SkewEngine: Reorganización de mallas regulares para cálculo intensivo

Felipe Romero [1], Luis F. Romero [2], Gerardo Bandera[2]

Resumen— **En muchas aplicaciones, tales como la manipulación de datos hiperespectrales, la exploración de datos de resonancia magnética o la identificación de cuencas visuales en modelos digitales de elevación, es necesario realizar operaciones aritméticas sobre cada punto de una malla de datos que implican al resto de puntos de la misma, lo que puede resultar en un problema computacionalmente intratable. Se presenta en este trabajo SkewEngine, una herramienta diseñada para mejorar el rendimiento de cálculos intensivos en mallas regulares de datos 2D o 3D, como imágenes, volúmenes de datos multiespectrales o modelos digitales de elevación. SkewEngine soluciona este problema mediante la reorganización de la malla en memoria según una dirección espacial preferente, lo que permite una mayor eficiencia en la realización de cálculos intensivos. Se demuestra que SkewEngine ofrece una mejora significativa en la velocidad de los cálculos para una variedad de casos de prueba, lo que sugiere que puede ser una herramienta útil en una mucho más amplia gama de aplicaciones en las que se requiere procesamiento intensivo de datos en mallas regulares.**

Palabras clave— **Paralelismo embarazoso, Jerarquía de Memoria, Mallas estructuradas**

I. Introducción

EXISTEN problemas, en muchos y muy diversos campos, en los que es necesario realizar un cálculo extremadamente intensivo sobre cada uno de los puntos de una malla de datos 2D o 3D, como por ejemplo, sobre cada uno de los píxeles de una imagen, sobre cualquier punto de un mapa, o en los datos de una resonancia magnética.

En algunos casos, la intensidad aritmética es tan alta, que hace que el problema sea intratable, desde el punto de vista computacional. Veamos el ejemplo de la cuenca visual de un punto, en un modelo digital de elevaciones. Imagine que desea conocer qué parte de un territorio es visible desde un determinado lugar del territorio:

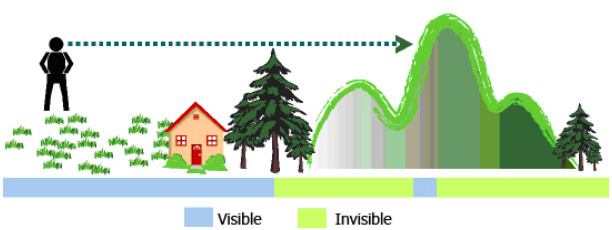


Fig. 1: Ejemplo de cuenca visual de un punto.

Como es obvio, para saber si desde un punto A se puede ver B (donde B puede ser cualquier otro punto de la geografía considerada), habría que tener en cuenta la altura del resto de puntos del modelo, ya que cualquiera, a priori, puede ser un obstáculo para la visión. Así, en un modelo digital de elevaciones, o DEM, (de *digital elevation model*), con $N = dimx \times dimy$ datos, la complejidad del problema, usando la notación big-O, sería de orden $O(N^2)$, aunque es evidente que dicha complejidad se puede reducir si sólo se consideran como obstáculos los puntos C que están en la línea A-B. Aún así, la complejidad del problema, $O(N^{1,5})$, es bastante alta. La conocidad aplicación Google Earth, por ejemplo, tarda varios segundos en obtener un resultado aproximado.

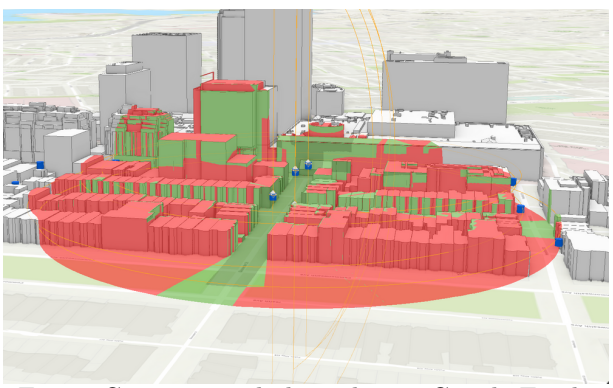


Fig. 2: Cuenca visual obtenida con Google Earth.

Cuando se desea calcular la cuenca visual no sólo ya de un observador situado en un lugar concreto de un territorio, sino la que se obtendría desde una trayectoria arbitraria que transcurre por el terreno, o la que se observaría desde una región, o incluso desde todo el territorio, donde cualquier punto de un DEM se convierte en observatorio. En este caso, la complejidad se dispara a $O(N^2{,}5)$. Incluso con baja precisión, se requieren meses de CPU para un cálculo de un modelo sencillo.

Afortunadamente, este tipo de problemas pertenece a una categoría en la que los parámetros de estudio están afectados por un decaimiento que depende de la distancia geométrica en la malla de datos, ya sea lineal o cuadrático, siendo esto último lo más frecuente. Son esos problemas en la que la influencia de un punto alejado (como puede ser una mariposa en Singapur), no afecta, o al menos inmediatamente, a lo que sucede en el otro extremo de la geometría (p.e., el Caribe). Este tipo de problemas se suele simplificar mediante un procesamiento en un conjunto discreto de direcciones radiales respecto a un punto de estudio, ya que los radios están más proximos en el punto de origen, y por tanto, la malla es más fina en el punto de estudio.

[1]Dpto. Informática, CEiA3, Universidad de Almería, e-mail: fr@uma.es

[2]Dpto. Arquitectura de Computadores, Universidad de Málaga, e-mail: felipe@uma.es, gbandera@uma.es

Sin embargo, en este tipo de problemas, donde la localidad espacial de la información es un factor crítico, existe la necesidad de trasladar dicha localidad espacial a los propios datos que representan a la información, y muy en particular, al almacenamiento en memoria de los mismos. Veamos un ejemplo. Imagine que en la imagen de la Fig. 3 (izquierda) se desea aplicar un filtro en todas las direcciones paralelas a la de la flecha negra. Es evidente que cualquier algoritmo de cierta complejidad de los que suelen aplicarse en gráfica computacional, como la fft, sería mucho más eficiente si los datos estuvieran alineados en memoria, como ocurriría en la imagen deformada de la derecha.

