International Journal of Geo-Information

MDPI

Article

A Lightweight CUDA-Based Parallel Map Reprojection Method for Raster Datasets of Continental to Global Extent

Jing Li [1,*], Michael P. Finn [2] and Marta Blanco Castano [1]

[1] Department of Geography and the Environment, University of Denver, Denver, CO 80208, USA; marta.blancocastano@yahoo.com

[2] U.S. Geological Survey, Center of Excellence for Geospatial Information Science, Denver, CO 80225, USA; mfinn@usgs.gov

* Correspondence: jing.li145@du.edu; Tel.: +1-303-871-4687

Academic Editor: Wolfgang Kainz

Received: 2 February 2017; Accepted: 19 March 2017; Published: 23 March 2017

Abstract: Geospatial transformations in the form of reprojection calculations for large datasets can be computationally intensive; as such, finding better, less expensive ways of achieving these computations is desired. In this paper, we report our efforts in developing a Compute Unified Device Architecture (CUDA)-based parallel algorithm to perform map reprojections for raster datasets on personal computers using Graphics Processing Units (GPUs). This algorithm has two unique features: a) an output-space-based parallel processing strategy to handle transformations more rigorously, and b) a chunk-based data decomposition method for projected space in conjunction with an on-the-fly data retrieval mechanism to avoid memory overflow. To demonstrate the performance of our CUDA-based map reprojection approaches, we have conducted tests between this method and the traditional serial version using the Central Processing Unit (CPU). The results show that speedup ratios range from 10 times to 100 times in all test scenarios. The lessons learned from the tests are summarized.

Keywords: CUDA; parallel processing; raster map reprojection; raster datasets; high performance computing; geospatial data

1. Introduction

Different map projections are useful for different purposes and applications (e.g., spatial analysis, data visualization), and as such, certain projections that minimize distortions of one or more aspects (e.g., area, angle) are often necessary. Map reprojection is the process of applying different mathematical algorithms to convert geospatial datasets with existing projections from one projection to another (while we concede that the definition of the terms "projection" and "reprojection" are not unambiguous, we use, in this work, the following general definition: A projection goes from geodetic/geographic coordinates to 2D Euclidean (planar) coordinates; a reprojection changes between different Euclidean coordinates). When performing reprojections for large geospatial datasets, complex algorithms are applied to ensure data quality. As a result, computational challenges may rise when the data resolution is high, especially for projection or reprojection of raster datasets of continental to global extent (i.e., when the cartographic scale is small). To solve these types of computationally intensive problems, previous approaches usually relied on the use of high-end workstations, thus making efficient map reprojections for raster datasets less accessible to scientists who do not possess expensive infrastructural resources [1] such as high-end computer clusters or supercomputers. Since the last decade or so, NVIDIA's Compute Unified Device Architecture (CUDA) platform was introduced as

a lightweight and affordable alternative to the traditional parallel computing paradigm based on high-end supercomputers [2]. This architecture removes previous Information Technology barriers by allowing scientists to perform parallel processing to accelerate computational-intensive problems with any computing devices (e.g., desktop machines) equipped with NVIDIA Graphics Processing Units (GPUs) [3].

In this paper, we report our efforts to develop a CUDA-based parallel algorithm to reproject raster datasets with personal computers. Our algorithm is based on the rigorous map reprojection method that uses the four-corner technique for calculating pixel values and further utilizes both forward and inverse projection processes [4]. This method yields more accurate results, especially for images of global coverage even though it is more compute-intensive compared to map reprojection methods based on approximation [4]. Our algorithm introduces two key strategies for handling the data. One involves implementing an output space-based decomposition strategy so that map reprojection can be carried out in a parallel manner. The other is the usage of chunks, which are subdivisions of a dataset used for parallel processing decomposition. Chunks are used to partition large output raster images by retrieving the pixels in the input images within a spatial extent defined by the bounding pixels of those chunks [5,6]. This is particularly beneficial when processes are executed with personal computers, as large datasets could not otherwise be processed given the limited memory of computing devices.

Besides the methodological design and implementation, we also conducted a series of tests to demonstrate the superior performance of GPU over CPU processes as well as the different data decomposition strategies with GPU components. These assessments allowed us to explore the best practices of utilizing our CUDA algorithm, thus providing guidance for: (a) fine tuning the parallel implementation algorithm and (b) providing information to design algorithms in various computer architectures, including supercomputers and cloud computing.

The rest of the paper is organized as follows: Section 2 discusses the status and benefits of GPU-based geospatial applications, as well as the challenges of implementing map reprojections for raster datasets. Section 3 describes the parallel design and implementation of map reprojections with CUDA, as opposed to the traditional serial methods (CPU). Section 4 summarizes the experiment methods, variables, and results using both CPU and GPU assessments. The conclusions and future work suggestions are provided in Section 5.

2. Literature Review

In recent years, map reprojection methods have been available in most commercial and open source geographic information system (GIS) software packages, which usually rely on approximations for projection equations (with a few exceptions [7–9]) to enable relatively faster processes. With the explosion of very large, high-resolution geospatial data, problems with map reprojections remain significant even in expensive enterprise GIS software. High-resolution raster datasets often contain accuracy issues directly attributable to projections from the equal-angular grids in geographic coordinates to a plane coordinate system [9,10]. Research has shown [9–12] that there are errors associated with various spatial data resolutions as well as particular map projection selections and subsequent resampling. Therefore, various attempts to more precisely handle these problems and minimize accuracy errors (spatially and categorically) have been developed within geospatial data processing programs [11,12]. Computational intensity, additionally, often becomes a bottleneck when those methods are used, due to the complexity of algorithms and large volume of datasets [12–14].

Research into large geospatial data handling and processing problems frequently focuses on topics related to the computational aspect of cyberinfrastructure (CI) [15], along various lines of inquiry such as high-performance computing (HPC), big data, data-intensive computing, data analytics, cloud computing, and geocomputation [3,16–18]. To tackle the computational issues, parallel computing with its associated computer architecture and programming methods has been more commonly used

since the 1990s [19,20]. pRasterblaster, a processing software program for performing global raster reprojections using supercomputers, has also been investigated in cyberinfrastructure studies [21].

However, supercomputers require large monetary and infrastructural resources, and are thus far less accessible to everyday scientists. A more affordable solution to powerful processing involves GPU-based parallel implementations using CUDA. This CUDA environment and computing programming model utilizes NVIDIA Corporation's GPUs for general purposes [2]. CUDA parallel implementations consist of a two-level decomposition model made up of units including grids, blocks and threads. Threads are the smallest programmable units capable of handling tasks in a GPU and are contained in blocks, which in turn make up a grid in a GPU. A thread is a portion of a process and is the smallest execution unit in a CUDA program [2]. For example, performing map reprojection on a raster image is a process. Performing map reprojection for a subset of a raster image can be considered as the thread of that process. A CUDA-enabled GPU has multiple processors that allow multiple threads to run concurrently to accelerate computing speed. A block includes a group of threads and is allocated with a certain amount of shared memory to store data associated with its threads. Threads in a same block can cooperate with each other and access the shared memory. Depending on the hardware configurations of devices, the amount of shared memory and the number of maximum threads of a block can vary. A grid is equivalent to the entire process, which consists of one or multiple blocks. When designing a CUDA program, developers should configure the grid dimension (i.e., the number of blocks of a grid) and the block dimension (i.e., the number of threads in a block). This configuration is termed a two-level block-thread configuration [2]. With CUDA, any personal computers equipped with NVIDIA GPUs can be used to perform parallel processing, therefore proving a cheaper option compared to supercomputing with CPU clusters.

