GPU-accelerated raster map reprojection

Petr Sloup

Department of Geomatics, Faculty of Civil Engineering
Czech Technical University in Prague
Thákurova 7, 166 29 Prague 6, Czech Republic
petr.sloup@fsv.cvut.cz

Abstract

Reprojecting raster maps from one projection to another is an essential part of many cartographic processes (map comparison, overlays, data presentation, ...) and reduction of the required computational time is desirable and often significantly decreases overall processing costs. The raster reprojection process operates per-pixel and is, therefore, a good candidate for GPU-based parallelization. The architecture of GPU allows a high degree of parallelism. The article describes an experimental implementation of the raster reprojection with GPU-based parallelization (using OpenCL API). During the evaluation, we compare the performance of our implementation to the optimized gdalwarp utility and show that there is a class of problems where GPU-based parallelization can lead to more than sevenfold speedup.

Keywords: Raster reprojection, warping, parallelization, OpenCL, GPGPU, GPU.

Introduction

Representation of the surface of the Earth in two dimensions has always been important for performing various tasks such as navigation or planning. Although this process has developed significantly throughout the history, it is not possible to represent Earth's surface on a plane without distortion (as proven by [4]). Many *map projections* exist and each distorts the map in a specific way (angles, areas, distances, ...).

Because of this, maps are created in various projections depending on the depicted area and the intended use. There is often need to visually compare two or more maps or even digitally display one map over another (*overlay*). This task is usually impossible without having the maps in the same projection, which can be achieved through the process of *reprojection*.

Precise transformation of digital raster map from one projection to another requires computationally intensive per-pixel calculations, which can cause the reprojection process to take very long time (even hours for larger datasets). Because of this, working with many GIS applications can be slow and inefficient. This is especially unacceptable in certain situations such as natural crisis management (hurricanes, tsunamis, wildfire, ...) when rapid response is crucial in order to minimize property damage or even save lives. Longer processing can also be more costly – especially with modern cloud-based computing, which is often charged depending on the computation durations.

This article briefly describes the process of raster reprojection, parallelization techniques based on GPU (*Graphics Processing Unit*) and outlines how it can be utilized to achieve significantly faster reprojection times without reducing output quality.

Raster reprojection

As mentioned above, *raster map reprojection* is a process, when a new raster map in one projection is mathematically derived from an existing raster map in a different projection.

The reprojection process inputs are usually:

- Input properties: raster data (this can be one or more files, possibly accessed via network or too large to decompress as one piece); projection; extent
- Output requirements: projection; extent (if it differs from the input extent); raster dimensions or resolution
- Reprojection parameters: resampling method; `nodata` values; ...

The first phase of the process is to determine the extent of the required source data in order to avoid loading and handling of unnecessarily large data. This is usually achieved using the inverse transformation (from the output projection to the input projection) to transform regularly sampled points (at least corners) of the desired output extent. The bounding box of the transformed points (minimum and maximum in each axis) then outlines the required area in the input data that needs to be processed.

Then, there are two common approaches to the actual data transformation:

a) Forward transformation (*source-oriented*)
The more intuitive approach is to read each input data pixel, determine its coordinate in the input projection and then calculate its coordinate in the output projection using the forward transformation. The pixel color is then written to the proper output pixel position.

This approach can be quite straightforward to implement on certain platforms, but provides significantly less control over the reprojection process and implementation of different resampling methods can be very complicated. It can also be inefficient for downsampling (when the input raster is significantly larger than the desired output) or when only a subset of the raster is needed.

b) Inverse transformation (*output-oriented*)
The more common approach is to process individual pixels of the *output* raster – determine the output pixel coordinate in the output projection and calculate input coordinate using the inverse transformation.

The color of the output pixel is then determined by sampling one or more pixels near the appropriate position in the input raster. The particular sampling method depends on specific application needs.

This solution provides more control over the process and can be often computationally more efficient (since no unnecessary transformations are performed).

The `gdalwarp` utility – part of GDAL (*Geospatial Data Abstraction Library*) [5] – is often used for raster data reprojection between arbitrary projections and data formats. It uses the *inverse projection* approach described above.

Parallelization

Large datasets usually cannot be processed at once, because the uncompressed raster data would not fit into the computer's operating memory (or even hard drive). The raster datasets are therefore split into *chunks* and each of them is processed individually.