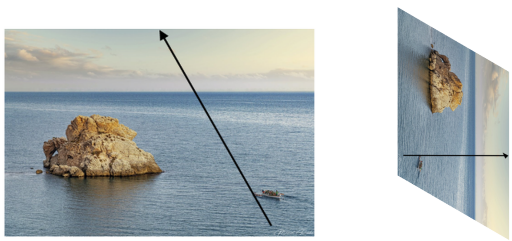


Fig. 3: Reorganización de datos con sesgo

La propuesta de este trabajo consiste en analizar y explotar los beneficios de una reorganización en memoria, pero sobre todo demostrar que, en problemas de muy elevada complejidad, la reorganización de la información mediante una interpolación sesgada de los datos es extremadamente beneficiosa.

A. Antecendentes: el algoritmo SDEM

En 2013, Tabik *et al* [1] publican un algoritmo que considera la dependencia radial en el cálculo de la cuenca visual, para crear un algoritmo que la calcula para todos los puntos de un modelo digital, en un conjunto discreto de direcciones alrededor de cada punto ($s = 360$, normalmente). Es decir, para cada sector angular de un grado, sólo tiene en cuenta los puntos en el eje central del sector, reduciendo la complejidad del problema de $O(N^{2,5})$ a $O(sN^{1,5})$

```
for pov in 1,N //Point of View loop
    for s in 1,360
        viewshed[pov]= function(s, DEM)
```

Además, mediante una simple inversión de los lazos, el código aprovecha que todos los datos están ya alineados en una determinada dirección para hacer el cálculo de la cuenca visual desde todos los puntos de la línea en una determinada dirección:

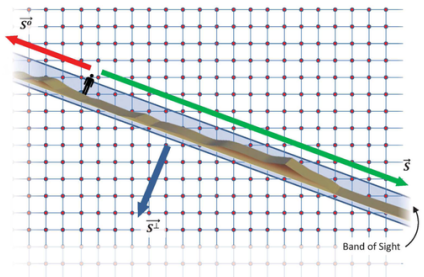


Fig. 4: Cálculo de la cuenca visual a todos los puntos (lazo interno) para un valor (s=22º) del lazo externo.

```
for s in 1,360
    for pov in 1,N
        viewshed[pov]= function(s, DEM)
```

En la siguiente figura, las imágenes de la fila interior muestran cómo, al intercambiar los lazos, se puede aprovechar para aplicar el algoritmo a todos los puntos que se encuentran alineados en la misma dirección del lazo exterior. En la fila superior, se han elegido cuatro puntos del territorio, y se ha calculado la cuenca visual en 4 direcciones alrededor de los puntos, mientras que en la fila de abajo, para cada una de las cuatro direcciones de cálculo se calcula la cuenca visual a los puntos de estudio. Se observa que algunos puntos se aprovechan del alineamiento de los datos empleados en otros cálculos.

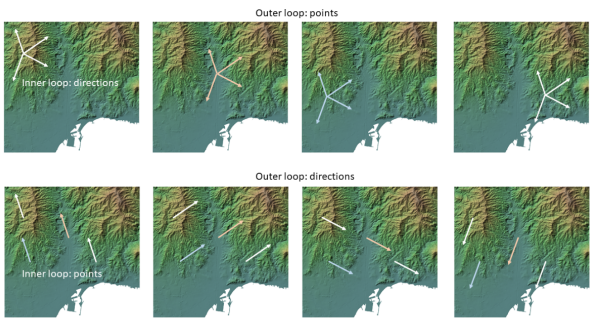


Fig. 5: Intercambio de bucles.

A.1 Almacenamiento sesgado de los datos

Recientemente, Romero *et al* publican en [2] una importante modificación del algoritmo en el que se consideran dos aspectos claves para mejorar aún más el rendimiento: 1) Si, dado un sector, sólo vamos a utilizar el resto de puntos que están en la misma línea ¿por qué no colocar los datos correspondientes en la misma línea de memoria, para aprovechar la localidad espacial? 2) Si, dado un sector, sólo vamos a utilizar el resto de puntos que están en la misma línea (no hay dependencia con otras líneas), ¿por qué no trabajar en paralelo con todas las líneas simultáneamente, usando además, GPUs?

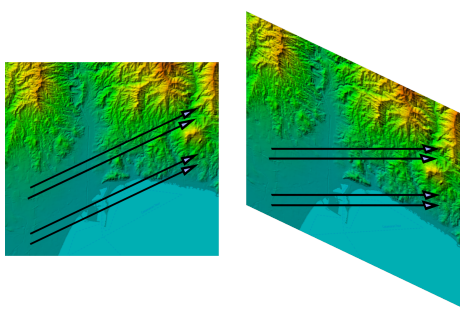


Fig. 6: Almacenamiento de un DEM con sesgo

Nace así el algoritmo sDEM (de skew-DEM, modelo sesgado de elevaciones), que se basa en la idea de colocar los datos en memoria de una forma apropiada para el cálculo en una determinada dirección, ya que el coste de "deconstruir y reconstruir el mapa" merece la pena, teniendo en cuenta la intensidad de los cálculos que se van a realizar sobre los datos en su estado "sesgado".

II. Un motor para optimizar datos en memoria

La propuesta de este trabajo consiste en generalizar esa idea y extenderla a cualquier algoritmo, mediante una plantilla que sea independiente del algoritmo. Para ello, se propone la herramienta *skewEngine*: un código empaquetado en una clase C++ que facilite la parte más tediosa de los muchos algoritmos que pueden aprovecharse de la propuesta. En concreto, el motor sería el encargado de la reorganización de los datos de mallas regulares para que estén alineados en memoria (considerando la necesaria interpolación), y que, al final del algoritmo, reubique los datos a su ubicación original, mediante una interpolación en sentido inverso. En particular, la clase esté diseñada para considerar la siguiente secuencia de etapas:

1. Lectura de la imagen o el modelo
2. Preparación de datos para cada dispositivo (CPU o GPU). Se discute en la sección III.
3. Puesta en marcha de los hilos necesarios
4. Puesta en marcha del iterador (por ejemplo, sobre 360 direcciones). Cada iteración es asignada a un dispositivo. Así, cada CPU/GPU recibirá varios sectores sobre los que va a realizar distintas operaciones.
   a) Preparación de datos (interpolación *skew*)
   b) Procesamiento en CPU o GPU
   c) Restauración de datos (interpolación *deskew*)
5. Recolección de resultados de los dispositivos.