CUDA-based parallel processing techniques have been widely utilized in geospatial applications. Examples include Li et al. [22], Lukac and Zalik [23], and Ortega and Rueda [24]. CUDA has been utilized to support map projections as well. Tang and Feng [25] report an effort of designing a CUDA-based parallel algorithm for projecting vector data. When designing and implementing a CUDA-based map reprojection method for raster datasets, three issues of GPU parallel processing should be addressed. First, the GPU memory is limited given that GPUs have smaller memory capabilities compared to main memory in personal computers. Therefore, large datasets cannot be directly processed with a single GPU. A preprocessing step is usually conducted with the CPU to prepare datasets, so they can be handled without introducing memory overflow. Secondly, because CUDA employs a two-level block-thread configuration, the performance varies significantly when setting different block and thread dimensions. Fine-tuning of block and thread configuration to best utilize GPU parallel processing capabilities is thus recommended [26]. Lastly, a parallel algorithm design for spatially dependent map reprojections is challenging. Compared to reprojecting vector datasets, which are based on the point-by-point coordinate transformation between input and output datasets, reprojecting raster datasets involve conditions where multiple input raster cells are aggregated and/or resampled from smaller components to calculate pixel values and generate output datasets [4,21]. Because of the spatial dependency in data aggregation, input data are difficult to split among multiple processors. The first two issues exist for both raster and vector reprojection whereas the last issue is unique to raster reprojection.

Consequently, we propose a CUDA parallel algorithm driven by a chunk based output space for map reprojections of these large raster datasets. We also explore the best practices behind block and chunk assignments through conducting a set of tests, which are compared to traditional serial processing methods using only CPUs.

3. Parallel Design and Implementation of Map Reprojections with CUDA

3.1. Rigorous Raster Reprojection in a Serial Processing Manner

Map reprojections for raster datasets have been studied for several decades (e.g., [4,26–28]). Despite various ways of performing reprojections, we refer to the accurate raster reprojection technique presented in [4]. Compared to map reprojection methods based on the center point of a pixel and treated as a vector map reprojection (with only nearest neighbor resampling), this cited method computes the four corner points of a pixel and allows various options for resampling the resulting pixel value. This method yields results that are more precise, although it is more time consuming. The rigorous raster reprojection method, for serial processing, consists of three primary steps described below:

Step 1: Read the input raster dataset and calculate output space. In this step, the algorithm reads the information from the input dataset. This information can then be used to determine the configuration information of the output raster. In particular, the geographic extent of the output image is determined using methods such as the *MinBox* specification [4]. This *MinBox* is a rectangular box, which spans from the top left corner of the dataset to its lower right (thus covering the entire area of the raster, regardless of overall shape or curvature). In practice, a user specifies the minimum and the maximum coordinates of the output extent and the numbers of rows and columns to create the frame for the output raster area.

Step 2: Perform the reprojection for pixels. Instead of applying the center values from each raster pixel as would be standard, the Steinwand [27,28] algorithm uses the coordinate values of the four corner points from each pixel in the transformations. Two groups of equations are used which support the conversion between projected coordinate values and the column and the row numbers of a pixel. The equations [4] are:

$$X_{\text{output}} = ULprojX_{\text{output}} + \left((column_{\text{output}} - 1) * pixelSizeX_{\text{output}} \right) \quad (1)$$

$$Y_{\text{output}} = ULprojY_{\text{output}} - \left((row_{\text{output}} - 1) * pixelSizeY_{\text{output}} \right) \quad (2)$$

Alternatively:

$$Column_{\text{input}} = \left(\frac{(X_{\text{input}} - ULprojX_{\text{input}})}{PixelSizeX_{\text{input}}} \right) + 1 \quad (3)$$

$$Row_{\text{input}} = \left(\frac{(ULprojY_{\text{input}} - Y_{\text{input}})}{pixelSizeY_{\text{input}}} \right) + 1 \quad (4)$$

where*:*

X, *Y* represents a pair of projected coordinate values;

Row, *Column* represents the row and the column numbers;

pixelSizeX, *pixelSizeY* are the size of a pixel on the ground;

ULprojX, *ULprojY* are a pair of projected coordinate values of the upper-left most pixel (in image coordinates) in the dataset.

To identify the value of each pixel in the output image, the algorithm calculates the coordinate values of each pixel's four corner points based on Equations (1) and (2). Given a pair of coordinate values for a corner point, the algorithm first performs an inverse projection to derive a pair of geographic coordinate values of the point and then performs a forward projection with the obtained geographic coordinate values to derive the projected coordinate values in the input coordinate system. Upon identifying the projected coordinate values for all four corner points, the algorithm applies Equations (3) and (4) to generate a quad consisting of the four points in the input image, again regardless of the raster's shape (which is why, once more, using four pixel corners proves more precise than standard procedures). This overall process is repetitively performed for every pixel in the output

image. The values of all pixels falling in the quad are retrieved and resampled to produce aggregated pixel values. These values are assigned as the values of the pixels in the output image.

Figure 1 illustrates the process flow for a map reprojection process of one pixel. The example in Figure 1 uses simulated coordinates to illustrate the processes. Before explaining the figure, we formally introduce the following concepts: *Rows* are series of cells that span across a raster dataset horizontally; each row therefore stacks on top of the next set of cells in the y direction (i.e., up and down). *Columns* are series of cells that span across a raster dataset vertically; each column therefore stacks next to the following set of cells in the x direction (i.e., left and right). *Resolution or pixel size* is the pixel or cell size (in meters for all of our datasets), across one direction. *Raster grid size* is the number of pixels per dataset (i.e., the numbers of columns by rows). For a given pixel, we obtain the column and row numbers of the four corner points (denoted as red dots in Figure 1) and identify the projected coordinates (Equations (1) and (2)). Then, we perform inverse projection to convert projected coordinates (2D) to geographic coordinates (3D, real position on Earth) as well as forward projection to convert the geographic coordinates to projected coordinates in the input image. The projected coordinates are converted into column and row numbers (Equations (3) and (4)). With column and row numbers, we can locate the point in the input image. Four points consisting of a quad with all pixels are used to generate the pixel value in the output space.

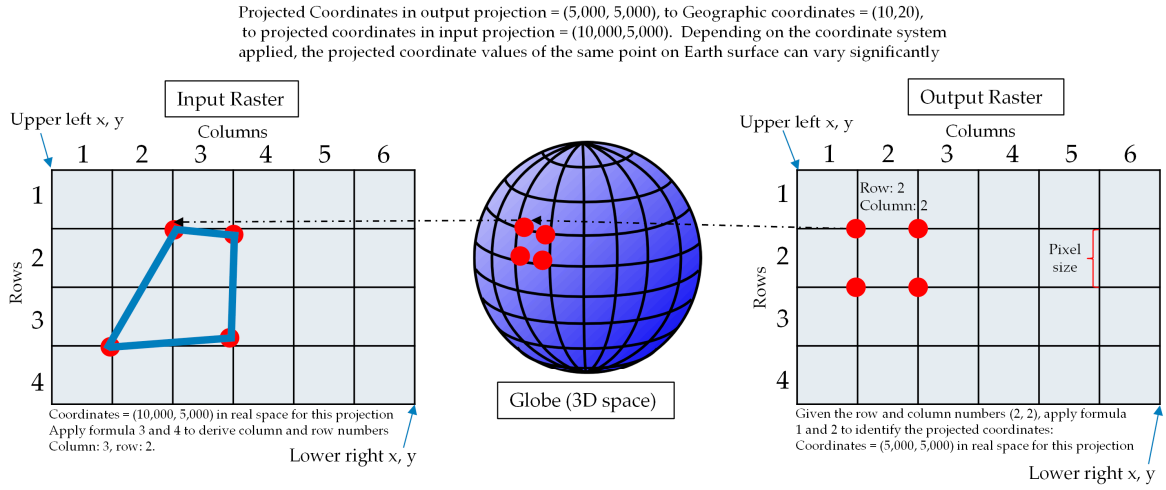


Figure 1. The process of performing map reprojection using one pixel in the output image as an example.

Step 3: Combine map reprojection results and produce output. In this final step, the pixel values from previous steps are used to produce the resulting projected image. Meanwhile, the configuration information of the output image is written as a part of the image. Thus, pixel values and the configuration information are used to form the final output raster dataset with the given projection.

3.2. CUDA-Based Parallel Design and Implementation

As discussed in Section 2, the parallel implementation should be carefully designed to allow the map reprojection to be performed in an independent manner. The parallel implementation is based on the output dataset space and every pixel in the output space is processed with one GPU thread while using CUDA. This parallel design is defined for two reasons. On the one hand, the output pixels are determined by variable pixels in the input image, as it would be difficult to distribute input pixels accurately to different computing threads otherwise. On the other hand, every pixel in the output image is independent from each other. The independence is well suited for parallel implementation, though it nevertheless requires that the algorithm identifies a spatial quad (i.e., MinBox) in the input

image to enclose all pixels, to then calculate the corresponding pixel values of the raster cells in the output image.