The process can be parallelized on several levels – ranging from *task parallelism* (or control parallelism) to *data parallelism*, which usually scales better with the size of a problem [1]:

a) Input/Output operations
The reprojection is done sequentially by a single thread (chunk-by-chunk), but the blocking IO operations are done asynchronously in a second thread (preparing data for the main thread).

b) Raster blocks (*chunks*)
Several chunks can be processed in parallel – modern CPUs can efficiently run up to 16 threads. GDAL implements this parallelization approach. The calculations over particular chunk (including transformations) are, however, still performed sequentially.

c) Individual pixels
The most fine-grained approach is to parallelize the individual per-pixel calculations: inverse transformation, resampling, postprocessing operations.

The per-pixel parallelization is not suitable for regular CPU (*central processing unit*) architecture – the overhead of creating and running thousands of threads on (up to) tens of cores would significantly outweigh any performance gain.

Modern GPUs, on the other hand, can have up to thousands of cores that can be programmed to perform various calculations. This allows the developers to perceive the GPUs as parallel computers and employ data-parallel programming style [6].

General-purpose computing on graphics processing units

At the beginning of the 21st century, the GPU manufacturers (driven by the entertainment industry) started to implement the *programmable pipeline* model (as opposed to the *fixed-function pipeline*, where the graphics data processing is largely fixed and cannot be programmed). The programmable pipeline allows the developers to manipulate the graphics processing and rendering by writing small programs called *shaders*. The popularity of this model led to a gradual increase in the number of processing cores, which are used to execute the shaders. The most common frameworks for GPU programming of graphics are OpenGL (*Open Graphics Library*) and DirectX.

Since a large number of cores (up to thousands) can be very beneficial for many non-graphics applications [9], there has been a significant development in the area of GPGPU (*General-purpose computing on graphics processing units*) over the last several years.

OpenCL

OpenCL (*Open Computing Language*) [8] is a framework for applications executing across heterogeneous platforms such as CPUs and GPUs. It is an open standard that provides a cross-platform abstraction to the GPGPU capabilities (as well as CPU parallelism). OpenCL drivers are available for all the latest graphics cards from major manufacturers for all major operating systems.

The framework can be used to create applications running on the *host* (CPU) that can initiate data transfers and execute programs (called *kernels* – the analogy of the shaders) on the *device* (GPU). The kernels are written in *OpenCL C language*, which is based on C programming language [7]. The code is compiled at runtime by the hardware-specific OpenCL drivers to ensure maximal portability. The kernels do not have access to IO operations (filesystem, networking, . . .) – this has to be handled by the host process and all the required data need to be explicitly transferred to the device memory prior to the kernel execution.

When the kernel is executed, it can run many times in parallel (up to the number of available cores), but the execution is similar to SIMD model (*Single Instruction, Multiple Data*; according to Flynn's taxonomy [3]) when running on GPU device – all the threads are executing the same instruction at any given time. It is therefore important to avoid code branching (conditional statements, loops with a dynamic number of iterations, . . .) to optimize performance. This is a restriction of given hardware architecture (in comparison to CPU), which allows for the much higher number of cores.

OpenCL-accelerated warping

Experimental implementation was created to evaluate the proposed idea of raster warping using OpenCL. The implemented method can be summed up into the following steps:

1. The required output raster is divided into several more manageable *chunks* than can fit into the GPU memory.
2. For each chunk, the required *source window* is calculated – by applying the inverse transformation on the uniformly sampled grid of 32×32 points covering the chunk extent and calculating their bounding box. (The value of 32 was empirically chosen as sufficient for determining the source window. GDAL uses 21 by default for a similar process.)
3. The input raster data covering the calculated source window are loaded.
4. The output pixels (covering the chunk) are calculated using the inverse transformation approach described above.
5. The calculated chunk content is written to the proper position in the output file.

Steps 1 and 2 are executed sequentially and only once. Steps 3–5 are executed sequentially and repeated for every chunk, but the execution of these steps can overlap, so the reprojection itself can run in parallel with the IO operations.

During the reprojection (step 4) the input data has to be explicitly transferred from CPU memory to GPU memory and output buffer has to be allocated. The kernel function is designed to calculate color of a single output pixel (*output pixel position* $\rightarrow$ *coordinate in output projection* $\rightarrow$ *coordinate in input projection* $\rightarrow$ *input pixel position* $\rightarrow$ *sample the input buffer*) and write it to the output buffer. The calculation is carried inside the kernel that is written in OpenCL C and compiled at the start of the application.

