Accelerating and Predicting Map Projections with CUDA and MLP

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Jiaqi Zhang

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Thesis Committee

Dr. Han-Wei Shen, Advisor

Dr. Yusu Wang

Copyrighted by

Jiaqi Zhang

2018

Abstract

We provide a precise mathematical definition of map distortion, and introduce three strategies for finding the best oblique map projections, having the same distortion properties as their normal aspect counterparts on earth regions, that have minimal distortion for certain specific regions of the earth (namely, regions whose boundary consists of one or two parallels of latitude).

We use heap map to visualize the distortion of several unusually shaped countries under particular oblique projection applying our strategies, and then optimize the time efficiency of original sequential executed programs with CUDA, a GPU parallel computation method, impressively.

Acknowledgments

It's a great honor to have the opportunity to study and research at the Ohio State University. These years at OSU are one of the most precious periods in my life. I would like to thank my advisor Dr. Han-Wei Shen, the most crucial and kind person in my academic life. Besides teaching and sharing his knowledge and experiences about how to conduct research with me, he also cultivates my independent thinking ability and attitude on scientific research. Also, I would like to give my thanks to Dr. Alan John Saalfeld, who inspires me a lot and provides supports for our map projection research. Moreover, I want to show my respect and appreciations to those excellent professors in my department: Dr. Ten-Hwang Lai, Dr. Atanas Rountev and Dr. Deliang Wang. Last but not least, I want to thank my parents. They provide numerous financial and emotional support for me during my formative years.

Vita

2010 - 2014 B.S., Geographic Information System and Cartography, Wuhan University, China

2016 – present M.S., Computer Science and Engineering, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

People started creating all kinds of maps for thousands of years. A majority of those maps were created for military use and merely aimed to describe the comparative positions of different regions. However, people were unable to conduct any precise distance measurement due to the limited accuracy. Moreover, instead of a country or the entire world, most of those maps only cover limited regions.

The development of earth sciences as well as related disciplines promoted the improvement of cartography and helped people to understand the earth better. Currently, more and more cartographers start to focus on distortion research using Tissot's indicatrix, an ellipse evaluating and visualizing distortions of two principal directions at the center of an ellipse on the maps. Bad projected maps can mislead readers. For example, the high distortion around the polar regions in most world maps make people believe that Greenland is as wide as Africa, which is absolutely wrong.

To minimize the distortion, we need to choose a proper projection according to the target region's latitude. There are three kinds of prevalent projections: normal aspect azimuthal projection, normal aspect cylindrical projection, and normal aspect conic projection. Generally, cylindrical projection projects the equatorial regions; conic projection projects the mid latitude regions; and azimuthal projection projects the polar regions. Although those three normal aspect projections have been widely used for decades and suit perfectly for east-west elongated narrow countries, they are unable to project those countries covering vast territory well (e.g. the United States and China).

Since we already reached the limit of normal aspect projection, we must think out of the box and find a new way to minimize the distortion—the oblique projection. Benefiting from the flexibility of oblique projections, we can customize an ideal one according to the characters of given unshaped region.

In this thesis, we provide the definitions of map distortion and projection from the perspective of mathematics. After comparing the characters and features of traditional normal aspect projections versus revolutionary oblique aspect, we know that oblique projections tend to provide minimal distortion for any irregular region, and preserve the distortion properties when compare with their normal aspect counterparts. Although the geodetic graticules and projections of meridians and parallels may not be as recognizable as before, oblique projection can minimize the distortion markedly. By applying oblique projection, we can lower the distortion of Chile, a long narrow country, to 0.079%.

Compared with traditional Tissot's indicatrix, heat map can denote the value of a single point and visualize the separation of data directly and intuitively. Thus, we choose heat map to visualize the distortions of projected regions under the best fitting oblique projections found by applying our three strategies for oblique equidistant azimuthal, cylindrical, and conic projections.

The discontent with the efficiency of previous Python version programs about how it limits the application of real-time oblique projection had been general for a long time. CUDA introduces us to the world of GPU acceleration and parallelism. Compared to the previous sequential executed Python version code, it accelerates up to 98%, finishing in milliseconds. Starting with explanations of the fundamental and crucial data structures and functions used, we examine, compare, and analyze the intermediate and final running results of every projection at every step. Meanwhile, we directly draw the heat maps of a specific research region generated by two

different version programs under the same projection to argue and prove the correctness of CUDA version code. Also, we conduct additional research on the influence of different threads and blocks settings under different circumstances.