Figure 2 describes the process of performing map reprojections using CUDA. This process is similar to the three-step procedure described in the previous section: the parallel algorithm takes the configuration information of the output raster image (such as spatial resolution and output coordinate system) and an input raster image as input information. To enable fast parallel processing, every GPU thread processes one or a set of pixels in the output image. This part of the process is identical to Step 2 of the serial processing method, though with GPU components. When the algorithm identifies the pixel values of all pixels in the new image, the results are written to an output raster file. (Note: depending on the selected type of map projection, the coordinate transformation process applies different mathematical formulas, though there are no additional steps in the parallel version regarding these.)

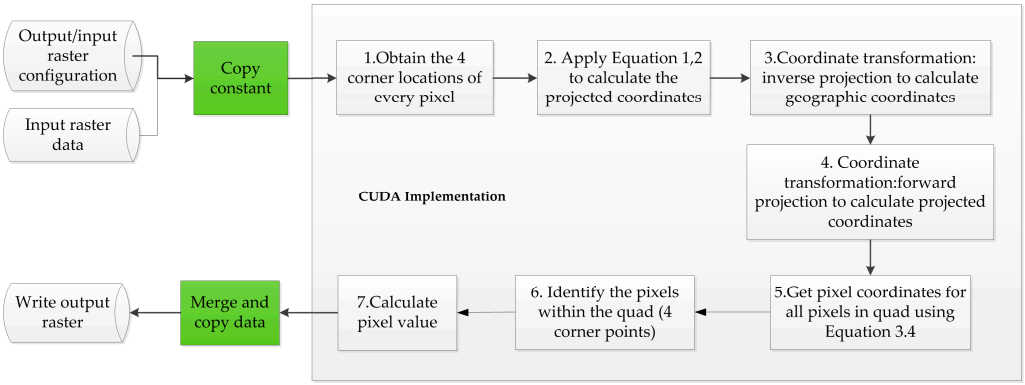


Figure 2. The 4-corner point based map reprojection using CUDA.

Compared to the serial processing method of map reprojection described in [27], the parallel implementation version described in Figure 1 has special steps. First, the parallel implementation involves two additional procedures of data transfers between GPU and CPU (denoted as green boxes in Figure 2). Data transfers can introduce additional processing overhead, due to the limited amount of GPU memory available in personal computers (very similar to the main memory). Secondly, the pixel value assignment, which corresponds to Step 2 in the serial processing method, is instead performed with GPU threads in parallel (i.e., alongside each other, but simultaneously). This ability for tasks to be distributed for processing at the same time thus reduces computational time costs, even if communication between CPU and GPU is added for these parallel processes.

3.3. Handling of Raster Chunks

In parallel computing, particularly with large datasets, data decomposition is necessary to break up the entire process into manageable tasks. Since the memory of the on-board GPU is limited, data decomposition is extremely important to ensure the map reprojection process is performed successfully without memory overflow. The memory of a GPU ranges from 1 GB to several GB in high-end expensive GPU devices. GPU memory is generally less than main memory. When the file size of the input image exceeds the on-board memory of GPU, which occurs very frequently, the concept of "chunk" is introduced [5] to allow GPUs to perform map reprojections on subsets of the input image. A *chunk* is defined as a subdivision of the dataset used for parallel processing decomposition, measured in pixels/cells across one direction. Each chunk is processed as one grid. A *grid* is a container of blocks, created when a kernel (i.e., CUDA function) is launched. Grids are then subdivided into blocks. A *block* contains many threads. A *thread* thus processes one pixel (or a set of pixels) in a chunk of the output image.

To illustrate the process of generating chunks, Figure 3 shows a decomposition example. In Figure 3a, the dataset has a raster grid size of 16 rows by 16 columns, or 16 pixels in the 'x' direction (columns), and 16 pixels in the 'y' direction (rows). The spatial resolution is 1000 meters. This raster dataset (green outline) is divided into four chunks (blue outlines). The blue example chunk comprises 8 columns by 8 rows of pixels (in red), and these pixels are 1000 meters in resolution too. Chunks then have the same spatial resolution as the original image, but with smaller raster grid sizes. Figure 3b shows the block and thread composition for a grid containing a chunk from Figure 3a. In this case, the grid has four blocks, and each block contains four threads. In practice, when the data volume exceeds the on-board memory, the image will be decomposed into many more chunks and contain many more blocks and threads. If memory overflow does not take place (e.g., due to a small size dataset), the standard CUDA-based map reprojection described in Figure 2 is applied directly.

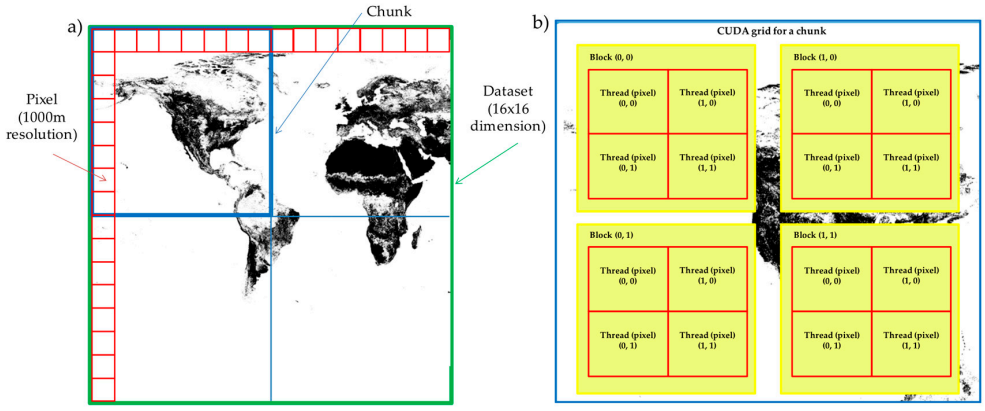


Figure 3. Chunk creation and task assignment: (**a**) Decomposition of dataset into chunks and pixels; (**b**) The structure of threads within blocks of one grid.

The chunk-based map reprojection method (see Figure 4) starts with the provision of an image as input, and configurations of the projected output image. Additionally, the size the chunks for breaking up the output image should be specified. Based on these provided raster grid sizes for chunks, the algorithm generates the divisions, so that every chunk is indexed with a pair of row and column numbers. Given the raster grid size information, the algorithm only retrieves the pixels in the input image required by each of the chunks on the fly. As a result, instead of loading the complete input image, only a portion of the image raster pixels are loaded at a time for GPU processing, to avoid memory overflow. Then, the algorithm performs parallel map reprojections with every chunk available. The process ends in the ultimate merging of the initial dataset components after performing map reprojections on individual chunk pixels. Depending on how users may want to access the reprojected results, the datasets can be organized through different data indexing methods and loaded individually.

A chunk-based map reprojection requires one critical step, where corresponding pixels in the input dataset image are retrieved from matches in the output image. To identify those pixels, we first gather all bounding pixels of the chunk in the output dataset space (that is, the pixels on the boundary of the chunk), and then perform an inverse map projection for all those bounding pixels. We then find the locations of the corresponding pixels in the input image and formulate region(s) or polygon(s) based on those pixels. All pixels in the input image falling in the region(s) then undergo map reprojection for the chunk in question. To facilitate fast data retrieval, instead of using irregular regions to identify the pixels, we create a MinBox with simpler bounds from which to identify pixels. This step is also implemented in a parallel manner. Figure 4 above again summarizes the basic flow of the process, while Figure 5 below has two examples of the input and output dataset spaces as well as forward and inverse projections, for visual representations:

