustración 45. Tiempo medio de ejecución en función de la estrategia de paralelización OMP

Los resultados obtenidos en materia de gráficas son muy similares a los obtenidos con MPI. Merece la pena analizar la gráfica de comparativa del tiempo total en base al número de hilos (colores azules, amarillo, verde, rojo, lila).

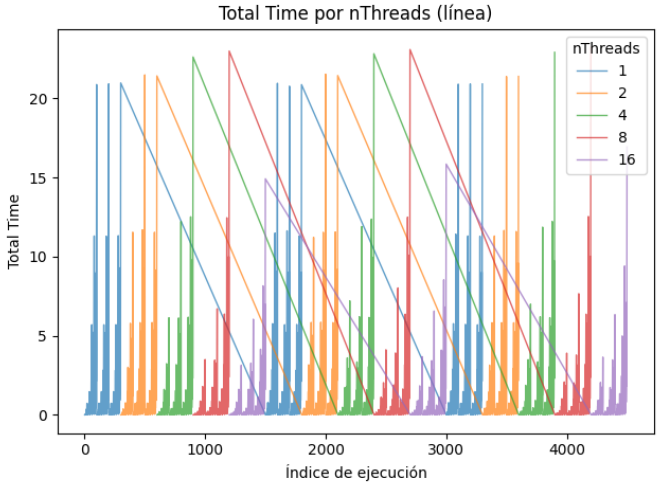

Ilustración 46. Tiempo totales de cada ejecución OMP

Cada pico que se aprecia es la sucesión de tiempos totales para un cierto número de hilos. A medida que incrementamos los hilos, el tercer pico de tiempo aumenta mientras que los dos primeros se van reduciendo cada vez más, hasta que al usar 16 hilos, los tres se reducen respecto al caso de 8 hilos.

El solapamiento entre curvas nos indica que, dependiendo del tamaño del trabajo, la ganancia varía, pero la tendencia general es de ahorro de tiempo creciente con más hilos.

Se han generado boxplots para analizar la mediana y la variabilidad de los datos. En el caso del tiempo total con respecto al número de hilos, se aprecia que el rango intercuartílico y los bigotes se hacen más pequeños a medida que aumentan los hilos, lo que implica que la variabilidad disminuye porque, a imágenes más grandes, el coste de paralelizar amortiza mejor las diferencias entre resoluciones y muestras.

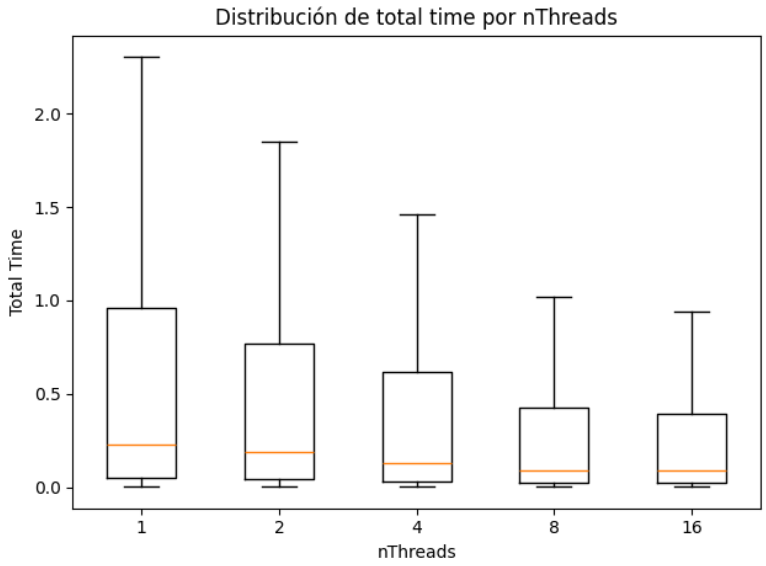

Ilustración 47. Boxplots OMP

Se observan también retornos decrecientes, ya que de 1 a 2 la reducción es grande, pero de 8 a 16 apenas mejora, lo cual significa que el overhead de gestión de hilos y los recursos (cachés, memoria…) empiezan a reducir el speed-up.

