The Journal of Supercomputing (2024) 80:12910–12928
https://doi.org/10.1007/s11227-024-05923-2

SkewEngine: enhancing performance of intensive calculations on regular meshes

Felipe Romero[1,2] · Pilar M. Ortigosa[2] · Gerardo Bandera[1] · Luis F. Romero[1]

Accepted: 21 January 2024 / Published online: 24 February 2024

Abstract
In various applications such as hyperspectral data manipulation, MRI data exploration, or viewshed identification in digital elevation models, performing arithmetic operations on each point of a data mesh that involves other points can lead to computationally intractable problems. This paper presents SkewEngine, a tool designed to improve the performance of intensive calculations on regular 2-D data meshes, such as images, multispectral data volumes, or digital elevation models. SkewEngine addresses this problem by reorganizing the mesh in memory according to a preferred spatial direction, enabling more efficient execution of intensive calculations. It is demonstrated that SkewEngine offers significant speed improvements for various test cases, suggesting its usefulness in a broader range of applications requiring intensive data processing on regular meshes.

Keywords Hybrid computing · Regular mesh · Memory locality

✉ Luis F. Romero
felipe@uma.es

Felipe Romero
fr@uma.es

Pilar M. Ortigosa
ortigosa@ual.es

Gerardo Bandera
gbandera@uma.es

[1] Departamento de Arquitectura de Computadores, Universidad de Málaga, Avda. Cervantes, 2, 29071 Málaga, Spain

[2] Departamento de Informática, Universidad de Almería, CeIA3, C. Sacramento, s/n, 04120 Almería, Spain

1 Introduction

Challenges arise in numerous and diverse fields where a highly intensive calculation is imperative for every individual point within a 2-D or 3-D data mesh. Some noteworthy cases include computations conducted on each pixel constituting an image, every geographical point on a digital elevation model (DEM), or within the dataset acquired from a magnetic resonance imaging procedure.

In specific scenarios, the level of arithmetic intensity becomes so high that it renders the problem computationally intractable. To illustrate, let's consider the case of the viewshed of a specific point within a DEM. Contemplate a scenario where the objective entails determining the observable extent of a given territory from a designated location within that territory, as shown in Fig. 1.

To determine whether point A is visible from point B (where B is any other location within the geographical context under consideration), it becomes imperative to account for the elevation of all the points in the model, as any point a priori could potentially obstruct the line of sight. Therefore, within a DEM, with a dataset size of $N = \mathrm{dimx} \times \mathrm{dimy}$, the computational complexity of the problem, represented by big-O notation, would be of order $O(N^2)$. However, this complexity could be mitigated by exclusively considering points C closer to the A-B line as potential obstructions. Nevertheless, the complexity of the problem, $O(N^{1.5})$, remains significantly high. For instance, noteworthy applications such as Google Earth require several seconds to yield an approximate solution.

When the calculation of the viewshed is required not only for an observer situated at a specific location within a territory but also for arbitrary paths traversing the terrain, regions of interest, or even the entirety of the territory, where any point within a DEM serves as an observation point, the computational complexity scales to $O(N^{2.5})$. Even with reduced precision, months of CPU time are usually required for computations involving a typical-size model [1].

Fortunately, these types of problems belong to a category in which the parameters under examination are subject to decay based on geometric distance within the data mesh. This decay can be linear or quadratic, with the latter being more prevalent. In such scenarios, the influence of a distant point does not immediately impact, if at all, the occurrences at the opposite end of the geometry. These issues are commonly simplified through processing involving a discrete set of radial directions originating from a focal point under investigation. This approach

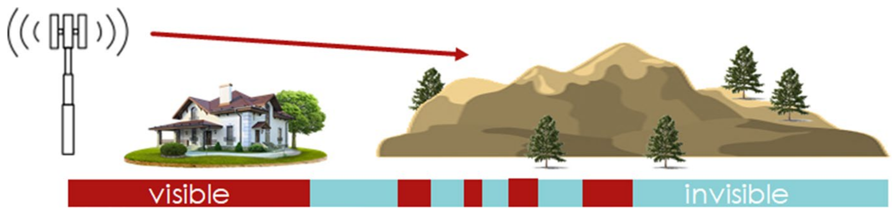


Fig. 1 Self–explaining figure of 1-D Viewshed seen from a tower

is taken because the radii are closer to each other at the point of view, thereby resulting in a finer mesh near the study point, where it is most needed.

However, in such problems where the spatial locality of information becomes a critical factor, the necessity arises to translate this spatial locality onto the data itself, explicitly concerning memory storage. Consider the following example for elucidation. Imagine the application of a filter in many lines parallel to the black arrow on the image depicted in Fig. 2 (left panel). Any algorithm of significant complexity, typically employed in computer graphics, such as the Fast Fourier Transform (FFT), would be considerably more efficient if the data array were memory-aligned, as exemplified by the distorted image on the right panel.

In this scenario, the spatial locality of data contributes significantly to optimization, allowing for more efficient computation due to the aligned memory structure. This alignment facilitates the operation of computationally demanding algorithms, ultimately enhancing the efficiency of data processing operations in spatially dependent problems.

This study's proposition involves analyzing and utilizing the advantages of a memory reorganization strategy. Primarily, the objective is to demonstrate that reorganizing information using skewed data interpolation confers significant benefits, particularly in scenarios characterized by exceptionally high levels of complexity.