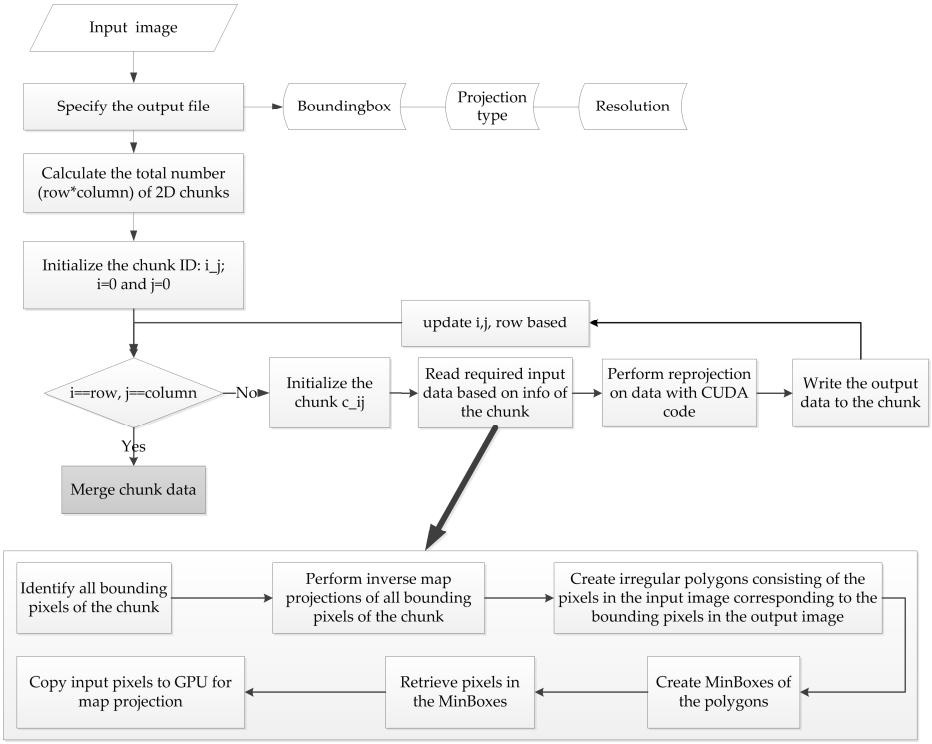


Figure 4. Process for CUDA-based map reprojection with chunk decomposition.

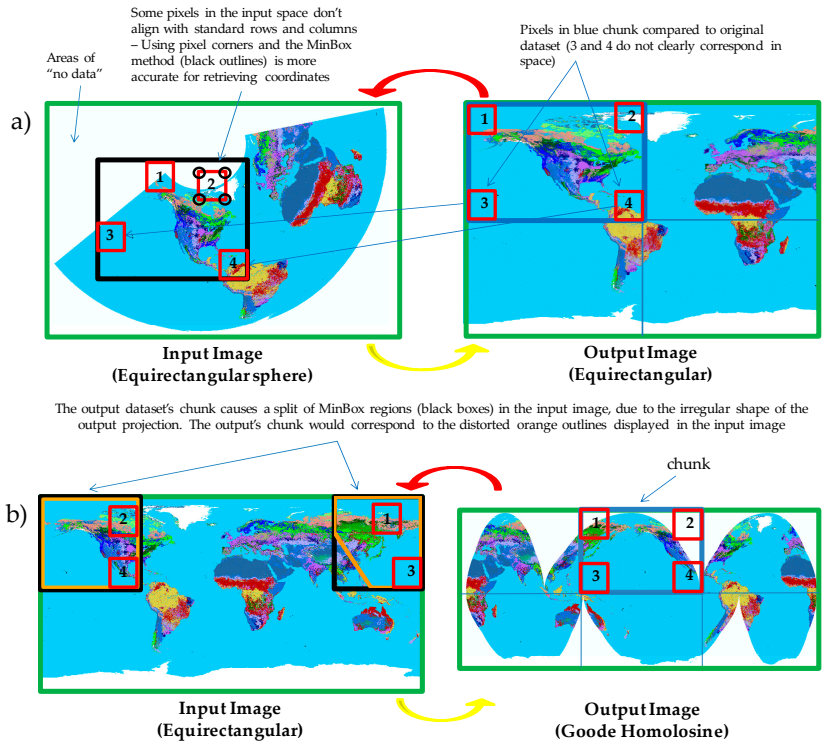


Figure 5. (**a**) Equirectangular sphere and Equirectangular standard projections. Pixel corners in the input image are shown in black circles; (**b**) Goode Homolosine and Equirectangular projections. Both figures show correspondence between example pixels in an output image's chunk (blue outline) and where those would fit in the input image's MinBox (black outlines).

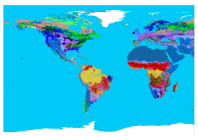
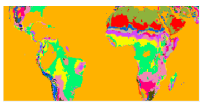
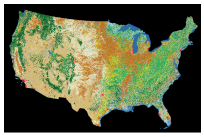
Figure 5 shows the unequal matching between pixels in the output datasets' chunks and where those would fit in the input images. Pixels 1, 2, 3 and 4 are examples of the bounding pixels for the output images' chunks. In Figure 5b, we demonstrate how splitting of a MinBox may occur based on irregularities in a dataset (black outline). The importance of using the four corners of a pixel rather than the standard pixel center, given these missing correspondences of bounding boxes, is portrayed in both figures too. These aspects highlight the benefits of using inverse and forward projection as part of the reprojection process with chunks, as transformation distortions and irregularities can be common in standard procedures (see Figure 5b and the orange outlines, which indicate the irregular chunk shape correspondence from the output image space to the input image's). Lastly, our process of using the four corners in a pixel increases accuracy for finding pixel coordinates.

4. Experiments

4.1. Test Environment

To demonstrate the advantages of using a CUDA-based parallel map reprojection, we performed a set of experiments. Due to the variability of map projection types, we only selected three representative map projection types to conduct tests; these are Equirectangular, Mollweide and Albers (provided in Geospatially referenced Tagged Interchange File Format (GeoTIFF) format). Table 1 describes the metadata of the datasets used. We selected these three for the following reasons. First, the chosen map projections are used for producing maps with large spatial extent, which require the use of accurate map reprojection methods. Second, they are applicable to different spatial coverages. Albers is used to model North American regions whereas Equirectangular and Mollweide are common for global maps. The conversion between these map projections can test how our algorithm can handle the issues described in Figure 4. Finally, the projections carry different yet common types of distortions, which allow us to test the effectiveness of the four-corner point projection method with several projections.

Table 1. A summary of the test datasets (https://cegis.usgs.gov/projection/mapProjections.html).

Example Image	Projection	Resampled Dataset Resolution (in meters)	Raster Grid Size (rows by columns)	Description
	Equirectangular	1000 2000 5000 10,000 50,000	28,030 × 20,015 14,015 × 10,008 5606 × 4003 2803 × 2002 560 × 400	This dataset covers most of the world and shows different land cover types across the globe.
	Mollweide	1000 2000 5000 10,000 50,000	18,039 × 8021 9019 × 4010 3608 × 1604 1804 × 802 361 × 162	This dataset covers areas near the equator on the globe and displays either land cover types or life zone types.
	Albers	150 300 600 1200 6000	32,238 × 20,885 16,119 × 10,442 8060 × 5221 4030 × 2611 806 × 522	This dataset covers the entire contiguous United States and shows different land cover types across the area.

For testing and comparison purposes, we performed resampling using the nearest neighbor method to generate multiple images at different spatial resolutions. The resampling method may introduce noise at boundaries [29], and the spatial extent of the images may change slightly after resampling. Resampling allowed us to achieve two goals. First: to derive images of different spatial resolutions to examine the role of spatial resolution in changing the performance of map reprojections. Second: to reduce the dataset sizes and allow the computing environment to manage and process the images properly (as some of them were originally over 16 GB in size). This is due to memory allocation limitations, given that a machine cannot handle files larger than its main memory. As a

result, every type of image was resampled into a group of images. For example, the file sizes of the Equirectangular (resolution at 1000 meters) and Albers (resolution at 150 meters) images exceeded the GPU on-board memory of the desktops for testing (Table 2). The corresponding raster grid size information is provided in Table 1. Given the raster grid sizes of resampled datasets, chunking was applied to the first four image raster grid sizes from each group except for the last one (i.e., that with the lowest spatial resolution), due to the ability of the machine to handle projecting without chunking these coarser datasets. Note that resampling is not necessary if the file size of the image fits the main memory of the computer.

Table 2. Configuration of the testing machines.