This means, however, that the coordinate transformation needs to be written in OpenCL C. GDAL uses PROJ.4 library [2] for the transformations, but in our implementation selected projections were ported manually.

Our implementation also uses GDAL drivers for reading and writing files to ensure the performance of input/output operations is comparable and does not distort the performance evaluation of the parallelization itself.

Evaluation

The performance of the experimental implementation was evaluated on common desktop computer (Intel Core i5-6600 @ 3.30 GHz × 4 cores, 16 GB RAM) running Ubuntu 15.10 64-bit. Internal SSD was used for reading input files and storing outputs. The computer was equipped with AMD Radeon R9 380 graphics card with 4 GB memory and 1792 cores clocked at 1000 MHz.

The *Blue Marble Next Generation* dataset [10] was used for the evaluation process (various input sizes, see Table 1 for details). The reprojection was performed from WGS84 (EPSG:4326) to Mollweide projection with *bilinear* resampling method being used.

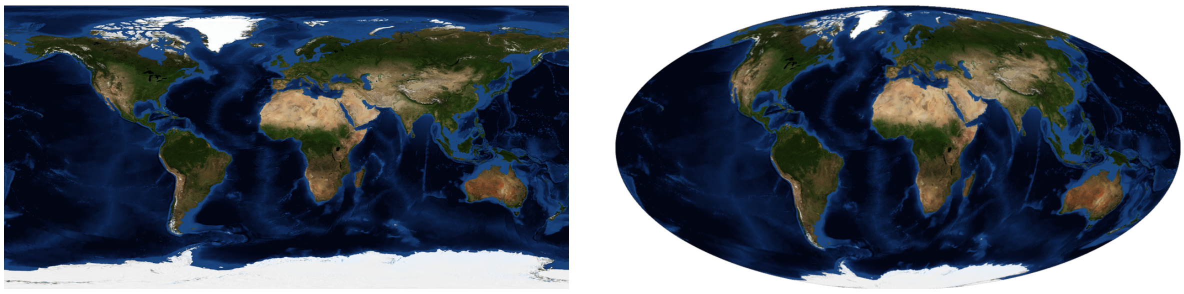

Figure 1: The *Blue Marble Next Generation* dataset [10] displayed in EPSG:4326 (left) and the Mollweide projection (right)

The `gdalwarp` utility (part of GDAL version 1.11.2) was used for verification and performance comparison with two different levels of parallelization. The following parameters were used for the first test: `gdalwarp -s_srs EPSG:4326 -t_srs +proj=moll -multi -wm 500` (the `-multi` argument enables parallelization of computation and IO operations; `-wm 500` increases allowed memory usage to 500 MB to increase performance). For the second test, `-wo NUM_THREADS=ALL_CPUS` was added to enable parallelization of the raster chunk processing up to the maximal number of threads the CPU can operate at once (equals to 4 on the testing PC).

Results from the `gdalwarp` and our implementation were compared using `idiff` utility. The outputs were per-pixel identical inside the bounds of the projection.

See Table 1 for a detailed comparison of execution times for different input and output sizes.

Results

Although the evaluation of the initial implementation shows certain performance gain when compared to `gdalwarp`, certain limitations can be observed.

Table 1: Performance testing of OpenCL warping in comparison to GDAL. Values in GDAL[†] are for parallelizing IO operations with computation, while values in GDAL[‡] are for parallelizing also the individual chunk processing. Times for the small dataset are averaged from 100 consecutive independent executions, for the other datasets from 5 executions.

	Input [px]	Output [px]	GDAL[†] [s]	GDAL[‡] [s]	OpenCL [s]	speedup
Small	1024×512	1024×512	0.2051	0.0992	0.2562	0.39×
		2048×1024	0.7054	0.2667	0.3756	0.71×
Large	21600×10800	1024×512	2.34	2.30	1.83	1.26×
		8192×4096	11.87	5.21	1.97	2.64×
		21600×10800	73.41	27.65	3.80	7.28×
Huge	86400×43200	8192×4096	173.15	172.29	168.28	1.02×
		21600×10800	198.64	191.16	172.17	1.11×
		86400×43200	980.82	865.37	528.68	1.64×

The computation time for the *small* dataset turned out to be actually worse than `gdalwarp`. This is caused by the fact, that the performance gain from OpenCL parallelization is smaller than the runtime kernel compilation overhead. The effect, however, would be less significant when more computations are required (complex transformations, postprocessing, etc. – see below) or for batch processing use cases (which would require only one kernel compilation for warping multiple datasets and/or extracting more extents).