Structured and unstructured mesh algorithms are a popular means of solving problems across a broad range of disciplines, from texture mapping to computational fluid dynamics, and they are often dominated by computation and memory overhead. On the one hand, enhancing memory organization for intensive calculations is a fundamental component of high-performance computing and parallel processing. On the other, optimizing memory layout entails strategically structuring data in memory to reduce the time spent accessing memory and to maximize the proximity of relevant data, leading to significant improvements in computational speed and efficiency. The work presented here is not the first that rewrites a mesh to optimize memory access. See, for example, [2–4]. And, in the geographic scope, both QGIS and CDO software offer command line tools for rotating and regridding

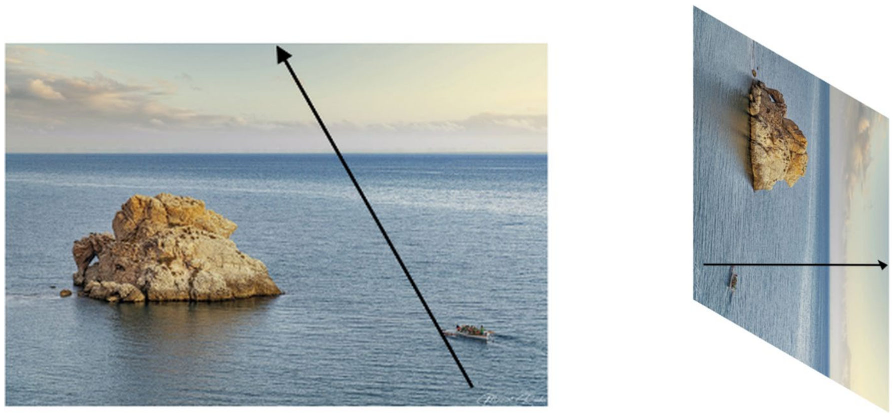


Fig. 2 Restructuring image data to a skewed array (110° case)

geographical data sets [5, 6]. However, through systematic investigation and empirical evaluation, this research seeks to establish that implementing in-memory skewed data interpolation offers substantial enhancements in computational efficiency and problem-solving efficacy. In problems marked by computational intensity, this technique proves to be highly advantageous, optimizing the utilization of memory resources and accelerating computation by exploiting the inherent spatial relationships within the data.

1.1 Background: the SDEM algorithm

In 2013, Tabik et al. [1] introduced an algorithm that accounts for radial dependency in the computation of the viewshed, culminating in the development of an algorithm capable of calculating this viewshed for all points within a DEM. This is achieved through a discrete set of directional evaluations surrounding each point (typically encompassing $s = 360$ directions). Specifically, within every one-degree angular sector, the algorithm considers solely the data points situated along the central axis of that sector.

By adopting this approach, the complexity of the original problem, originally denoted as $O(N^{2.5})$, undergoes a notable reduction to $O(s \cdot N^{1.5})$. This algorithmic optimization significantly enhances computational efficiency while preserving a significant level of accuracy in computing the viewshed across the DEM. The work by Tabik et al. is a pivotal advancement, showcasing how targeted strategies, such as radial dependency and discrete directional analysis, can yield substantial improvements in solving complex spatial problems.

Furthermore, by employing a straightforward inversion of loops that, instead of processing all the points of the DEM with an external loop and then calculating the 1-D viewshed in all sectors with the internal loop, the code leverages the pre-existing alignment of all data in a specific direction. This enables the computation of the viewshed from all points along a given line in that direction.

In Fig. 3, the images in the inner row illustrate how, by reversing the loops, it becomes possible to leverage the algorithm's application on all points aligned in the same direction as the outer loop. Four points within the territory have been selected in the upper row, and the viewshed has been computed in four directions around these points. In contrast, in the lower row, the viewshed of the study points are calculated for each of the four computation directions. It can be seen that some points benefit from the alignment of data employed in other calculations.

1.1.1 Skewed data storage

The concept of skewed data storage involves an intentional arrangement of data that capitalizes on inherent patterns or relationships within the dataset. This strategic arrangement aims to optimize computational efficiency and processing speed when performing specific operations on the data. By aligning data according to the anticipated usage patterns, re-grided (skewed) data storage reduces memory access latency and accelerates computation.

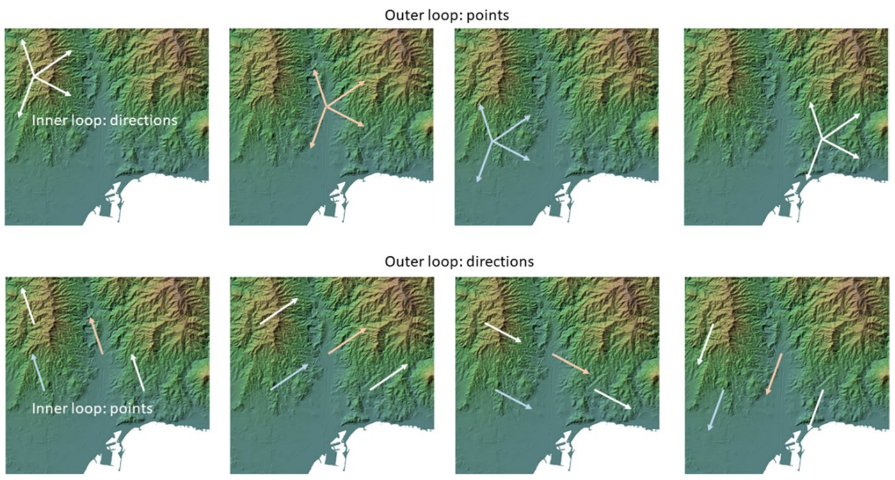


Fig. 3 Loop exchange. Above, for each point, the viewshed is calculated in all sectors. Below, for each sector, the 1-D viewshed is calculated for all points

In cases where spatial relationships or dependencies exist within the data, skewed storage can exploit these patterns to streamline operations. For instance, aligning data in memory according to the anticipated access patterns in spatial computations can significantly enhance the overall performance of algorithms. Using skewed data storage, computational tasks involving complex calculations and intricate dependencies can be expedited, making it a valuable strategy in various computational contexts.