En forma de diagrama de tarta, se refleja el peso relativo de la cantidad de muestras por píxel en el tiempo medio por frame, lo cual confirma que el coste escala casi linealmente con el número de muestras, como sucedía en MPI: doblar ns, dobla prácticamente el tiempo medio por frame.

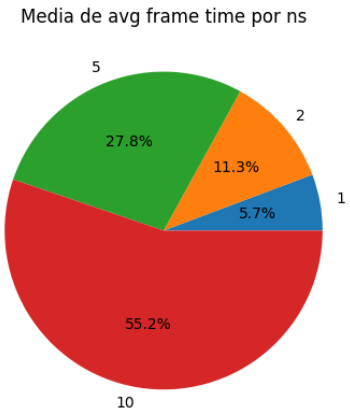

Ilustración 48. Tiempo medio por frame en función de los muestreos OMP

Con todo, se observa que el programa implementado con OMP tiene buen speed-up con hasta 8 hilos y se aprovecha bien el paralelismo, pero doblar el número de hilos hasta 16 compensa poco en cuanto a mejora temporal (más o menos del 10%), lo cual sugiere que el hardware tiene unos núcleos físicos limitados y los hilos extra compiten por los mismos recursos.

MPI + OpenMP

Para la implementación combinada de dos tecnologías se ha optado por MPI + OMP y en los resultados se aprecia una mejora temporal cuantiosa con respecto a las obtenidas con las tecnologías independientemente. En este esquema se combinan dos niveles de paralelismo: MPI reparte el dominio de la imagen entre varios procesos y OpenMP se emplea dentro de cada proceso MPI de paralelizar el render de pixeles entre múltiples hilos de CPU en memoria compartida.

Se han tomado mediciones temporales bastante más cuantiosas que en los casos anteriores porque se incluyen 2 nuevos parámetros (subestrategia y número de hilos por proceso), por lo que las muestras tomadas alcanzan los 15553 datos.

El tiempo medio por cada frame crece prácticamente de forma lineal como hemos observado en las implementaciones anteriores y como se observa en la gráfica:

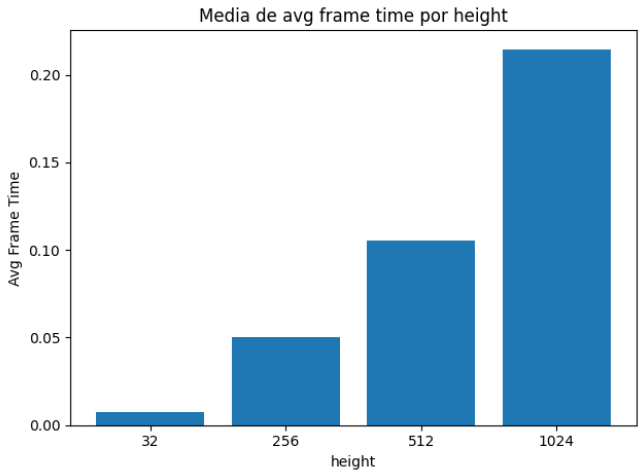

Ilustración 49. Tiempo medio por frame en función de la altura MPI+OMP

Esto confirma que, incluso, con dos niveles de paralelismo, el coste computacional va a escalar linealmente con el tamaño de la imagen.