Considerando la anterior estructura, un código sencillo, con OpenMP, que realice operaciones en 360 direcciones, tendría esta forma:

Código 1: Programa básico

```
int main(int argc, char *argv[]) {
    configure(argc, argv);
    omp_set_num_threads(nthreads);
    execute();
    deallocate();
    return 0;
    }

int execute() {
    inData_t inData=prepareData();
#pragma omp parallel default(shared)
    {
        skewer= new skewEngine();
        ...
#pragma omp for schedule(dynamic) nowait
        for (int i = 0; i < 359; i++) {
            // ... in CPU or GPU, do work
        }
#pragma omp critical
        {
            // ... reduce function
        }
        delete skewer;
    }
```

Se propone a continuación una clase (skewEngine) que sirva para cualquier tipo de datos (enteros, flotantes, double, o píxeles, por ejemplo) y que se encargue de todo lo relacionado con la preparación de los datos y la recolección de resultados.

A. La clase SkewEngine

La clase debe tener las siguientes características:

- Etapa 3: Se creará un objeto skewEngine en cada hilo, al que le corrsponde un dispositivo de cálculo. Llamaremos "motores" a dichos objetos.
- En la etapa 4.1, el motor se encarga de"sesgar" los datos de entrada
- En la etapa 4.2, una función externa aplica el algoritmo supuestamente costoso:

  ```
  skewOutput = FuncionCostosa(skewInput);
  siendo, por ejemplo:
  FuncionCostosa = cuencaVisualTotal en 1D
  FuncionCostosa = radonTransform en 1D
  FuncionCostosa = Identity
  ```

- En la etapa 4.3, el motor se encarga de reparar"los resultados sesgados.
- En la etapa 4, una sección crítica recupera la información de los motores, y los destruye.

De esta forma, el kernel del código 1 tendría la siguiente estructura:

Código 2: Kernel de ejecución

```
int execute() {
    inData_t inData=prepareData();
#pragma omp parallel default(shared)
    {
        skewEngine<float> *skewer=
            new skewEngine<float>
                (dimx, dimy,inData);
        skewer->kernel= funcionElegida;
#pragma omp for schedule(dynamic) nowait
        for (int i = 0; i < 359; i++) {
            skewer->skew(i);
            skewer->runKernel();
            skewer->deskew(i);
        }
#pragma omp critical
        {
            reduce(skewer,outData);
        }
        delete skewer;
    }
```

Obsérvese que, entre las funciones costosas elegidas. se puede implementar una función identidad, de coste cero. Dicha función tendría una triple función. En primer lugar, como depuración, ya que un volumen de datos deconstruido y reconstruido debe ser casi idéntico. En segundo lugar, sirve para determinar los errores de redondeo implicados en los procesos de interpolación. Finalmente, sirve para estimar el sobrecoste que implica la reorganización de datos, y en consecuencia, para valorar si merece la pena aprovecharse del algoritmo.

III. Implementación de *skewEngine*

Consideremos una imagen o mapa 2D (todo será extrapolable a 3D, de forma anidada, según se explica en la sección V) a la que queremos aplicar el algoritmo. En primer lugar, hay que considerar que:

- Los datos alineados se pueden procesar en doble sentido, por lo que sólo se considerarán 180^{o} direcciones en los cálculos de 1^{o} de precisión..
- Los 180 sectores se clasifican en cuatro bloques, con el objeto de hacer el algoritmo más simple

y eficiente.
Los límites de los cuatro bloques dependen del aspect-ratio de los datos de entrada. En particular, la variable $fAngle = atan(dimy/dimx)$ determina que:

```
set0=[0,fAngle[,
set1=[fAngle,90[,
set2=[90,180-fAngle[,
set3=[180-fAngle,180[
```

En la siguiente imagen se muestran, para una imagen, 4 ángulos de cada uno de los conjuntos, así como las 4 posibles situaciones a las que se debe enfrentar *skewEngine*:

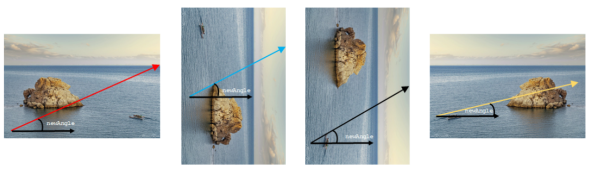


Fig. 7: sDEM.

Antes de proceder al sesgado, se preparan 4 versiones de la imagen o modelo de entrada, a las que podríamos denominar Normal-Normal, Transpose-Normal, Transpose-Mirror y Normal-Mirror, y que se corresponderán con las entradas que necesitaría el algoritmo para los ángulos de cada uno de los conjuntos de sectores. Si disponemos previamente de estas 4 versiones del modelo, el código no necesitará diferenciar la forma en la que se realiza el sesgado y el restablecimiento de los datos:

Código 3: Generación de 4 conjuntos de entrada del algoritmo

```
indata_t prepareData(T *input){
    for(int i=0;i<dimy;i++) {
    int ci=dimy-1-i;
    for (int j = 0; j < dimx; j++) {
        int cj=dimx - 1 - j;
        T val= input [dimx * i +  j];
        input1[dimy * j + ci] = val;
        input2[dimy *cj + ci] = val;
        input3[dimx * i + cj] = val;
        }
    }
    \\ return 4 arrays of type T
}
```

Con estas 4 versiones del modelo, el algoritmo no tiene que distinguir entre signos para la tangentes, ni tipos de acceso, ni está limitado a modelos cuadrados. El procesamiento es similar en los cuatro casos. Consiste, básicamente, en sesgar los datos de entrada, mediante un mecanismo simple de interpolación.

En concreto, será necesario calcular varios parámetros simples, como el ángulo de sesgo (*newAngle*), el desplazamiento que tendrá la última columna (*offset*), las dimensiones verticales y horizontales de la correspondiente versión de entrada (dim_o, dim_i) (subindexados con *o* por *outer*, *i* por *inner*) y que serán los límites de los lazos externos e interno que recorran los datos.