Recently, Romero et al. introduced a significant modification to the algorithm [7], which addresses two critical aspects to enhance performance further.

1. Spatial locality exploitation: Given a specific sector, if only the remaining points within the same line will be utilized, the proposal suggests aligning the corresponding data along the same memory line. This approach capitalizes on spatial locality, minimizing memory access delays and optimizing data processing.
2. Parallelism and GPU usage: The modified algorithm exploits parallelism when considering a sector wherein only points along the same line will be utilized (thus eliminating cross-line dependencies). This involves concurrently processing all lines, benefiting from graphics processing units (GPUs) capabilities for enhanced computational efficiency.

By strategically addressing these aspects, the revised algorithm showcases a notable advancement, demonstrating the potential for substantial improvements in memory management and computational speed. Integrating spatial locality and parallel processing, coupled with GPU utilization, represents a sophisticated approach to tackling complex spatial problems with heightened efficiency and efficacy.

Thus, the sDEM algorithm (derived from skew-DEM, skewed digital elevation model) emerges, founded upon strategically arranging data in memory to suit computations along specific directions. The algorithm's core principle is underpinned by the notion that the costs associated with the "deconstruction and reconstruction of the map" are justifiable given the magnitude of computations to be executed on the inherently "skewed" data state.

The sDEM algorithm exploits spatial locality and parallel processing by adopting this approach. The effort invested in reorganizing the data structure aligns with the computational intensity of the subsequent calculations, ultimately leading to substantial gains in overall performance and enabling the exploration of new frontiers in computational geography.

2 A framework for memory data optimization

This work proposes to generalize and extend this idea to any algorithm using a template independent of the algorithm. To do this, the *skewEngine* tool is presented: a code packaged in a C++ class that facilitates the most tedious part of the many algorithms that can benefit from the proposal. Specifically, the engine would be responsible for reorganizing the data of regular meshes so that they are aligned in memory (considering the necessary interpolation) and that, at the end of the algorithm, relocates the data to its original location through an interpolation in the reverse direction. In particular, the class is designed to consider the following sequence of stages:

1. Data input (a DEM, an image,...)
2. Data allocation for each device (CPU or GPU) See Sect. 3.
3. Thread deployment and *skewEngine* objects initialization
4. Directional iteration (e.g., over 360 Directions). Each iteration is assigned to a different device. Thus, each CPU/GPU will receive several sectors to perform different operations.

   (a) Data preparation (*skew* interpolation)
   (b) Data processing using CPU and/or GPU (external function)
   (c) Data restore (*deskew* interpolation)

5. Recollection from device results.

It is also proposed that the class (skewEngine) be used for any data type (integers, floats, doubles, or pixels, for example) using C++ templates and that it handles everything related to data preparation and collection of results.

2.1 Operations of the SkewEngine class

The SkewEngine class within the proposed framework encapsulates a range of operations that collectively facilitate the efficient execution of algorithms while optimizing memory utilization and parallelism. These operations are designed to seamlessly integrate with various computational tasks, regardless of the specific algorithm employed. The class, specifically, will carry out its operations in the following stages:

- Stage 3: A skewEngine object will be created in each thread to which a computation device corresponds. These objects have been called *Engines*.
- In stage 4.a, the engine takes care of "skewing" the input data
- In stage 4.b, an external function applies the supposedly expensive algorithm:

```
skewOutput = expensiveFuntion(skewInput);
```

- In stage 4.c, the engine takes care of unskewing the results.
- In stage 5, a critical section retrieves and reduces the data from the engines.

Note that an "identity function" can be easily implemented as one of the available "expensive functions." This function would have a triple role. First, as a debug, since a deconstructed and reconstructed data volume should be almost identical. Secondly, it determines the rounding errors involved in the interpolation processes. Finally, it is used to estimate the extra cost involved in data reorganization and, consequently, to assess whether it is worth taking advantage of the algorithm.

3 SkewEngine implementation

Let's consider a 2-D image or map (everything can be extrapolated to 3-D, nested, as explained in Sect. 5) to which we want to apply the algorithm. First of all, it must be considered that:

- Aligned data can be processed in both directions, so only 180° directions will be considered in 1° precision calculations.
- The 180 sectors are classified into four sets to make the algorithm more straightforward and efficient. The boundaries of the sets depend on the aspect ratio of the input data. In particular, variable $fAngle = atan(dimy/dimx)$ determines that:

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

Figure 4 shows four angles of each of the sets.

Before proceeding to skew the data, four variations of the input image or model are prepared, which we could call Normal-Normal (*input*0), Transpose-Normal (*input*1), Transpose-Mirror (*input*2) and Normal-Mirror (*input*3), and which will correspond to the data inputs that the algorithm would need as replacement of the original data, depending on the angle. These variations are observed in Fig. 5.

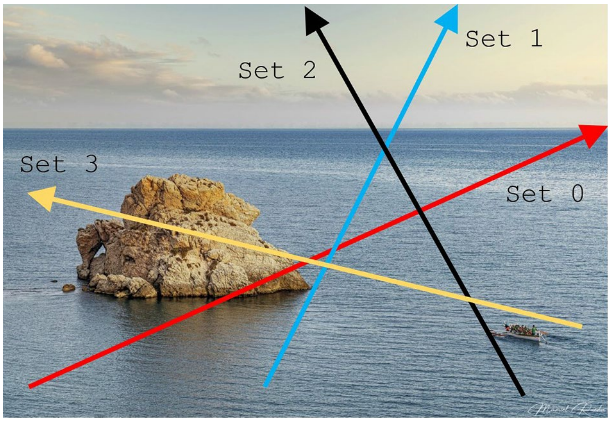


Fig. 4 Four angles corresponding to different sets

In this way, the algorithm will not have to differentiate access patterns based on the sign of the tangent, nor will it be limited to square meshes. The processing is similar in all four cases. It consists of skewing the input data through a simple interpolation mechanism, represented in Fig. 6.