Machine Information	Machine 1	Machine 2
CPU	Intel Quad-core i5 3.10 GHz	Intel Quad-core i7 3.40 GHz
Main memory	8 GB	8 GB
GPU	NVIDIA GeForce GT 640, 384 GPU Cores 1 GB Memory	NVIDIA GeForce GT 640, 384 GPU Cores 1 GB Memory

Output images are configured with the same pixel type, number of bands, and pixel size as the original input image. The output image has a different coordinate system and thus shape compared to the input image. The spatial extent and the number of columns and rows are adjusted based on the extent of the input image. After the adjustment, the output image should cover the same spatial extent as the input image, but in the format of the coordinate system of the output image.

We ran map reprojection tests with two desktop machines. We did not perform the tests with powerful workstations because we want to evaluate the feasibility of applying our lightweight map reprojection method within a less powerful, more conventional desktop computing environment. The specific configurations are provided in Table 2. Both desktops are equipped with the same GPUs, which have 384-GPU cores. Machine 2 has a slightly better CPU speed, however. The slightly different configuration allows us to examine the role of CPU in changing the overall performance. The development took place using NVIDIA CUDA SDK version 5.5, in a Visual Studio 2010 environment.

4.2. GPU Speedup Ratios

We first examine the time costs of performing the same sets of map reprojections with GPU and CPU separately. In the case of CPU based implementations, every pixel in the output image is processed in a serial manner. In the case of GPU based implementations, every pixel is processed by one thread, though a parameter exists in our script that allows the user to adjust the number of pixels handled by each thread, potentially minimizing the GPU load imbalance. We recorded the combined time costs, consisting of the time costs from reading data, performing pixel-by-pixel projection, and writing output data. CUDA-based implementations only accelerate the pixel-by-pixel projections. In all scenarios, therefore, tests with GPU delivered faster computational times. To show the performance gains and highlight the benefits of using this parallel setup, we calculated the speedup ratios. A speedup ratio is defined as the proportion between the time cost of performing a map reprojection with CPU and the time cost of performing the same map reprojection with GPU (see Equation (5) below).

$$R = \frac{t_{cpu}}{t_{gpu}} \tag{5}$$

where:

R is the speedup ratio;

t_{cpu} is the time cost of running the map reprojection with CPU, in seconds;

t_{gpu} is the time cost of running the map reprojection with GPU, in seconds.

Figure 6 below shows the raw CPU time costs and the speedup ratios when performing different types of map projections. The raw CPU time costs generally decline linearly with the decrease of spatial resolutions (as they become coarser), because fewer pixels are contained in those input images. Note that, since multiple runs of tests are conducted for each reprojection instance, only the best performance of CPU and GPU is used. The speedup ratios range from ~10 times to ~100 times. In Figure 6b, the reprojections from Albers to Mollweide with Machine 1 overall yield the highest GPU speedup ratios (shown with black bars), almost reaching the 100 times mark for the lowest pixel resolution (i.e., 6000 meters). In Figure 6d, the reprojections from Equirectangular to Mollweide with Machine 1 overall yield the highest speedup ratios (shown with black bars). These are more obvious in the datasets with lower resolutions. In Figure 6f, the reprojections from Mollweide to Equirectangular with machine 1 show the highest speedup ratios (shown with black bars), but this time it is the highest resolution dataset that yields the best speedup (i.e., 1000 meters).

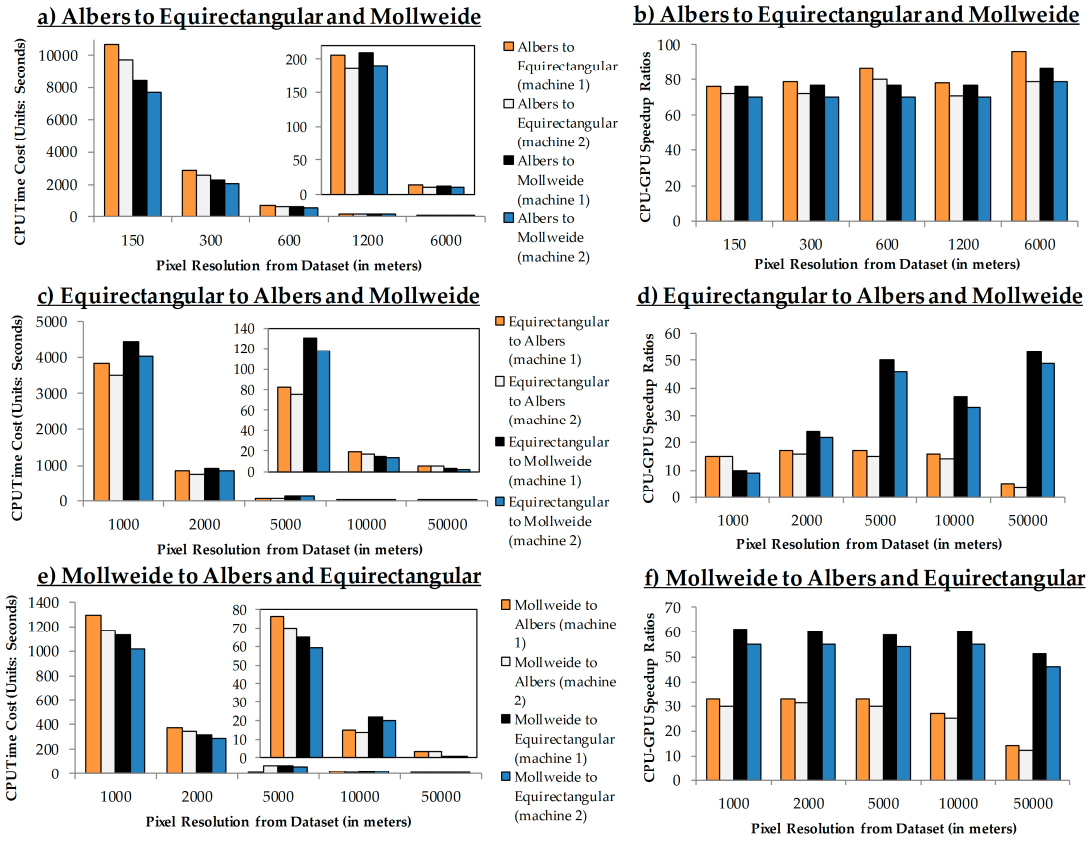


Figure 6. Time Costs of Performing Map Reprojection with CPU and CPU-GPU Speedup Summaries for the same reprojections: (**a**) and (**b**) From Albers to Equirectangular and Mollweide; (**c**) and (**d**) From Equirectangular to Albers and Mollweide; (**e**) and (**f**) From Mollweide to Albers and Equirectangular.

When we examined differences in speedup ratios, we found that these vary significantly with map projection types. The speedup ratios also vary slightly with different spatial resolutions. Overall, we observe the best speedup ratios from Machine 1, which implies that the CPU-based implementations of Machine 1 are slower than those of Machine 2 (and therefore create larger time cost differences between CPU and GPU). The difference in speedup ratio can be attributed to several factors. First, the complexity of different map reprojection algorithms varies from relatively simple spherical projections to complex approximations with many polynomial ellipsoidal projections [29]. The time cost of map reprojection of every pixel can thus vary depending on the location of the pixel in the

output and input image. The pixel size can also determine the variation of reprojection. This is confirmed by the fact that the time cost of CPU is not linearly correlated with the number of pixels for different images. In the case of parallel processing, threads are synchronized to obtain the final output, though a load imbalance can lower the performance (which is not an issue for serial processing). Second, due to the irregular raster grid size of images and block sizes (e.g., raster grid size of 100 rows by 100 columns, with block size of 32 threads by 32 threads) and because of the round up issue in CUDA algorithms (in the previous example, 4 by 4 blocks process 128 rows by 128 columns of pixels), some blocks are created which include threads that do not process any pixels. This again can introduce a load imbalance. Third, CUDA-based implementations involve data transfer between CPU and GPU. The number of rounds of data transfer varies with different images based on size and projection complexity. Finally, when the concept of chunks is introduced, the computational cost and the data transfer cost between CPU and GPU vary with the locations of these chunks in the output image (as illustrated in Figure 5). Formulating an optimal strategy to maintain load balance and best utilize GPU computing capabilities is beyond the scope of this article, however.