Finalmente, hay que tener en cuenta que al sesgar los datos de entrada (al pasar del rectángulo al romboide), cada elemento de entrada se va a ubicar en la misma columna, pero en un lugar intermedio entre dos filas del array destino. Por ese motivo, necesitamos calcular un par de vectores que sólo dependen del índice de columna: *target* y *weight*, siendo *target* el número de filas que avanza un determinado punto hacia abajo (redondeado a inferior), y *weight* un factor de ponderación, según la ubicación sesgada del punto, entre *target* y *target* + 1. En la imagen anterior se representan en rojo dichos parámetros, y que son independientes de la fila.

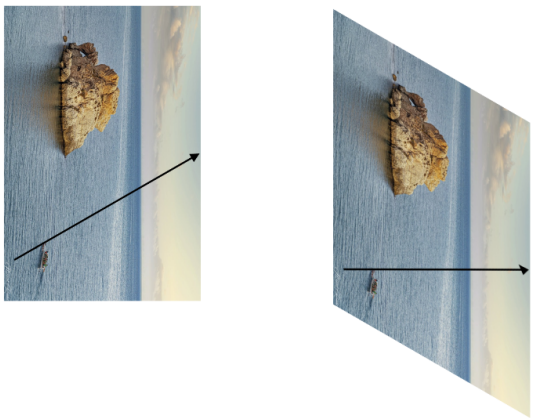


Fig. 8: Sesgado de una imagen, para el sector 113º, perteneciente al conjunto 1

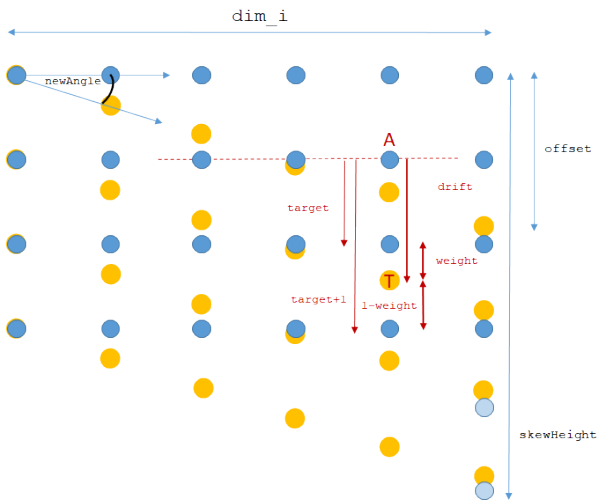


Fig. 9: Representación gráfica de los parámetros necesarios para el sesgado de un modelo

Por último, calcularemos el número de filas que se van a procesar sobre el modelo sesgado, skewHeight, así como los límites de cada fila, que dependen de *newAngle* y *offset*, y de dim_i, dim_o, y que, básicamente, definen la forma del romboide:

Código 4: Cálculo de los límites del sesgado

```
for (int i=0;i< skewHeight; i++){
    first[i]=0;
    last[i]=dim_i;
    if(i<offset)
        last[i]=(i+1)*iskewness+1;
    if(i>dim_o)
        first[i]=(i-dim_o)*iskewness+1;
    }
```

Y, finalmente, se realiza el sesgado del modelo:

Código 5: Sesgado del modelo

```
for(int i=0;i<=skewHeight*dim_i;i++)
    skewInput[i]=skewOutput[i]=0;
//isT: is transpose
//isM: is mirror
T *source=isT?(isM?input2:input1)
    :(isM?input3:input0);
for (int i = 0; i < dim_o; i++) {
    for (int j = 0; j < dim_i; j++) {
        int row = i + target[j];
        skewInput[row * dim_i + j] +=
            (1.0 - weight[j]) *
            source[dim_i * i + j];
        skewInput[(row + 1) * dim_i + j]
            += weight[j] *
            source[dim_i * i + j];
    }
}
```

A. Computación intensiva tras el sesgado

Una vez preparado los datos, el algoritmo intensivo es un simple bucle, que se puede ejecutar con paralelismo anidado (y embarazoso), ya que cada fila es independiente:

Código 6: Ejecución del kernel

```
for(int i=0;i<skewHeight;i++>)
    skewOutput[i]=
      kernel(skewInput[i],first[i],last[i]);
```

Es importante no olvidar que el espacio se ha deformado, por lo que las distancias que se apliquen en el algoritmo (si se necesitan) tienen un factor de escala 1 en vertical y $1/cos(newAngle)$ en horizontal.

B. Reconstrucción de los datos no sesgados

El proceso de interpolación de los resultados es mucho más simple, ya que no es necesario computar nuevos parámetros. Para ello, definimos un método para la reconstrucción de los datos:

Código 7: Reconstrucción de datos no sesgados

```
skewEngine::deskew()
{
    T *output;
    if(sectorType==0)output=output0;
    if(sectorType==1)output=output1;
    if(sectorType==2)output=output2;
    if(sectorType==3)output=output3;
    for(int i=0;i<dim_o;i++)
     for(int j=0;j<dim_i;j++)
      output[i*dim_i+j]+=
       (1-weight[j])*skewOutput[(i+target[j])
        *dim_i+j]+
        weight[j]*skewOutput[(i+target[j]+1)
        *dim_i+j];
}
```

C. Reducción

Finalmente, cuando una CPU o GPU termina de procesar los sectores que le han sido asignado, tiene que pasar a la fase crítica de reducción. El correspondiente thread puede tener datos de sectores asignados de más de un tipo de los 4 conjuntos, por lo que se apoya en cuatro variables booleanas. En este momento, los datos traspuestos o en espejo recuperan la ubicación original:

Código 8: Reducción para recuperar la ubicación original

```
#pragma omp critical
{
    for (int i = 0; i < dimy; i++){
        int ci=dimy-1-i;
        for (int j = 0; j < dimx; j++) {
            int cj=dimx-1-j;
            if (skewer->has0)outData[i][j]
              += skewer->output0[i*dimx+j];
            if (skewer->has1)outData[i][j]
              += skewer->output1[j*dimy+ci];
            if (skewer->has2)outData[i][j]
              += skewer->output2[cj*dimy+ci];
            if (skewer->has3)outData[i][j]
              += skewer->output3[i*dimx+cj];
        }
    }
}
```

La implementación de *skewEngine* se ha realizado en lenguaje C++, utilizando la librería OpenMP para la distribución de las direcciones de procesamiento entre los distintos núcleos. Normalmente se trabaja con 180 sectores, por lo que el grano de paralelismo es suficiente para que no haya desequilibrio de carga en ordenadores de 32 o menos núcleos. El código implementa también la transferencia opcional de los modelos sesgados a una GPU, para que el kernel de un determinado ángulo se pueda ejecutar en la misma. Dichas transferencias se han programado tanto en CUDA como en OpenCL.