Several simple parameters must be calculated, such as the skew angle (*newAngle*), the vertical and horizontal dimensions of the corresponding input version (dim_o, dim_i) (subindexed with *o* for *outer* and *i* for *inner*), the skewness ($skewness = tan(newAngle)$), and the vertical drift for the last column ($offset = dim_i * skewness$).

It must be considered that when skewing the input data (from rectangle to rhomboid), each input element will be located in the same column but in an intermediate place between two rows of the destination array. For that reason, we need to calculate a pair of vectors that only depend on the column index: *target* and *weight*, where *target* is the number of rows a given point goes down (rounded down), and *weight* a weighting factor, based on the skewed location of the point, between *target* and $target + 1$. In the previous image, these parameters are represented in red. Note that they do not depend on the row.

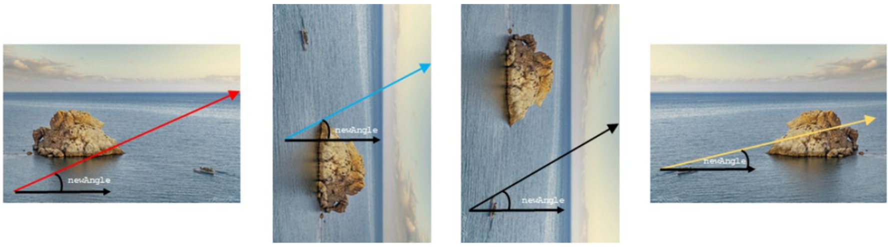


Fig. 5 Image variations corresponding to 4 angles in different sets

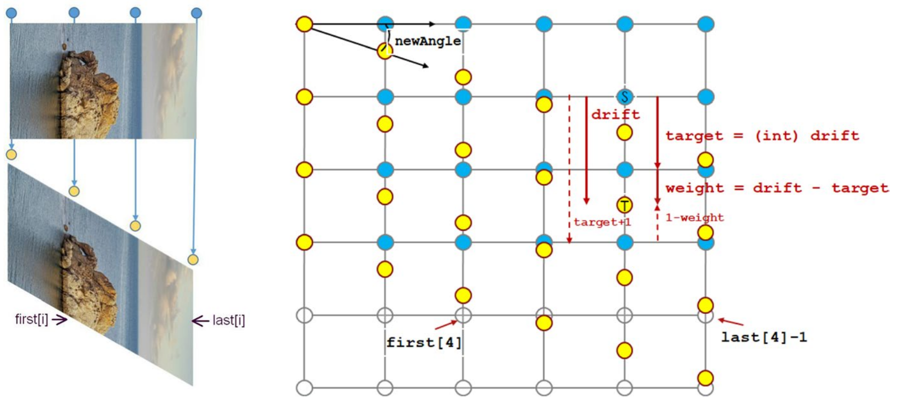


Fig. 6 Graphical representation of the parameters necessary for skewing a model

The number of rows to process on the skewed model, *skewHeight*, will be calculated, as well as the limits of each row (*first*[*i*] and *last*[*i*]), which depend on *newAngle*, *offset*, and on dim_i, dim_o (Algorithm 1). In short, these two vectors define the shape of the rhomboid, as shown in Fig. 6, on the left.

Algorithm 1 Calculation of skew limits

```
1: for $i \leftarrow 0$ to $skewHeight - 1$ do
2:     $first[i] \leftarrow 0$
3:     $last[i] \leftarrow dim_i$
4:     if $i < offset$ then
5:         $last[i] \leftarrow (i+1)/skewness + 1$
6:     end if
7:     if $i > dim_o$ then
8:         $first[i] \leftarrow (i - dim_o)/skewness + 1$
9:     end if
10: end for
```

Finally, the model is skewed (Algorithm 2).

Algorithm 2 Model skew

```
1:  for $i \leftarrow 0$ to $skewHeight \times dim_i$ do
2:      $skewInput[i]0$
3:  end for
4:  $source \leftarrow isTranspose?(isMirror?input2 : input1) : (isMirror?input3 : input0)$
5:  for $i \leftarrow 0$ to $dim_o - 1$ do
6:      for $j \leftarrow 0$ to $dim_i - 1$ do
7:          $row \leftarrow i + target[j]$
8:          $skewInput[row \times dim_i + j] \leftarrow (1.0 - weight[j]) \times source[dim_i \times i + j]$
9:          $skewInput[(row + 1) \times dim_i + j] \leftarrow weight[j] \times source[dim_i \times i + j]$
10:     end for
11: end for
```

3.1 Intensive computations onto skewed data

Once the data are prepared, the intensive algorithm is a simple loop, which can be executed with nested (and embarrassing) parallelism since each row is independent, as shown in Algorithm 3.

Algorithm 3 Kernel execution

```
1:  for $i \leftarrow 0$ to $skewHeight - 1$ do
2:      $skewOutput[i] \leftarrow \mathrm{kernel}(skewInput[i], first[i], last[i])$
3:  end for
```

It must be observed that the space has been warped, so the distances applied in the algorithm (if needed) have a scale factor of 1 vertically and 1/*cos*(*newAngle*) horizontally. Consequently, the interpolation of *skewEngine* is bilinear.

3.2 Reduction

Finally, when a CPU or GPU finishes processing the sectors that have been allocated to it, the process of interpolating the results back to the unskewed location is much simpler since it is unnecessary to compute new parameters. After that, the algorithm goes on to a critical reduction phase. The corresponding thread can have allocated sector data of more than one type of the four sets, so it relies on four boolean variables. At this point, the mirrored or transposed data returns to the original location.