4.3. Data Chunk Size, Spatial Resolution, GPU Block Size, and Performance

To examine the impacts of chunk sizes on the performance of map reprojections, we carried out map reprojections for every image using a fixed block size and a set of varying chunk raster grid sizes of 128 rows by 128 columns (chunk 128), 256 rows by 256 columns (chunk 256), 512 rows by 512 columns (chunk 512) and 1024 rows by 1024 columns (chunk 1024). We additionally conducted a similar set of experiments to examine the role of block sizes in changing the performance, by testing constant chunk sizes against varying blocks of 32, 64, 128, and 256 threads. We then grouped images by their original map projections to summarize the relationship between the changes of spatial resolution (which mainly lead to the changes in number of pixels) and performance. Figures 7–9 show the map reprojections results with GPU using Machine 1. Because both machines have the same GPU devices but Machine 1 shows the most GPU performance gains, we only present the results from this machine.

As Figure 7 shows, in both sets of projections we see that chunk 512 is the most efficient in terms of processing speed (see Figure 7a,c). On the other hand, the fixed chunk size yields the best processing speeds with block 128 (as shown in Figure 7b,d). In particular, with the fixed chunk pattern, we noticed that the time costs decrease significantly when the block size changes from 32 to 64 (Figure 7b,d).

For the first reprojection in Figure 8a below, we see that chunk 1024 is the most efficient in terms of processing speed, though in Figure 8c chunk 256 is the fastest in reprojecting (shown with black bars). In Figure 8b,d, however, we notice the same pattern as in Figure 7 with the fixed chunk size, where the best processing speeds come from block size 128. Concerning the fixed chunk pattern, we again observed that the time costs decrease significantly when the block size increases from 32 to 64 (see Figure 8b,d).

Next, in Figure 9, for both sets of reprojections, we see that chunk 512 is the most efficient in terms of processing speed (see Figure 9a,c). On the other hand, the fixed chunk size yields the best processing speeds while using block size 128 (as shown in Figure 9b,d). The fixed chunk pattern, where the time costs decrease significantly upon changing the block size from 32 to 64 (see Figure 9b,d), is noticeable once more.

Given the results shown in Figures 7–9, the processing times generally linearly change in the negative direction with the spatial resolution. Changing spatial resolution leads to the changes in pixel numbers. For example, changing from 600 meters to 300 meters leads to 4 times the number of pixels needing to be reprojected, and thus the processing time is about 4 times higher. Changes in the pixel numbers are hence correlated with the increases in time costs. Given the different block and chunk configurations we have tested, we found that the best results are overall found when the chunk size is 512 rows by 512 columns and the block size is 128, or somewhere in between the decomposition configuration options. The results are consistent across the different map reprojection processes.

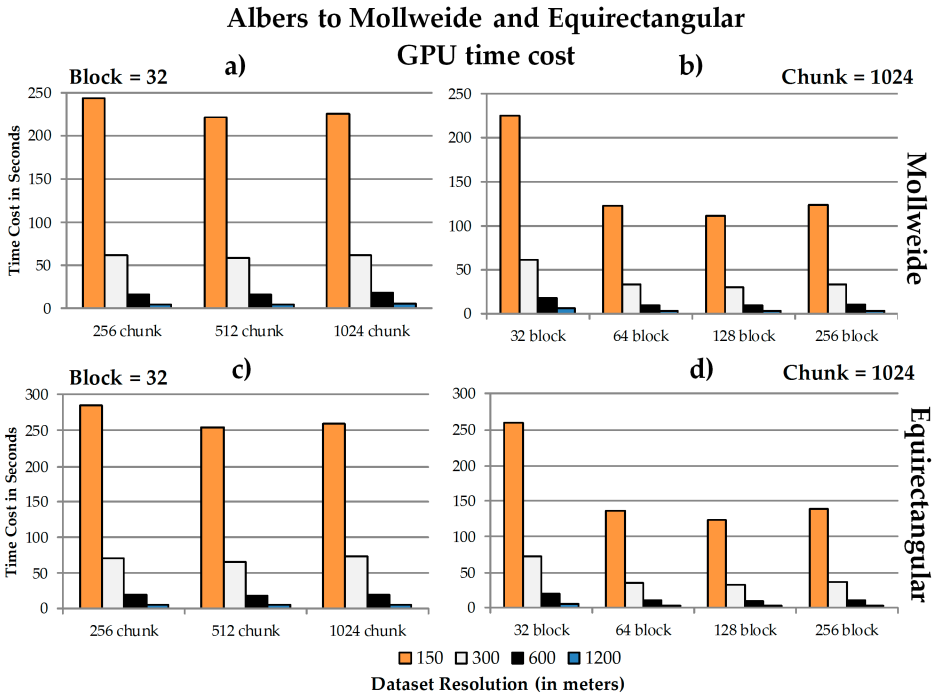


Figure 7. From Albers to Equirectangular and Mollweide: (**a**) and (**b**) show Albers to Mollweide reprojections; (**c**) and (**d**) show Albers to Equirectangular reprojections. Charts (**a**) and (**c**) use a fixed block size of 32 threads, while charts (**b**) and (**d**) use a fixed chunk raster grid size of 1024 rows by 1024 columns.

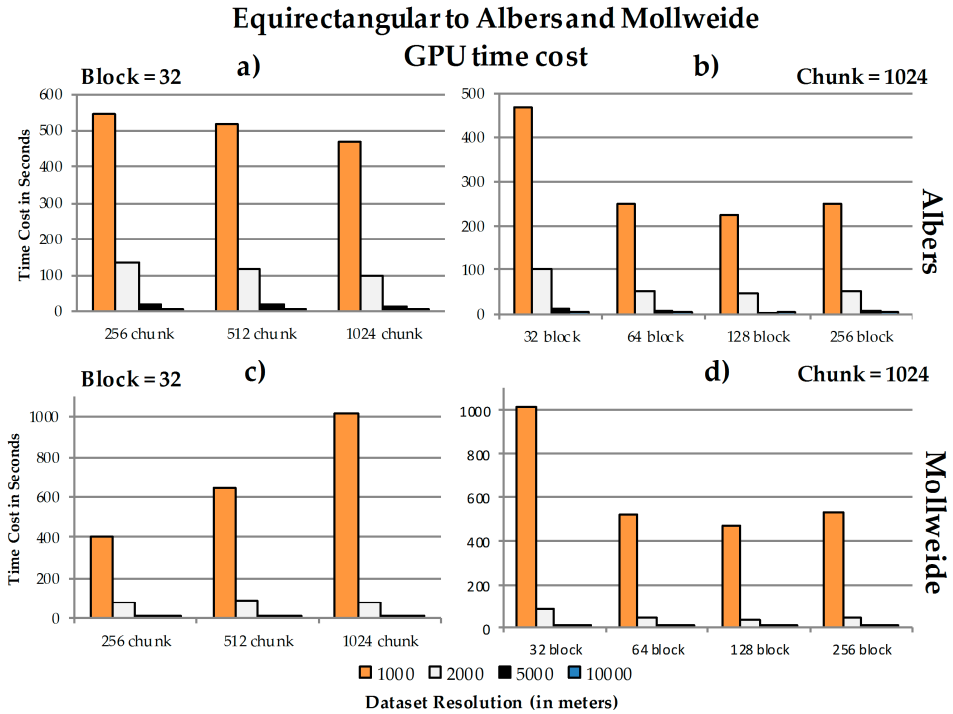


Figure 8. From Equirectangular to Albers and Mollweide: (**a**) and (**b**) show Equirectangular to Albers reprojections; (**c**) and (**d**) show Equirectangular to Mollweide reprojections. Charts (**a**) and (**c**) use a fixed block size of 32 threads, while charts (**b**) and (**d**) use a fixed chunk raster grid size of 1024 rows by 1024 columns.