También se ha implementado un kernel unitario, en el que, básicamente, el Código 6 se convierte simplemente en un lazo con la sencilla igualdad $skewOutput[i][j] = skewInput[i][j]$. Además, se han implementado diferentes casos de estudio, y en todos ellos, tan sólo se necesita definir un método de acceso de tipo void, cuyo único argumento es el objeto de la clase *skewEngine* que corresponde al sector. Lo normal es que el programador tan sólo tenga que cambiar la línea $skewer- > kernel = funcionElegida$, en el Código 2, por su propio caso de estudio. Por ejemplo, para la Cuenca Visual Total:

```
skewer->kernel=isGPU?viewshedGPU:vieshedCPU;
```

El código *skewEngine* está disponible públicamente en un repositorio Github [3].

IV. Casos de estudio

Los excelentes resultados obtenidos por el algoritmo sDEM [4] para el cálculo de la cuenca visual total fueron la motivación principal para el desarrollo de la herramienta *skewEngine*. Sin embargo, en sDEM se detectaron algunas deficiencias que se han corregido en este trabajo, y que, básicamente se centran en que sDEM no hace una preparación de los datos antes del procesamiento, sino que simultáneamente hace el trabajo de sesgado, aplica el algoritmo, y reconstruye los datos originales. Además de ser un código tremendamente complejo y poco exportable a otros casos, sDEM no diferencia previamente entre los 4 conjuntos descritos en el código 3, por lo que incluye numerosas bifurcaciones que ralentizan el código, especialmente en GPUs. La implementación de la cuenca visual total con *skewEngine* ha sido precisa-

mente (tras la obvia implementación de la identidad) el primer caso de estudio considerado. Sin embargo, para comprobar la facilidad de la implementación de otros códigos con el modelo propuesto, hemos elegido otros dos casos adicionales: el filtrado de imágenes borrosas y la transformada Radon. Por otro lado, en la Tabla I se muestran las arquitecturas utilizadas en este trabajo.

TABLA I: Arquitecturas utilizadas en los casos de estudio.

	Máquina 1	Máquina 2	Máquina 3
Nombre	Desktop–PC	Server	HPC node
CPUs	16x Intel	32x Intel	64x Intel
	i7-10700K	Xeon E5-2698 v3	Xeon E5-2698 v4
GPUs	1x NVidia	4x NVidia	8x NVidia
	Ampere RTX4080	Maxwell GTX980	Volta V100

En los siguientes apartados se muestra un breve resumen de los resultados de los casos de estudio. Sin embargo, es muy importante aislar previamente el tiempo empleado en las dos fases en las que está realmente implicada la herramienta *skewEngine*, pues es lo que servirá para identificar las aplicaciones en las que merezca la pena utilizarla:

A. Caso 1: Identidad

Para estimar el coste de las etapas *skew* y *deskew*, se ha utilizado un kernel identitario (los datos se deconstruyen y recostruyen sin cálculos intermedios), en sus versiones para CPU y GPU. Se han seleccionado 180 sectores, y se ha aplicado a dos conjuntos de datos diferentes: una fotografía RGB de 2122×2122 píxeles, y un DEM de una zona montañosa de 2000×2000 puntos. Para simplificar, se muestran solo los resultados obtenidos, en tiempo de ejecución, para la fotografía, ya que el escalado del tiempo con el tamaño es prácticamente lineal (12,5 %× más lenta la imagen que el DEM). Los resultados se muestran en la Tabla II

TABLA II: Tiempos de ejecución del kernel Identidad

	Máquina 1	Máquina 2	Máquina 3
tiempo CPUs	2.34 s.	1.3 s.	0.45 s.
tiempo GPUs	0.24 s.	0.46 s.	0.10 s.

Teniendo en cuenta la perfecta escalabilidad de la herramienta respecto al número de núcleos o GPUs, se han utilizado sólo 15, 30 y 60 núcleos de los disponibles en las respectivas máquinas, para que sean divisores del número de sectores (180) y descontar así el desequilibrio de carga. Por otra parte, el código puede elegir una CPU o GPU mediante un paradigma de granja de tareas, por lo que los mejores tiempos (siempre con GPUs) podrían reducirse si reciben la colaboración de las CPUs, aunque apenas merece la pena en ninguna de las arquitecturas. En cualquier caso, se observan tiempos muy razonables, especialmente para algoritmos que pueden tardar minutos, horas e incluso días, en conjuntos de datos de tamaños similares.

B. Caso 2: Cuenca Visual Total

Como se ha descrito en la subsección I-A, la cuenca visual total determina la superficie de un territorio que es visible por un observador, calculada para todas las posibles ubicaciones del observador. Dada una ubicación cualquiera del observador, su visibilidad se determina en un conjunto discreto de s direcciones radiantes equitativamente distribuidas. Con *skewEngine*, y dada una dirección discreta, hemos visto que fácilmente se puede calcular la cuenca visual a todos los puntos que estén en la misma línea, reutilizando todos los datos de elevación.

En trabajos previos [5], [4], [6], la mayoría de los modelos de elevación utilizados tienen tamaños de alrededor de 2500x2500 puntos. Estas dimensiones son suficientes para cubrir, por ejemplo, la superficie del Parque Nacional Sierra de las Nieves, con una resolución de 10 metros. Teniendo en cuenta que una línea de datos tiene un tamaño de (a lo sumo) 2 a 4 mil datos de elevación, y que normalmente cada dato es de sólo 2 bytes, lo normal es que toda la información de una línea quepa en la cache L1 de un núcleo. Sin embargo, por la propia naturaleza del algoritmo, el número de operaciones es de cientos de millones de FLOPs por línea, que al ejecutarse sin apenas fallos en la L1, producen unos rendimientos altísimos en todas las arquitecturas empleadas en este trabajo.