Processing of the *huge* dataset also yielded no significant performance gain (for the smaller output sizes) since the time required for the loading of the input data and memory management is far longer than the processing time. Although the relative speedup of 1.64× (in the case of the largest output size) does not seem to be very significant, the absolute time difference is more than 5 minutes (14:24 vs 8:48), which can be very important in certain situations.

Overall, the memory transfers (CPU RAM $\leftrightarrow$ GPU VRAM) are the most time-consuming part of the process. Therefore, the performance gain of this parallelization approach is more significant for reprojections with more complex per-pixel calculations: mathematically complex transformation, resampling method, and possibly postprocessing in the future (noise reduction, color corrections, edge detection, . . .).

Conclusions and future work

In this paper, we have briefly described a possible approach to GPU-based per-pixel parallelization of raster map reprojection process. The experimental implementation has shown that there is a set of problems that can significantly benefit from GPU-based acceleration.

To fully utilize the potential of this parallelization method, the amount of raw computation needs to be as high as possible relative to the amount of data transfers. It is therefore most suitable for high output resolutions (same as the input resolutions) or even upscaling.

On the other hand, this approach is not very effective for small datasets, where the overhead of initializing the OpenCL environment is too high to have any actual benefit, or for significant downsampling, where the whole input needs to be copied to the GPU, which takes too long in comparison to the amount of the subsequent transformation calculations.

In the future implementations, the benefit of this parallelization approach can further increase with more complex transformations (e.g. warping based on *ground control points*) and resampling methods. Furthermore, various raster operations (e.g., noise reduction, color corrections, color mapping) can also be performed really fast both before and after the warping since the data are already present in the GPU memory.

Development of a tool for batch processing would also help better utilize the GPU potential – the OpenCL initialization could only be done once for multiple input and output datasets. Similarly, this parallelization approach could be beneficial for creating tile pyramids – input file would be loaded and transferred to GPU memory only once and used to create many smaller files.

Acknowledgements

This work has been done as part of Ph.D. research at Czech Technical University in Prague in cooperation with the Klokan Technologies GmbH for the future version of the MapTiler product (http://www.maptiler.com/).

References

[1] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture. A Hardware/-software Approach.* Morgan Kaufmann Publishers, Sept. 1998. ISBN: 9781558603431.

[2] G. Evenden, F. Warmerdam, et al. *PROJ.4 – Cartographic Projections Library.* [online]. http://proj.osgeo.org/. May 2015.

[3] M. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *Computers, IEEE Transactions on* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[4] C. F. Gauss, J. C. Morehead, and A. M. Hiltebeitel. *General investigations of curved surfaces of 1827 and 1825.* The Princeton University Library, 1902, p. 148.

[5] GDAL Development Team. *GDAL – Geospatial Data Abstraction Library, Version 1.11.2.* Open Source Geospatial Foundation. 2015. URL: http://www.gdal.org.

[6] W. D. Hillis and G. L. Steele Jr. “Data Parallel Algorithms”. In: *Communications of the ACM* 29.12 (Dec. 1986), pp. 1170–1183. ISSN: 0001-0782. DOI: 10.1145/7902.7903.

[7] Khronos OpenCL Working Group. *OpenCL C Specification.* Ed. by Lee Howes Aaftab Munshi and Bartosz Sochacki. 2015. URL: https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf.

[8] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1.* Ed. by Aaftab Munshi. 2011. URL: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[9] J. D. Owens et al. “A Survey of General-Purpose Computation on Graphics Hardware”. In: *Computer Graphics Forum* 26.1 (Mar. 2007), pp. 80–113. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2007.01012.x.

[10] R. Stöckli et al. *The Blue Marble Next Generation - A true color earth dataset including seasonal dynamics from MODIS*. Published by the NASA Earth Observatory. 2005. URL: http://visibleearth.nasa.gov/view.php?id=73751.