The implementation of *skewEngine* has been done in C++, using the OpenMP library to distribute processing threads between the different cores. Typically, 180 sectors are used, so the degree of parallelism is sufficient so that there is hardly any load imbalance in computers with 16 cores or less. The code also implements the optional transfer of skewed models to a GPU so that the kernel of a given angle can run on it. For GPUs, *skewEngine* has been written in both CUDA and OpenCL.

A unitary kernel has also been implemented, where the kernel becomes just a loop with the simple equality $skewOutput[i][j] = skewInput[i][j]$. In addition, different study cases have been implemented. In all of them, it is only necessary to define a void type method, whose only argument is the object of the *skewEngine* class corresponding to the sector. The usual case is that the programmer only has to change a line $skewer- > kernel = chosenFunction$ for his case study. For example, for the Total Viewshed:

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

The *skewEngine* code is publicly available in a Github repository [8].

4 Case studies

The excellent results obtained by the sDEM [9] algorithm for calculating the total viewshed were the primary motivation for developing the *skewEngine* tool. However, in sDEM, some deficiencies were detected that have been corrected in this work, which focuses on the fact that sDEM does not prepare the data before processing but simultaneously does the skewing job, applies the algorithm, and rebuilds the original data. In addition to being tremendously complex code and hardly exportable to other cases, sDEM has yet to differentiate between the four sets described above. Hence, it includes numerous branches that slow down the code, especially on GPUs. The implementation of the total viewshed algorithm with *skewEngine* has been precisely (following the straightforward implementation of the identity) the first case study considered. However, to check the ease of implementing other codes with the proposed model, we have chosen two additional cases for image filtering: Motion blurred Cepstrum transform and Radon transform. Table 1 shows the architectures used in this work.

The following sections provide a summary of the results of the case studies. However, it is imperative to previously isolate the time spent in the two phases in which the *skewEngine* tool is involved since it is what will serve to identify the applications in which it is worth using it.

Table 1 Computer architectures used in the case studies

Machine	Laptop-PC	Desktop–PC	Server	HPC node
CPUs	6xP 4xE Intel	8x Intel	32x Intel	64x Intel
	i7-13620H	i7-10700K	Xeon E5-2698 v3	Xeon E5-2698 v4
GPUs	1x NVidia	1x NVidia	4x NVidia	8x NVidia
	Ampere RTX4050	Ampere RTX4080	Maxwell GTX980	Volta V100

4.1 Case 1: identity

To estimate the cost of the *skew* and *deskew* stages, an identity kernel has been used (the data is deconstructed and reconstructed without intermediate calculations) in its versions for CPU and GPU. One hundred eighty sectors have been selected and applied to two different data sets: an RGB picture of 2122×2122 pixels and a DEM of a mountainous area of 2000×2000 points. For simplicity, only the results obtained, at runtime, for the picture are shown since the scaling of time with size is practically linear (The DEM is 12.5% faster than the image). The results are displayed in Table 2.

In these results, only 10, 15, 30, and 60 cores of those available in the respective machines have been used to be divisors of the number of sectors (180) and thus discount the load imbalance. All available GPUs have been used. Moreover, the code can choose a CPU or GPU using a task farm paradigm, so the best times (always with GPUs) could be reduced if they receive the collaboration of the CPUs, although it is hardly worth it in any of the architectures. In any case, very reasonable times are observed on data sets of similar sizes, especially if *skewEngine* is going to be applied to algorithms that can take minutes, hours, and even days.

4.2 Case 2: total viewshed

As described in Sect. 1.1, the total viewshed determines the surface of a territory that is visible by an observer, calculated for all possible locations of the observer. Given any observer's location, his visibility is determined in a discrete set of equally distributed *s* radiating directions. With *skewEngine*, and given a discrete direction, it is easy to calculate the viewshed to all points on the same line, reusing all the elevation data.

In previous works [9–11], most of the elevation models used have sizes around 2500×2500 points. These dimensions are enough to cover, for example, the surface of a 200 km^2 National Park, with a resolution of 10 ms. Considering that a line of data has a size of (at most) 2–4 thousand elevation data and that usually each data is only 2 bytes, it is expected that all the information of a line fits in a core's L1 cache. However, due to the very nature of the algorithm, the number of operations is hundreds of millions of FLOPs per line, which, when executed with hardly any L1 cache misses, produces very high performance in all the architectures used in this work.

Table 2 Elapsed time for the identity kernel

Machine	Laptop-PC	Desktop–PC	Xeon Server	HPC node
CPUs (OpenMP)	1.45 s.	1.83 s.	2.86 s.	1.10 s.
GPUs (CUDA)	0.31 s.	0.15 s.	0.89 s.	0.20 s.
GPUs (OpenCL)	3.22 s.	1.09 s.	1.21 s.	0.28 s.

Without going into specific details of the results, it should be noted that in the worst case (desktop computer, using only the CPUs), the execution time was 59 s. for the 25 M point model of Sierra de las Nieves Park in Spain and improving the sDEM results by 10%. However, the GPU did the calculations in just 3.5 s, while sDEM requires 10.2 s. It should be taken into account that the calculation tools used in Geographic Information Systems, such as gdal-viewshed, or GRASS [6, 12] do their operations in a few seconds for the calculation of a single viewshed, instead of 25 million of them, which means that our model is 6–7 orders of magnitude faster.