Sin entrar en detalles concretos de los resultados, cabe destacar que en el peor de los casos (el ordenador de sobremesa, utilizando sólo las CPUs) el tiempo de ejecución fue de 59 s. para el modelo de 25M de puntos del Parque Sierra de las Nieves, y mejorando un 10 % los resultados de sDEM. Sin embargo, la GPU hizo los cálculos en apenas 3.5 s, mientras que sDEM requiere 10.2 s. Hay que tener en cuenta que las herramientas de cálculo utilizadas en los Sistemas de Información Geográfica, como gdal-viewshed, o GRASS [7], [8] hacen sus operaciones en pocos segundos para el cálculo de ¡una sola cuenca visual, en lugar de 25 millones de ellas!, lo cual supone que nuestro modelo es de 6 a 7 órdenes de magnitud más rápido.

C. Caso 3: Transformada Cepstrum para filtrado de imágenes borrosas por movimiento

Pero el objeto de este estudio no es demostrar el rendimiento de una aplicación como la referenciada en la subsección anterior, sino su utilidad para implementar rápidamente versiones más eficientes de otros algoritmos. Y para ello, hemos elegido dos aplicaciones que requieren un cálculo muy intensivo. La primera de ellas es la transformada Cepstrum local, aplicada a una imagen borrosa por el movimiento. La transformada cepstrum es una técnica matemática utilizada para analizar la estructura espectral de una señal en el dominio cepstral. Se define como la transformada de Fourier del logaritmo del espectro de potencia de la señal. Su fórmula se expresa de la siguiente manera:

$$C(\tau) = \mathcal{F}^{-1}\left[\log\left(|\mathcal{F}[x(t)]|^2\right)\right]$$

Donde $x(t)$ representa la señal en el dominio del tiempo y τ es la variable de retardo. El uso de la transformada cepstrum es común en aplicaciones de procesamiento de señales y análisis espectral [9], [10]. En imágenes borrosas por movimiento, el dominio cepstral ayuda a identificar los coeficientes que identifican la dirección e intensidad del movimiento que ha provocado que la imagen esté borrosa. No se ha podido encontrar ningún experimento en la literatura que haya calculado la transformada cepstral centrada en cada uno de los píxeles de una imagen, posiblemente por su elevado coste computacional. El análisis cepstral sólo se aplica a toda la imagen, por lo que sólo se utiliza para detectar el movimiento de la cámara, y no de los diferentes objetos de la cámara, como ocurre en la imagen inferior, en la que distintos objetos se vuelven borrosos en direcciones y velocidades diferentes.

Fig. 10: Imagen borrosa por objetos con diferentes movimientos

Con *skewEngine* se ha implementado la transformada Cepstrum aplicada a cada píxel de la imagen, en 360 direcciones diferentes, y con ventanas de 32, 64 y 128 píxeles de radio. Los tiempos de ejecución oscilan entre 1 y 3 minutos para una imagen de 4Mpix, aunque lo más sorprendente es que se pudido programar en apenas unas horas, gracias a la herramienta.

D. Caso 4: Transformada Radon

la transformada de Radon es una herramienta matemática utilizada en radiología para la reconstrucción de imágenes de objetos a partir de mediciones de rayos X o de otras formas de radiación. En radiología, la transformada de Radon mide la cantidad de radiación que atraviesa el objeto desde todas las direcciones posibles y convierte esta información en una imagen bidimensional o tridimensional del objeto. Este proceso se llama tomografía y se utiliza comúnmente en medicina para visualizar estructuras internas del cuerpo humano y en otras áreas de la ciencia para la inspección de materiales o la investigación de la naturaleza de un objeto. La transformada Radon se define como la integral de la imagen a lo largo de todas las rectas que pasan por un punto fijo del espacio. En otras palabras, para cada dirección, se mide la cantidad de radiación que atraviesa el objeto a lo largo de esa dirección y se integra esta información a lo largo de todas las rectas de esa dirección.

El resultado de la transformada de Radon es una función que representa la suma de las proyecciones a lo largo de cada dirección posible. Esta función se llama sinograma y se utiliza como entrada para la reconstrucción de la imagen. La reconstrucción se realiza mediante una técnica llamada retroproyección filtrada, que consiste en tomar cada proyección del sinograma, rotarla y luego "proyectarla hacia atrás" a lo largo de la dirección correspondiente en el espacio. El resultado de este proceso, tras aplicarlo en todas las direcciones, se acumula para producir la imagen final del objeto.

La Transformada Radon Local (LRT, por sus siglas en inglés) es una variante de la Transformada de Radon que se utiliza para analizar imágenes en dominios locales y no globales. A diferencia de la transformada de Radon estándar, que utiliza la información de la proyección de la imagen en todas las direcciones posibles, la LRT utiliza una ventana de análisis local en la imagen para calcular la transformada de Radon. Esto permite analizar la imagen de forma más detallada y adaptativa a las características locales de la imagen. La LRT tiene varias aplicaciones, como el análisis de texturas en imágenes, la detección de bordes y la segmentación de imágenes. También se utiliza en campos como la inspección no destructiva de materiales, la visualización médica y la astronomía. La Transformada Radon Local (LRT) tiene un alto costo computacional en comparación con la transformada de Radon estándar. Esto se debe a que la LRT requiere el cálculo de la transformada de Radon para cada ventana local en la imagen, lo que puede ser computacionalmente intensivo, por lo que SkewEngine se presenta como la herramienta más adecuada para ello.

En este trabajo se ha implementado tanto la transformada Radon local como la general o estándar, aunque sólo presentamos datos de comparación con la última, ya que es la única de la que existe un software disponible en la bibligrafía. En particular, hemos comparado nuestros resultados con las dos aplicaciones más utilizadas y eficientes: SciKit [11] (en Python) y Astra Toolbox [12], para Matlab. En ambos casos, el código binario de los respectivos kernel está en C++, y también en CUDA, para el caso de Astra.

La imagen que hemos elegido para la trasformada Radon es una fotografía de 1500×1000 píxeles. Sin embargo, una de las aplicaciones mencionadas internamente extiende las imágenes para que estén circunscritas en un cuadrado, y éste, a su vez, en un círculo, por lo que hemos preferido crear una imagen ficticia de 2122x2122 píxeles para que la comparación sea en iguales condiciones, y a pesar de que perjudica los resultados de *skewEngine*, que trabaja con los límites ajustados. En todas las aplicaciones, la imagen resultante tras la aplicación del algoritmo ha sido idéntica (utilizando un filtro rampa), como se muestra en la Fig. 11.