En cuanto a las estrategias tomadas, se ha sacado un bloque de alrededor de 5000 ejecuciones con diferentes parámetros:

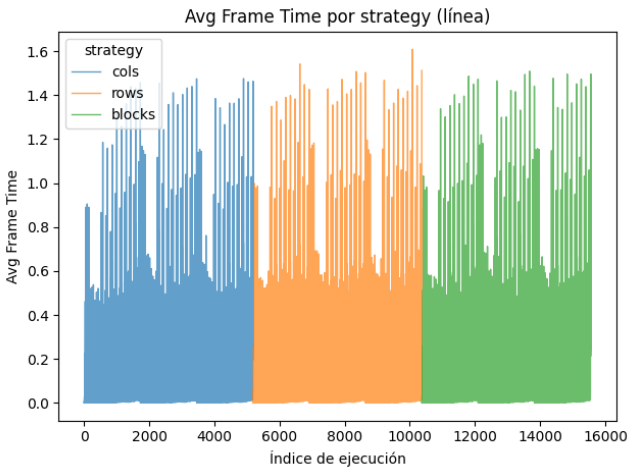

Ilustración 50. Tiempos por caso de ejecución MPI+OMP

Si bien parecen casi similares las 3 estrategias, nuevamente las columnas terminan siendo las que mejores tiempos dan y superan en rendimiento al resto de estrategias.

Para mostrar como la implementación combinada de MPI y OMP, es más rápida que las implementaciones parciales, se pueden observar los excels de las medias y medianas obtenidos, donde claramente la implementación combinada es mucho más rápida que la de OMP y MPI por separado.

CUDA

En CUDA la paralelización se logra ajustando los parámetros de los bloques e hilos. No se usan estrategias de filas o columnas, ya que el kernel se lanza sobre toda la imagen y los parámetros de threadX y threadY y definen el tamaño de bloque de hilos.

Por ejemplo, para la imagen de 1024 x 1024 con 10 muestras los mejores tiempos se obtuvieron con bloques de tamaño intermedio (por ejemplo, 16 x 8 hilos, 128 hilos por bloque), alrededor de 1,96 segundos, mientras que tamaños de bloque muy estrechos, 1 x 8 hilos por ejemplo, fueron mucho más lentos (más o menos 7,45 segundos). Esto concuerda con la idea de que bloques pequeños desperdician recursos de warp.

En general, CUDA ha sido la tecnología más rápida de las versiones implementadas, si bien su esquema de programación no tiene nada que ver con los implementados en CPU.

Cabe destacar que en las ejecuciones de CUDA se han obtenido resultados equilibrados para la mayoría de las ejecuciones. Si bien en las implementaciones de memoria compartida y distribuida en CPU había picos considerables en el tiempo de ejecución, la implementación en GPU con CUDA da valores más normales entre las distintas ejecuciones del programa.

El gráfico de a continuación muestra como varía el tiempo medio de render por frame al cambiar simplemente la dimensión Y de los bloques.

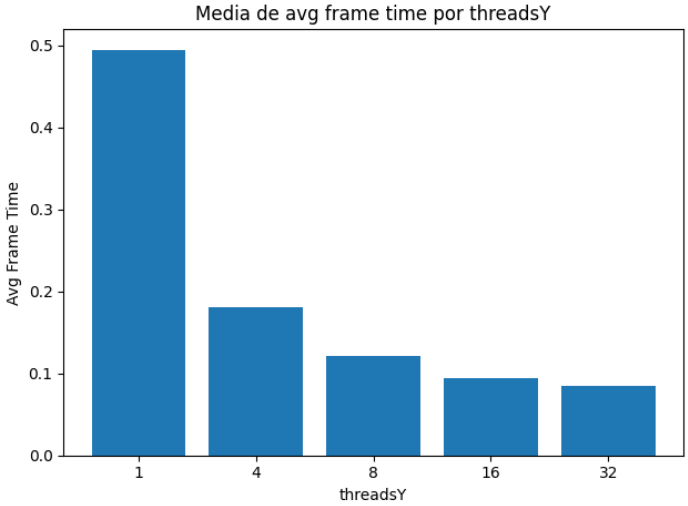

Ilustración 51. Tiempo medio en función de los threads Y

Con 1 thread en Y la GPU está infrautilizada con muy poca ocupación de multiprocesadores y un coste por frame de casi 0,5 segundos. Al crecer los threadY, la ocupación sube y se llenan mejor los warps, por lo que el tiempo baja drásticamente.