4.3 Case 3: Cepstrum transform for motion blur filtering

The purpose of this study is not to demonstrate the performance of an application like the one referenced in the previous subsection but its usefulness in quickly implementing more efficient versions of other algorithms. For that, we have chosen two applications that require very intensive calculations. The first is the local Cepstrum transform, applied to a motion-blurred image. The Cepstrum transform is a mathematical technique used to analyze the spectral structure of a signal in the cepstral domain [13, 14]. It is defined as the Fourier transform of the logarithm of the signal's power spectrum. Its formula is expressed as follows:

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

where $x(t)$ represents the signal in the time domain, and τ is the delay variable. In motion-blurred images, the cepstral domain aids in identifying coefficients that characterize the direction and intensity of the motion responsible for the image blur. No experiments have been identified in the literature that calculate the pixel-centered cepstral transform for each pixel within an image, likely due to its computationally intensive nature. Cepstral analysis is typically applied to the entire image, focusing on detecting camera motion rather than the distinct objects within the scene. This limitation is evident in the image in Fig. 7, where various objects become blurred in different directions and at varying speeds.

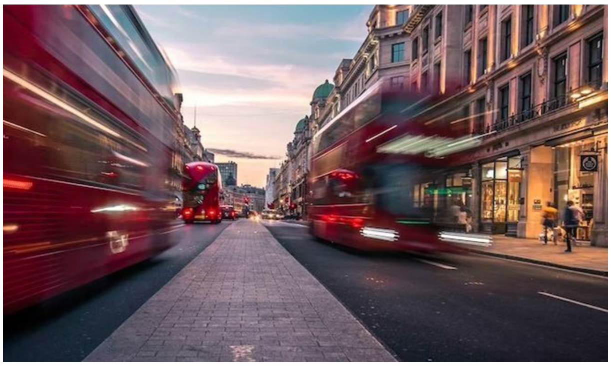


Fig. 7 Blurred image by objects with different movements

Using the *skewEngine* framework, the Cepstrum transform has been implemented to operate on each image pixel across 360 different directions, employing window sizes of 32, 64, and 128 pixels in radius. The execution times vary between 1 and 3 min for a 4-megapixel image. However, the most noteworthy aspect is that this implementation has been achieved in just a few hours, attributed mainly to the efficiency and facilitation provided by the framework.

4.4 Case 4: Radon transform

The Radon transform is a mathematical tool used in radiology for reconstructing images of objects from measurements of X-rays or other forms of radiation. The Radon transform measures the amount of radiation passing through the object from a discrete set of directions. It converts this information into a two-dimensional or three-dimensional image of the object. This process is called tomography and is commonly used in medicine to visualize the internal structures of the human body and in other areas of science for the inspection of materials or the investigation of the nature of an object. Mathematically, the Radon transform is defined as the integral of the image along all the lines that pass through a fixed point in space. In other words, the amount of radiation passing through the object along that direction is measured for each direction, and this information is integrated along all the lines in that direction.

The result of the Radon transform is a function that represents the sum of the projections along each possible direction. This function is called a sinogram and is used as input for the reconstruction of the image. The reconstruction is done using filtered back-projection, which involves taking each projection of the sinogram, rotating it, and then "projecting it back" along the corresponding direction in space. After applying it in all directions, the result of this process is accumulated to produce the reconstructed image of the object.

The Local Radon Transform (LRT) is a variant of the Radon Transform used to analyze images in local domains. Unlike the standard Radon transform, which uses information from the image projection in all possible directions, LRT uses a local analysis window on the image to calculate the Radon transform. This allows analysis of the image in a more detailed and adaptive way to the local characteristics of the picture. LRT has several applications, including image texture analysis, edge detection, and image segmentation. It is also used in fields such as nondestructive inspection of materials, medical visualization, and astronomy. The Local Radon Transform (LRT) has a higher computational cost than the standard Radon transform. This is because the LRT requires the calculation of the Radon transform for each local window in the image, which can be computationally intensive, so SkewEngine is the most suitable tool for it.

In this work, both the local and the general or standard Radon transform have been implemented. However, only data for comparison is presented for the standard case since it is the only software available in the literature. In particular, we have compared our results with the two most used and efficient applications:

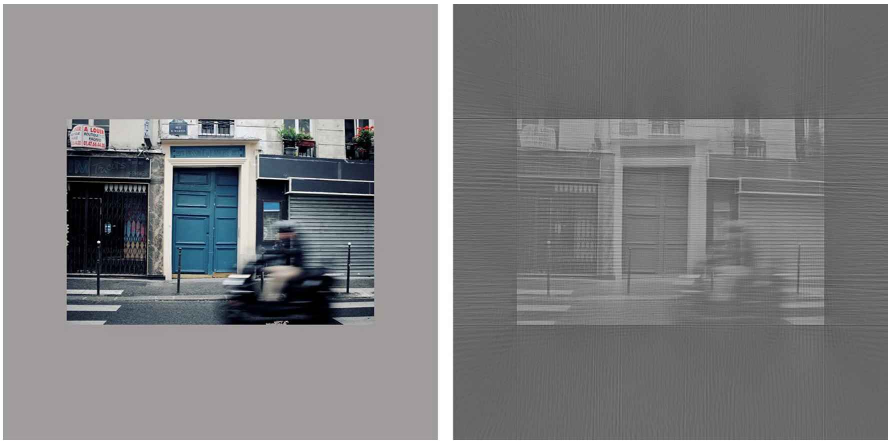


Fig. 8 Original image, and image restored from the sinogram

Table 3 Elapsed time for the Radon Transform

	SKE (Xeon Server)	SKE (Desktop-PC)	Scikit (Desktop-PC)	Astra (Desktop-PC)
CPUs	1.46 s.	2.03 s.	10.58 s.	–
GPUs (CUDA)	0.49 s.	0.317 s.	–	0.316 s.

SciKit [15] (in Python) and Astra Toolbox [16], for Matlab. In both cases, the binary code of the respective kernel is in C++ and also in CUDA in the case of Astra.

The image we have chosen for the Radon transform is a photograph of 1500×1000 pixels. However, one of the applications mentioned internally stretches the images so that they are circumscribed in a circle, and this, in turn, in a square, so we have preferred to create a dummy image of 2122×2122 pixels so that the comparison is on equal conditions, and although it hurts the results of *skewEngine*, which can do the job without using this halo. In all applications, the resulting image after applying the algorithm has been identical (using a ramp filter) and is shown in Fig. 8.