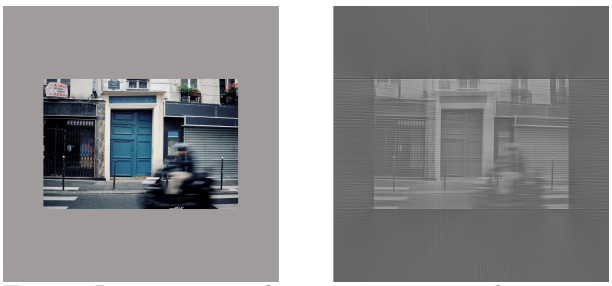


Fig. 11: Imagen original, e imagen restaurada a partir del sinograma

La siguiente tabla muestra un resumen de los resultados de la comparación, en las que se demuestra el alto rendimiento de *skewEngine*, a pesar de que no es precisamente un algoritmo especialmente costoso desde el punto de vista computacional:

TABLA III: Tiempos de ejecución de la transformada Radon

	SKE (Server)	SKE (PC)	Scikit (PC)	Astra (PC)
CPUs	1.46 s.	2.03 s.	10.58 s.	-
GPUs	0.49 s.	0.343 s.	-	0.316 s.

Como se puede observar, Astra Toolbox, que utiliza CUDA, es la que obtiene mejores resultados, y obviamente, esto se debe a que el coste de la reorganización de los datos en memoria, que en este caso supone el 70 % del coste de CPU, no merece la pena para una aplicación con tan poco coste computacional. Pero si sorprende que tan sólo se hayan necesitado apenas cuatro líneas de código para su implementación con *skewEngine*:

Código 9: Código de la transformada Radon con *skewEngine*

```
for (int i=start;i<end;i++)
    sum+=skewInput[row *dim_i+i];
for (int i=start;i<end;i++)
    skewOutput[row*dim_i+i]=sum;
```

Poniéndose así de manifiesto no sólo el rendimiento de la herramienta, sino también la altísima productividad en la programación de herramientas que se aprovechen de la misma.

V. Extensión a tres dimensiones

Si el procesamiento intensivo de datos, con una dependencia de todos con todos en cualquier dirección sobre una malla de datos bidimensional es muy costoso en dos dimensiones, su extensión a tres dimensiones puede ser excepcionalmente costoso, por lo que es evidente que cualquier forma de abordar el problema debe partir de dos premisas similares a nuestra anterior propuesta en dos dimensiones.

En primer lugar, el infinito número de direcciones en el que podríamos procesar los datos debe reducirse, de una forma similar a como ya se hizo con la discretización azimuthal en 2D con *skewEngine*, y que por defecto, dividía el espacio en 360º.

En segundo lugar, los datos deben alinearse en memoria, e incluso sería altamente recomendable un almacenamiento alineado de la información en memoria secundaria, tambien usando criterios similares a *skewEngine*. El primer problema, lo hemos resuelto utilizando un elegante y sencillo método de discretización del espacio 3D: la espiral esférica de Fibonacci: un algoritmo que obtiene una nube de puntos cuasi-equitativamente dstribuidas en la superficie de una esfera. El siguiente código muestra cómo se calculan las direcciones de procesamiento discretas (ejes de Fibonacci)

Código 10: Cálculo de los ejes esféricos de Fibonacci

```
double golden= (1+sqrt(5.0))/2;
for (int i=0; i<n; i++) {
    double the= 2*pi*i/golden;
    double phi= acos(1-2*(i+0.5)/n);
    double sphi= sin(phi);
    x= sphi * cos(the);
    y= sphi * sin(the);
    z= cos(phi);
    axes[i]= {x,y,z};
}
```

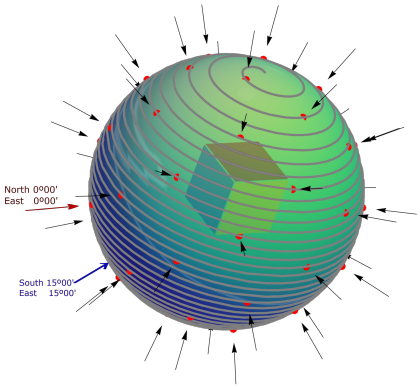
Como se observa en el código, y haciendo un simil de la esfera con nuestro planeta, las latitudes de los diferentes ejes están equitativamente distribuidos, de forma que si el número de ejes fuera $n = 180$, habría una separación exacta de un grado de latitud entre ejes (latitudes desde 0.5 a 179.5, para ser precisos). Por otra parte, a cada "latitud discreta"le corresponde una longitud exclusiva que depende de la razón áurea. Siguiendo con el simil, podría ser que a la latitud 36.5º le corresponda la longitud de Málaga, pero la anterior (35.5º), mientras que justo la anterior podría coincidir con Tokio.

Podemos aprovechar entonces la distribución equitativa de latitudes para resolver el problema del alineamiento de los datos en 3D mediante la misma reutilización del código para skewEngine en 2D. De esta forma, los n_x planos de datos (de dimensiones $n_y \times n_z$) serían reproyectados con el algoritmo *skewEngine* generando un cubo que estaría sesgado perpendicularmente a z.

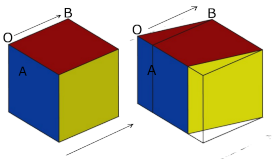
Véase con un ejemplo: Una malla tridimensional de datos, como la representada en la esfera tiene los datos alineados en memoria según la dirección marcada en rojo. Esto es, según la arista que separa su cara amarilla de la roja son datos consecutivos. A esa situación inicial, la marcamos con latitud y longitud cero.

Sin embargo, queremos aplicar un algoritmo intensivo que procese los datos en la dirección marcada por la flecha azul, de latitud -15º y longitud 15º. Para alinear la información, se realizaran dos pasos. En una de ellas se alinearía en latitud, mediante la aplicación del algoritmo *skewEngine* aplicado a cada plano Z, y en la segunda etapa, al cubo regular que contiene al cubo original sesgado, se le aplicaria el sesgo de latitud (o viceversa):

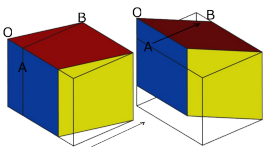
En [13] ya se utilizó la reproyección de objetos tridimensionales como una etapa de preprocesamiento en el entrenamiento de redes neuronales para el reconocimiento de moléculas. Gracias al alineamiento de la información, en ese trabajo se llegaron a generar cientos de millones de imágenes correspondientes a decenas de miles de moléculas (los componentes prin-



(a) En rojo, eje del alineamiento natural de datos. En azul, alineamiento deseado.