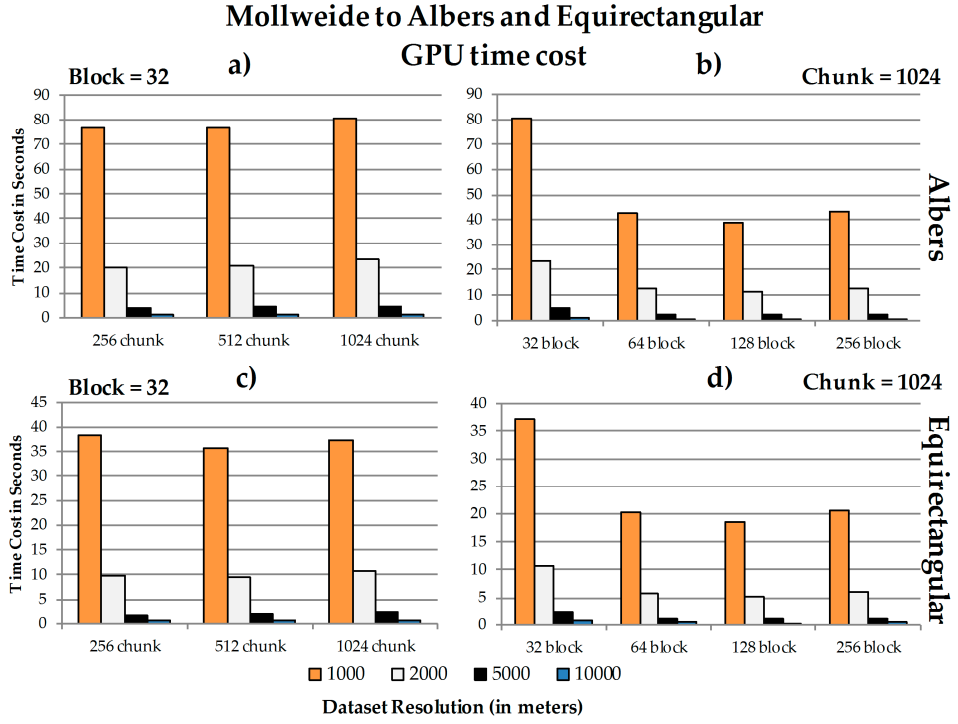


Figure 9. From Mollweide to Albers and Equirectangular reprojections: (**a**) and (**b**) show Mollweide to Albers reprojections; (**c**) and (**d**) show Mollweide to Equirectangular reprojections. Charts (**a**) and (**c**) use a block size of 32 threads, while charts (**b**) and (**d**) use a fixed chunk raster grid size of 1024 rows by 1024 columns.

4.4. Discussion

4.4.1. Performance

a. GPU speedup ratios

In all scenarios, we observed GPU speedups, proving how the parallel implementation can reduce the processing times when compared to CPU methods. The speedup ratios can vary with different map projection types. These variations are attributed to the complexity of transformations associated with those map projection types. They vary slightly with spatial resolutions as previously noted in Section 4.3. Given the same GPU configurations, CPU speed also plays a role in GPU processing gains because of the data transfer and pre/post processing communications involved. In our case, both machines have the same GPU devices but different CPUs. The CPU speed of Machine 1 is slightly slower than that of Machine 2, and thus the greater speedup changes were observed with this slower CPU machine (due to the larger the range in processing speeds and increased overhead from CPU-GPU communications). The results from Machine 2 were not shown in the GPU time cost figures to avoid providing very similar and redundant results, as the overall GPU timing patterns were the same.

b. The relationship between chunk size, block size and spatial resolution concerning the speedup ratios

When the chunk size falls within the limit of GPU memory, given the same block configuration, a larger chunk usually yields better performance (i.e., chunk with a raster grid size of 512 rows by 512 columns or 1024 rows by 1024 columns). This is likely because larger chunks lead to fewer rounds of on-the-fly data retrieval. Besides, when extremely small chunks are used, the performance tends to be worse because the over-decomposition process requires more computational overhead and more rounds of data transfer (e.g., communication between CPU and GPU), thus eliminating the

benefits of using CUDA. The results also show that, generally, a medium-larger block size yields better performances (128 is the overall best among block sizes of 32, 64, and 256), though again, performance can vary slightly depending on the utilization of parallel processing capabilities of a GPU. It is recommended that 128 to 256 threads be used to fully utilize CUDA occupancy [3]. Results show that the speedup ratios generally are linearly correlated with the number of output pixels given the same spatial extent, so that the coarser the pixel resolution (i.e., larger pixel sizes) in a dataset, the faster the GPU time costs will be.

4.4.2. Issues

When we conducted experiments, we found two particular issues that should be further investigated. First, the workload is not well balanced in the kernel setups, as the processing workloads may not be the same even though the chunk sizing is the same (See Figure 5a). We may want to introduce a module that can automatically identify the chunk and the block configuration based on the characteristics of the data and the computing devices, to improve individual performance (including the assignment of pixels per thread to optimize the GPU load balance). Secondly, creation of chunks in the parallel data decomposition process is not configured based on the characteristics of the input datasets (e.g., spatial extent). The way to retrieve input data chunks based on the bounding box of output chunks (with the MinBox) is not always the most effective. We identified an edge problem where, for chunks near the edges of datasets that require some overlap to be retrieved (or are out of bounds), negative coordinate values in the input raster were returned, which are not necessary in processing. This may introduce additional processing time in the map reprojection process. Figure 10 shows an example of where negative coordinate values may be derived. In future work, we plan to improve the data retrieval process to avoid this issue.

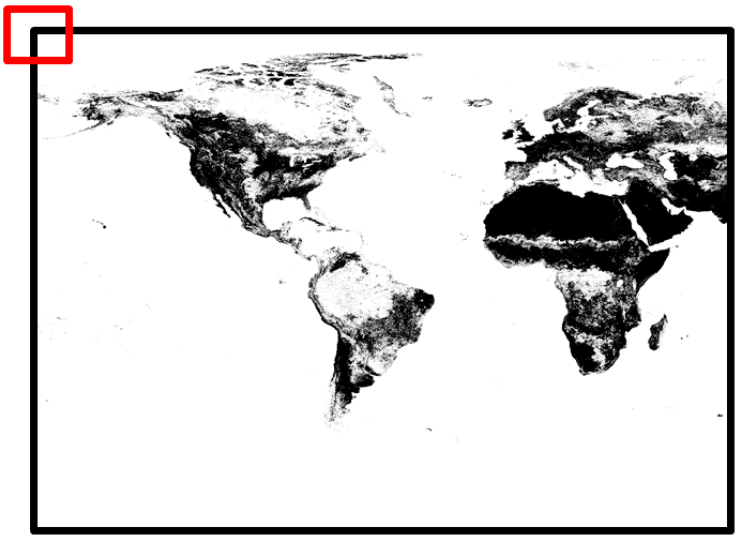


Figure 10. An example of a bounding box (MinBox) highlighting an edge at the top left (in red) where errors may be introduced for processing chunks, due to data section overlap/pixels falling out of bounds.

5. Conclusions

Rigorous reprojection of raster data using the four-corner point algorithm is more computationally intensive than the raster map reprojection handling a single point per pixel but yields a higher degree of spatial dependency. Our output space-based data decomposition method allows map reprojections of raster data to be implemented in a parallel manner. Implemented with CUDA-enabled GPUs, our approach can be performed in an affordable and lightweight desktop environment, instead

of requiring high-end supercomputers that have been previously suggested in literature [21,30]. The chunk-based on-the-fly data retrieval strategy we employed overcomes some limitations of GPU memory overflow in personal computers when processing large files. To our knowledge, this is the first lightweight parallel algorithm that handles map reprojection for large raster datasets using both this four-corner pixel algorithm and chunking decomposition within the CUDA environment.

We used varying approaches to assess the GPU based map reprojection method. As the results show, CUDA delivers a more efficient computing solution than the traditional CPU, with speedup ratios ranging from ~10 times to ~100 times of improvement. With the chunk-based data retrieval strategy, large raster datasets can be processed while only introducing minimal computational overhead times. We found that the most efficient configuration required using relatively larger chunk sizes and medium-large CUDA block sizes. The best performing setups used 512 rows by 512 columns of pixels in raster grid size for chunks with a block size of 32 threads, and block sizes of 128 threads in raster grid size when implementing chunks of 1024 rows by 1024 columns of pixels.

Future work includes expanding this research to support various additional types of map projections; we also hope to apply this projection method to parallel computing strategies in multi-GPU or GPU cluster settings, in order to examine these reprojection processes in state-of-the-art systems (e.g., supercomputer workstations). Based on the expansion, extensive tests with different computing environments and CUDA optimization parameters (e.g., chunk, block, thread assignments in a kernel) will therefore be conducted to explore optimized solutions tailored to those environments.

Acknowledgments: The authors want to thank the U.S. Geological Survey (USGS) Center of Excellence for Geospatial Information Science (CEGIS), for providing financial support under grant G16AC00152. The grant additionally covered the costs required to publish in this open access journal.