A partir de 16 y 32 hilos se empieza a llegar a la conclusión que se llegaba con CPU: La mejora adicional es marginal, señal de que estamos saturando la unidad de cómputo.

En otra gráfica se puede apreciar la curva de tiempo total para distintos valores de número de muestras:

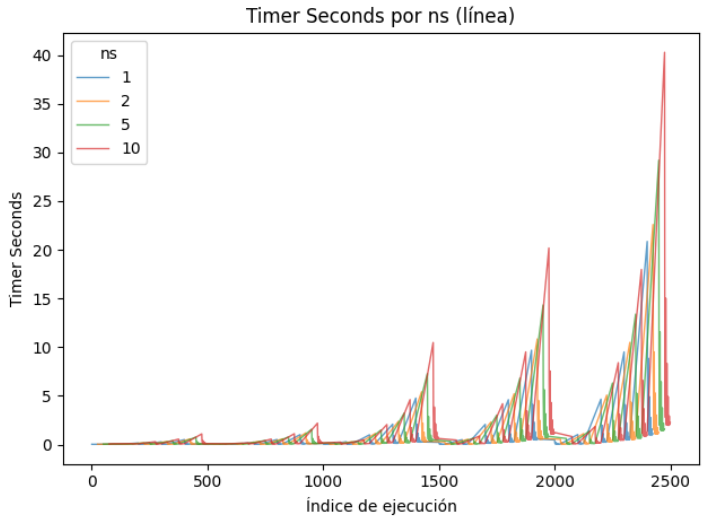

Ilustración 52. Tiempo en función de número de muestreos

Todas las curvas tienen forma idéntica, pero es la amplitud la que crece. El overhead de lanzamiento de kernel y memcpy es prácticamente fijo e imperceptible a gran escala, pero sí se ve un pequeño cambio en la base de la curva cuando la carga es mínima.

En CUDA elegir unas dimensiones threadX y threadY múltiplo de un warp (32) va a maximizar el throughput como hemos visualizado en las gráficas. El tiempo va a crecer linealmente con el número de píxeles y de muestras, tal y como sucedía con MPI y OMP, pero a mucha más velocidad.

Cabe destacar también que la mayor parte del tiempo se invierte en cómputo puro en el algoritmo de ray tracing, ya que la transferencia de datos y la configuración de hilos apenas penaliza en cargas grandes.

Conclusiones

Hemos concluido que de entre todas las estrategias de paralelización (columnas, filas o regiones cuadradas) la mejor es la de columnas debido a la cercanía de los datos. Puesto que los píxeles se almacenan en el código de manera row-major, y cada región se comienza a escribir en su elemento de menor X y menor Y, la distancia en memoria entre el comienzo de una columna y el comienzo de la siguiente, es mucho más cercano, lo que permite que sea mucho más probable que al acceder un dato, este ya se encuentre en caché y haya menos fallos que los que habría en filas o en regiones rectangulares.

Se ha concluido que OMP supone una muy ligera mejora sobre MPI debido al uso de la caché, comunicación simplificada y por la arquitectura de los ordenadores usados.

OpenMP resulta una opción mucho más amigable para el desarrollador que MPI, debido a que lo libera de las comunicaciones entre nodos. Por ello, en caso de tener que elegir una de las dos y no poder usar ambas, OpenMP supone una mejor opción.

En comparación con CUDA, MPI es un tecnología considerablemente más difícil de configurar, permitiendo CUDA una paralelización mucho más escalable y potente para operaciones simples con grandes cantidades de datos, como es el caso del ray tracing. Pese a ser más sencillo OMP para el desarrollador, CUDA aprovecha mejor los recursos, usando la GPU.

Otra conclusión a la que se ha llegado es que no siempre por aumentar los recursos se reduce el tiempo de ejecución, a veces la configuración y sincronización de las partes supone un costo de tiempo mayor a la mejora producida.