(b) Alineado en longitud



(c) Alineado en latitud

Fig. 12: Algoritmo *skewEngine* en 3 dimensiones aplicado a un eje de la secuencia de Fibonacci

cipales de los correspondientes fármacos) en apenas unos segundos.

VI. Conclusiones

El correcto alineamiento de datos en la memoria del computador es un factor cada vez más importante en el diseño de algoritmos eficientes que procesen datos estructurados de una forma intensiva. En el caso de mallas regulares de dos o más dimensiones, los algoritmos que realizan operaciones en una dirección concreta que sea diferente a la dirección principal de alineamiento (la que coincide con la secuencia de almacenamiento de los correspondientes datos), tendrán un patrón de acceso a los datos que resulta casi completamente aleatorio, y en consecuencia, provoca innumerables fallos de cache que pueden ser catastróficos, especialmente en el caso de grandes volúmenes de datos, como aquellos generados en un TAC o en radioastronomía.

El realineamiento de datos en la dirección correspondiente a las operaciones del algoritmo es una operación relativamente costosa si el volumen de datos es elevado, pero como su complejidad computacional es lineal y por tanto, escala bastante bien, puede merecer la pena el coste de la reorganización de la información, siempre y cuando las operaciones de los algoritmos tengan complejidades superiores. En los experimentos mostrados en este trabajo, correspondientes a tres problemas computacionales de alta y media complejidad, como son la cuenca visual total en modelos digitales de elevación, la parametrización de imágenes borrosas por movimiento, y la transformada Radon bidimensional, se ha demostrado que el re-alineamiento de la información ya merece la pena en algoritmo de media complejidad, superando incluso a los mejores resultados publicados, y produce espectaculares mejoras, de varios órdenes de magnitud, en los problemas de mayor complejidad.

Para facilitar el uso de los algoritmos de interpolación y extrapolación de las mallas en las n direcciones elegidas (que son las etapas que preceden y suceden a cualquier algoritmo que realice su operación en una dirección concreta), se ha elaborado una clase en lenguaje C++, denominada *skewEngine*, que gracias a la explotación del paralelismo embarazoso de la interpolación mediante OpenMP y OpenCL o CUDA, hace que la parte más incómoda de la propuesta de este trabajo sea simple y rápida. Además, *skewEngine* está diseñado de una forma que facilita la implementación de otros algoritmos de cálculo intensivo de datos en mallas regulares usando direcciones arbitrarias de procesamiento.

Los experimentos se han desarrollado mediante cálculos en direcciones equitativamente distribuidas en dos dimensiones (normalmente, en $n = 360$ direcciones, separadas un grado), pero también se presentan las directrices de una implementación de la herramienta *skewEngine* en 3 dimensiones, utilizando, igualmente, un conjunto equitativo de direcciones de búsqueda tridimensional que se calculan utilizando la espiral esférica de Fibonacci. De este último caso, se han publicado resultados preliminares en la que datos no estructurados de miles de moléculas, mediante interpolación numérica, se interpolan a mallas regulares que posteriormente se proyectan en imágenes, y que se han utilizado para el entrenamiento de una red neuronal. Al reconvertir y estructurar la información, se ha conseguido la generación de decenas de millones de imágenes en apenas unos segundos.

Agradecimientos

Este trabajo ha sido financiado a través del Ministerio de Ciencia y Tecnología de España con el PID2019-105396RB-I00, por la Junta de Andalucía con el proyecto UMA20-FEDERJA-127 y por la Universidad de Málaga (PIE22-099). Agradecemos asimismo al Servicio de Supercomputación y Bioinformática de la Universidad de Málaga y a la Red Española de Supercomputación por facilitar el acceso a los Supercomputadores Picasso y Loginexa.

Referencias

[1] S. Tabik, E.L Zapata, and L.F Romero, "Simultaneous computation of total viewshed on large high resolution grids," *International Journal of Geographical Information Science*, vol. 27, no. 4, pp. 804–814, 2013.

[2] Andres J. Sanchez-Fernandez, Luis F. Romero, Gerardo Bandera, and Siham Tabik, "Vpp: Visibility-based path planning heuristic for monitoring large regions of complex terrain using a uav onboard camera," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. PP, pp. 1–1, 12 2021.

[3] Felipe Romero, "Skewengine: Mesh reorganization for computing intensive applications," https://github.com/luisfromero/skewEngine, 2023.

[4] A. J. Sanchez-Fernandez, Luis F. Romero, G. Bandera, and S. Tabik, "A data relocation approach for terrain surface analysis on multi-gpu systems: a case study on the total viewshed problem," *International Journal of Geo-*

graphical Information Science, vol. 35, no. 8, pp. 1500–1520, 2021.

[5] A.R. Cervilla, S. Tabik, J. Vias, M. Merida, and L.F. Romero, "Total 3d-viewshed map: Quantifying the visible volume in digital elevation models," *Transactions in GIS*, 2016.

[6] Siham Tabik, Luis Felipe Romero, and Emilio López Zapata, "High-performance three-horizon composition algorithm for large-scale terrains," *International Journal of Geographical Information Science*, vol. 25, no. 4, pp. 541–555, 2011.

[7] GDAL/OGR contributors, *GDAL/OGR Geospatial Data Abstraction software Library*, Open Source Geospatial Foundation, 2020.

[8] QGIS Development Team, *QGIS Geographic Information System*, QGIS Association, 2023.

[9] Johann Radon, "On the determination of functions from their integral values along certain manifolds," *IEEE Transactions on Medical Imaging*, vol. 5, no. 4, pp. 170–176, 1986.

[10] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Pearson, 2010.

[11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[12] Wim Aarle, Willem Palenstijn, Jan De Beenhouwer, Thomas Altantzis, Sara Bals, Kees Batenburg, and Jan Sijbers, "The astra toolbox: A platform for advanced algorithm development in electron tomography," *Ultramicroscopy*, vol. 157, 05 2015.

[13] Felipe Romero, Luis F. Romero, and Pilar M. Ortigosa, "Reconocimiento de fármacos mediante inteligencia artificia," in *XXXII Jornadas de Paralelismo*. 2022, Jornadas Sarteco.