Author Contributions: Jing Li and Michael P. Finn conceived and designed the experiments; Michael P. Finn provided guidance and resources for applying methods from previous research to this project; Jing Li performed the experiments; all three authors analyzed the data and compiled the results; Marta Blanco Castano designed the tables and figures for summarizing the data and findings; all three authors contributed to designing and writing the paper.

Conflicts of Interest: The authors declare no conflict of interest.

Disclaimer: Any use of trade, product, or firm names in this paper is for descriptive purposes only and does not imply endorsement by the U.S. Government.

References

1. Wang, S. A CyberGIS Framework for the Synthesis of Cyberinfrastructure, GIS, and Spatial Analysis. *Ann. Assoc. Am. Geogr.* **2010**, *100*, 535–557. [CrossRef]
2. NVIDIA Corporation. Compute Unified Device Architecture (CUDA). Available online: http://www.nvidia.com/object/cuda_home_new.html (accessed on 10 January 2017).
3. Bakhoda, A.; Yuan, G.L.; Fung, W.W.L.; Wong, H.; Aamodt, T.M. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, Boston, MA, USA, 26–28 April 2009; pp. 163–174.
4. Finn, M.P.; Steinwand, D.R.; Buehler, R.A.; Mattli, D.; Yamamoto, K.H. A Program for Handling Map Reprojections of Small Scale Geospatial Raster Data. *Cartogr. Perspect.* **2012**, *71*, 53–67. [CrossRef]
5. Beynon, M.; Chang, C.; Catalyurek, U.; Kurc, T.; Sussman, A.; Andrade, H.; Ferreira, R.; Saltz, J. Processing Large-Scale Multi-Dimensional Data in Parallel and Distributed Environments. *Parallel Comput.* **2002**, *28*, 827–859. [CrossRef]
6. Peng, C.; Sandip, S.; Rushing, J. A GPU-accelerated approach for feature tracking in time-varying imagery datasets. *IEEE Trans. Vis. Comput. Graph.* **2016**. [CrossRef] [PubMed]
7. Tobler, W.R. Polycylindric Map Reprojections. *Am. Cartogr.* **1986**, *13*, 117–120. [CrossRef]
8. Tobler, W.R. Measuring the Similarity of Map Reprojections. *Am. Cartogr.* **1986**, *13*, 135–139. [CrossRef]

9. Usery, E.L.; Seong, J.C. A Comparison of Equal-Area Map Reprojections for Regional and Global Raster Data. Center of Excellence for Geospatial Information Science, 2000. Available online: https://cegis.usgs.gov/reprojection/pdf/nmdrs.usery.prn.pdf (accessed on 10 December 2016).
10. Mulcahy, K.A. Two New Metrics for Evaluating Pixel-Based Change in Data Sets of Global Extent Due to Reprojection Transformation. *Cartographica* **2013**, *37*, 1–11. [CrossRef]
11. Usery, E.L.; Finn, M.P.; Coz, J.D.; Beard, T.; Ruhl, S.; Bearden, M. Projecting Global Datasets to Achieve Equal Areas. *Cartogr. Geogr. Inf. Sci.* **2003**, *30*, 69–79. [CrossRef]
12. Finn, M.P.; Trent, J.R.; Buehler, R.A. *Users Guide for MapIMG 2: Map Image Re-Reprojection Software Package*; U.S. Geological Survey Open-File Report Series; U.S. Geological Survey: Reston, VA, USA, 2006.
13. Behzad, B.; Liu, Y.; Shook, E.; Finn, M.P.; Mattli, D.M.; Wang, S. A Performance Profiling Strategy for High-Performance Map Re-Reprojection of Coarse-Scale Spatial Raster Data. In Proceedings of the Auto Carto 2012, A Cartography and Geographic Information Society Research Symposium, Columbus, OH, USA, 16–18 September 2012.
14. Qin, C.Z.; Zhan, L.J.; Zhu, A.X. How to Apply the Geospatial Data Abstraction Library (GDAL) Properly to Parallel Geospatial Raster I/O? *Trans. GIS.* **2014**, *18*, 950–957. [CrossRef]
15. Atkins, D.; Droegemeier, L.; Feldman, S.; Garcia-Molina, H.; Klein, M.; Messerschmitt, D.; Messina, P.; Ostriker, J.; Wright, M. *Revolutionizing Science and Engineering through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure*; National Science Foundation: Arlington, VA, USA, 2003.
16. Baumann, P.; Mazzetti, P.; Ungar, J.; Barbera, R.; Barboni, D.; Beccati, A.; Bigagli, L.; Boldrini, E.; Bruno, R.; Calanducci, A.; et al. Big Data Analytics for Earth Sciences: the EarthServer Approach. *Int. J. Digit. Earth* **2016**, *9*, 3–29. [CrossRef]
17. Yang, C.; Huang, Q.; Li, Z.; Liu, K.; Hu, F. Big Data and Cloud Computing: Innovation Opportunities and Challenges. *Int. J. Digit. Earth* **2016**. [CrossRef]
18. Yang, C.; Manzhu, Y.; Fei, H.; Yongyao, J.; Yun, L. Utilizing Cloud Computing to Address Big Geospatial Data Challenges. *Comput. Environ. Urban Syst.* **2016**. [CrossRef]
19. Culler, D.E.; Gupta, A.; Singh, J.P. *Parallel Computer Architecture: A Hardware/Software Approach*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1997.
20. Hennessy, J.; Patterson, D. *Computer Architecture: A Quantitative Approach*, 4th ed.; Morgan Kauffman: San Francisco, CA, USA, 2007.
21. Finn, M.P.; Liu, Y.; Mattli, D.M.; Guan, Q.; Yamamoto, K.H.; Shook, E.; Behzad, B. pRasterBlaster: High-Performance Small-Scale Raster Map Reprojection Transformation Using the Extreme Science and Engineering Discovery Environment. In Proceedings of the 22 International Society for Photogrammetry & Remote Sensing Congress, Melbourne, Australia, 25 August–1 September 2012.
22. Li, J.; Jiang, Y.; Yang, C.; Huang, Q.; Rice, M. Visualizing 3D/4D Environmental Data Using Many-Core Graphics Processing Units (GPUs) and Multi-Core Central Processing Units (CPUs). *Comput. Geosci.* **2013**, *59*, 78–89. [CrossRef]
23. Lukač, N.; Žalik, B. GPU-based roofs' solar potential estimation using LiDAR data. *Comput. Geosci.* **2013**, *52*, 34–41. [CrossRef]
24. Ortega, L.; Rueda, A. Parallel Drainage Network Computation on CUDA. *Comput. Geosci.* **2010**, *36*, 171–178. [CrossRef]
25. Tang, W.; Feng, W. Parallel Map Reprojection of Vector-Based Big Spatial Data: Coupling Cloud Computing with Graphics Processing Units. *Comput. Environ. Urban Syst.* **2014**. [CrossRef]
26. Ryoo, S.; Rodrigues, C.I.; Baghsorkhi, S.S.; Stone, S.S.; Kirk, D.B.; Hwu, W.M.W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, USA, 20–23 February 2008; pp. 73–82.
27. Steinwand, D.R.; Hutchinson, J.A.; Snyder, J.P. Map Reprojections for Global and Continental Data Sets and an Analysis of Pixel Distortion Caused by Rereprojection. *Photogramm. Eng. Remote Sen.* **1995**, *12*, 1487–1497.
28. Steinwand, D.R. A New Approach to Categorical Resampling. In Proceedings of the American Congress on Surveying and Mapping Spring Conference, Phoenix, AZ, USA, 29 March–2 April 2003.

29. Snyder, J.P. *Map Projections: A Working Manual*; U.S. Government Printing Office: Washington, DC, USA, 1987.
30. Finn, M.P.; Liu, Y.; Mattli, D.M.; Behzad, B.; Yamamoto, K.H.; Guan, Q.; Shook, E.; Padmanabhan, A.; Stramel, M.; Wang, S. *High-Performance Small-Scale Raster Map Reprojection Transformation on Cyberinfrastructure*; Future Publication in CyberGIS: Fostering a New Wave of Geospatial Discovery and Innovation; Wang, S., Goodchild, M.F., Eds.; Springer: New York, USA, 2013.