Table 3 summarizes the comparison results, demonstrating the high performance of *skewEngine*, even though it is not a particularly computationally expensive algorithm.

As can be seen, Astra Toolbox, which uses CUDA, is the one that obtains the best results. This is due to the cost of reorganizing the data in memory, which in this case represents almost 70% of the total cost; it is not worth it for an application with so little computational cost. Furthermore, due to the halo, *skewEngine* had to multiply the number of pixels by two to make this comparison, even though the algorithm does not need it. But the most surprising thing is that only four lines of code were required for its implementation with *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;
```

Thus, the tool's performance and the very high productivity in programming tools that take advantage of it are revealed.

5 A 3-D extension for *skewEngine*

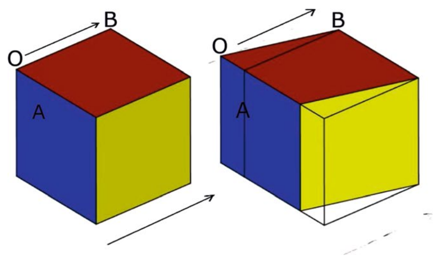
Data processing, which is already expensive in two dimensions due to the interconnection of all data points, becomes even more challenging in three dimensions. Therefore, any solution to this problem must be based on the same premises as our previous proposal for two dimensions. First, we need to reduce the infinite number of directions for data processing, similar to how it was addressed in the 2-D azimuthal discretization using *skewEngine*, which divided the space by default into 360°. For instance, the elegant Fibonacci spiral provides a potential equitable distribution of axes in 3-D. Secondly, the data volume could be realigned in memory through a two-phase skewing and interpolation process using the *skewEngine* tool. Intuitively, if we describe any of the three-dimensional axes in terms of longitude and latitude, the first skewing phase would align the information with zero longitude. In the second phase, each of the previously skewed planes would be aligned with zero latitude (Fig. 9).

6 Conclusions

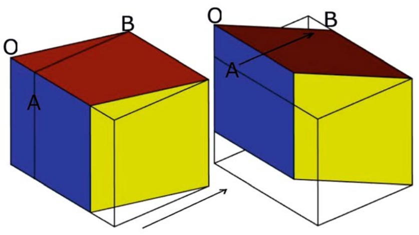
The correct alignment of data in computer memory is an increasingly important factor in designing efficient algorithms that process structured data intensively. In the case of regular meshes of two or more dimensions, the algorithms that perform operations in a specific direction that is different from the central alignment direction (the one that coincides with the storage sequence of the corresponding data) will have an access pattern to the data that is almost entirely random, and consequently, causes innumerable cache misses that can be catastrophic, especially in the case of large volumes of data, such as those generated in a computed tomography scan or radio astronomy.

Realigning data in the direction corresponding to algorithm operations is a relatively expensive operation if the volume of data is large. Still, since its computational complexity is linear and scales quite well, it may be worth the cost of doing so. The reorganization of information, as long as the operations of the algorithms have higher complexities.

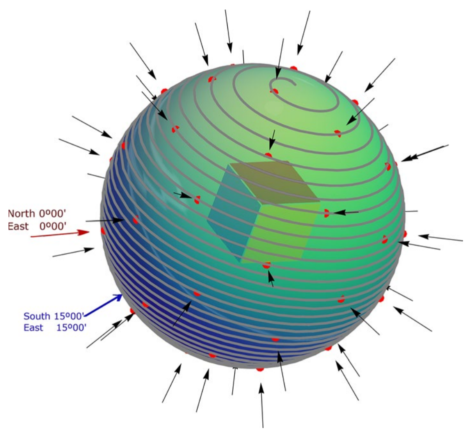
In the experiments shown in this work, corresponding to three computational problems of high and medium complexity, such as the total viewshed in digital elevation models, the parameterization of motion blur images, and the two-dimensional



(a) Alignment in longitude



(b) Alignment in latitude



(c)

Fig. 9 A 3-D skewing process, using a multi-plane *skewEngine* in two stages

Radon transform, it has been shown that the information realignment is already worthwhile in algorithms of medium complexity, surpassing even the best-published results, and produces spectacular improvements, of several orders of magnitude, in the most complex problems.

To facilitate the use of the mesh interpolation and extrapolation algorithms in the n chosen directions (which are the stages that precede and follow any algorithm that performs its operation in a specific direction), a C++ class named *skewEngine* has been developed by leveraging the inherent parallelism of interpolation through OpenMP, OpenCL and CUDA. *skewEngine* simplifies and accelerates the most complex aspect of this paper's case studies. Moreover, *skewEngine* is designed to facilitate the implementation of other data-intensive computational algorithms on regular meshes using arbitrary processing directions.

All the experiments have been conducted using computations in equally distributed directions in two dimensions (usually in 360 directions and $n = 180$ axes). As data processing on each axe is entirely independent, it is possible to exploit parallelism up to that number using multithreading and minimal effort.

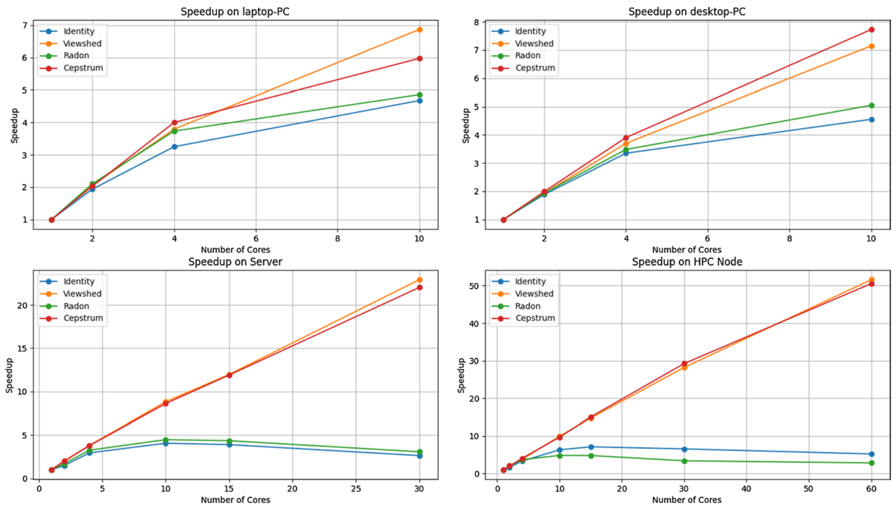


Fig. 10 Speedup of *skewEngine* for the 4 case studies on the 4 computers in Table 1

The speedup results, shown in Fig. 10, clearly demonstrate this, being particularly evident for the two applications where computational intensity is high (Viewshed and Cepstrum). Additionally, except for the HPC node, the computers were not used exclusively during the tests. In the two applications with minimal computational load (Identity and Radon), it becomes evident that the system time spent on creating and destroying OpenMP threads, which grows linearly, eventually dominates and penalizes the speedup, as predicted by Amdahl's law. However, these data only reinforce the idea that *skewEngine* requires a threshold beyond which it is worth using.

Finally, guidelines for a 3-dimensional implementation of the *skewEngine* tool are also presented, using a fair set of three-dimensional search directions computed using the Fibonacci spherical spiral. In this last case, preliminary results have been published [17] in which unstructured data of thousands of molecules, using numerical interpolation, are interpolated to regular meshes that are later projected into images, and which have been used to train a neural network. By reconverting and structuring the information, tens of millions of images have been generated in just a few seconds.

Author contributions All authors contributed equally to the design of the work and the organization of the paper. Felipe Romero was responsible for the majority of the programming work.

Funding Funding for open access publishing: Universidad Málaga/CBUA. Funding for open access charge: Universidad de Málaga/CBUA. This work has been financed by the Spanish Ministry of Science and Technology through the National Plan project PID2022-136575OB-I00, by the Andalusian Government and FEDER funds through the UMA20-FEDERJA-127 project, and by the University of Malaga (PIE22-099). We also thank the Supercomputing and Bioinformatics Service of the University of Malaga and the Spanish Supercomputing Network for facilitating access to the Picasso and Loginexa Supercomputers.

Declarations

Conflict of interest The authors have no competing interests as defined by Springer or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval Not applicable.

References

1. Tabik S, Zapata EL, Romero LF (2013) Simultaneous computation of total viewshed on large high resolution grids. Int J Geogr Inf Sci 27(4):804–814. https://doi.org/10.1080/13658816.2012.677538
2. Park N (2002) Improving memory hierarchy performance using data reorganization. PhD thesis, USA. AAI3093966
3. Voutchkov I, Keane A, Shahpar S, Bates R (2018) (Re-) meshing using interpolative mapping and control point optimization. J Comput Des Eng 5(3):305–318. https://doi.org/10.1016/j.jcde.2017.12.003
4. Iryanto S, Muttaqien FH, Sadikin R (2020) Irregular grid interpolation using radial basis function for large cylindrical volume. J Comput Sci Inf 13(1):17–23
5. Schulzweida U (2022) CDO user guide. Zenodo. https://doi.org/10.5281/zenodo.7112925
6. QGIS Development Team (2023) QGIS geographic information system. QGIS Association. https://www.qgis.org
7. Sanchez-Fernandez AJ, Romero LF, Bandera G, Tabik S (2021) VPP: visibility-based path planning heuristic for monitoring large regions of complex terrain using a UAV onboard camera. IEEE J Select Top Appl Earth Obs Remote Sens. https://doi.org/10.1109/JSTARS.2021.3134948
8. Romero F (2023) SkewEngine: mesh reorganization for computing intensive applications. GitHub. https://github.com/luisfromero/skewEngine
9. Sanchez-Fernandez AJ, Romero LF, Bandera G, Tabik S (2021) A data relocation approach for terrain surface analysis on multi-GPU systems: a case study on the total viewshed problem. Int J Geogr Inf Sci 35(8):1500–1520. https://doi.org/10.1080/13658816.2020.1844207
10. Cervilla AR, Tabik S, Vias J, Merida M, Romero LF (2016) Total 3D-viewshed map: quantifying the visible volume in digital elevation models. Trans GIS. https://doi.org/10.1111/tgis.12216
11. Tabik S, Romero LF, Zapata EL (2011) High-performance three-horizon composition algorithm for large-scale terrains. Int J Geogr Inf Sci 25(4):541–555
12. GDAL/OGR contributors (2020) GDAL/OGR geospatial data abstraction software library. Open Source Geospatial Foundation. Open Source Geospatial Foundation. https://gdal.org
13. Radon J (1986) On the determination of functions from their integral values along certain manifolds. IEEE Trans Med Imaging 5(4):170–176. https://doi.org/10.1109/TMI.1986.4307775
14. Oppenheim AV, Schafer RW (2014) Discrete-time signal processing, 3rd edn. Pearson, London
15. Pedregosa F et al (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
16. Aarle W, Palenstijn W, De Beenhouwer J, Altantzis T, Bals S, Batenburg K, Sijbers J (2015) The ASTRA toolbox: a platform for advanced algorithm development in electron tomography. Ultramicroscopy 157:35–47. https://doi.org/10.1016/j.ultramic.2015.05.002
17. Romero F, Romero LF, Ortigosa PM (2022) Reconocimiento de fármacos mediante inteligencia artificial. In: XXXII Jornadas Sarteco, Alicante, Spain