Chapter 2. Background

2.1 Map Projection

2.1.1 Scale and Distortion

The common notion of a map's scale is the ratio of distances on the map to distances on the ground. This layman's definition of "map scale as a single number" falls apart under closer mathematical scrutiny. This seemingly intuitive single number belies the fact there can never be a single ratio that works for all inter-point distances on a flat map depicting a region on round surface. There are always instead a range of scales; and if that range has both a non-zero minimum scale and a finite maximum scale, then we define the distortion to be the percentage increase in going from the minimum scale to the maximum scale.

If σmin is the minimum scale and σmax is the maximum scale, then

distortion = $[(\sigma max - \sigma min)/\sigma min] \times 100\%$.

The fact that σmin cannot equal σmax (and hence $\sigma min < \sigma max$) shows that distortion is always bigger than 0. In order to analyze distortion properly, we must carefully choose our definitions of map projections and scale in order to determine the maximum scale and the minimum scale of a projection. One observation about distortion is very important. Distortion does not depend on the size of the map in the following sense: If we photographically enlarge a map to make it X times higher and X times wider, called uniform scaling by X, then all scales in the new enlarged map will be X times larger. So the new map's minimum scale will be X times σmin and the new map's maximum scale will X times σmax; and the ratio of those two extreme scales will be

unchanged. So the enlarged map will have the identical distortion as the original map. Since uniform scaling does not change distortion, we may assume that our minimum scale is 1.

2.1.2 Map Projection

Let R, a region, be some subset of points on a sphere having non-empty interior. Then a map projection Π of R is a differentiable injective function Π from R to $\mathbb{R}^2$ such that Π has a differentiable inverse Π^{-1} defined on the image set Π(R).

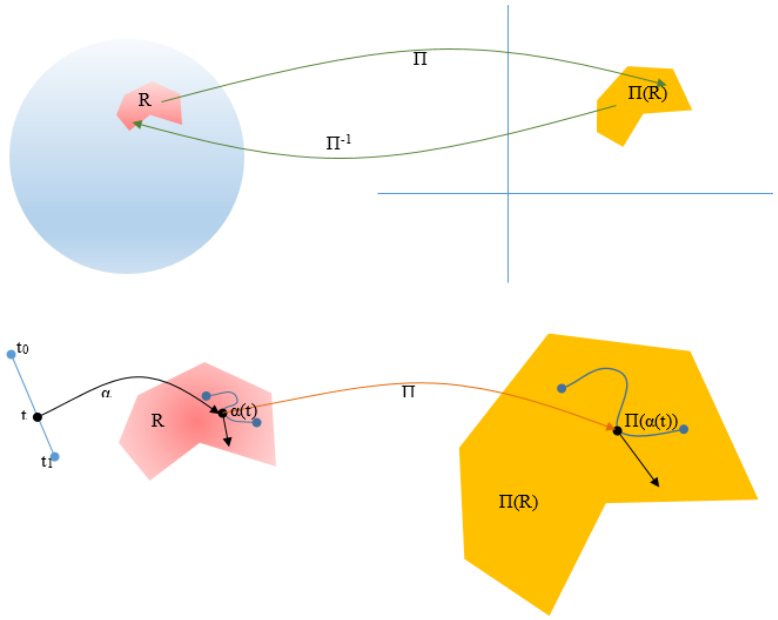


Figure 1. Map projections.

A projection Π maps a region of the sphere (or other datum surface) to a flat (or developable) projection surface. Any curve $\alpha:[t_0, t_1] \rightarrow R$ on the datum surface is transformed under the projection to a curve $\Pi\circ\alpha:[t_0, t_1] \rightarrow \Pi(R)$ on the projection surface. The velocity tangent vector α'(t) to the curve on the datum surface has a corresponding velocity tangent vector $(\Pi\circ\alpha)'(t) = d[(\Pi\circ\alpha)(t)]/dt$ to the transformed curve on the projection surface. The curve speeds at α(t) and $(\Pi\circ\alpha)(t)$ are just the norms of the tangent vectors: $\|\alpha'(t)\|$ for the datum surface and $\|d[(\Pi\circ\alpha)(t)]/dt\|$ for the projection surface.

The scale at any point $\alpha(t)$ on the curve in the velocity tangent direction $\alpha'(t)$ is the ratio of the two speeds: $\|d[(\Pi \circ \alpha)(t)]/dt\|/\|\alpha'(t)\|$. From differential geometry, we know that this ratio does not depend on our choice of curve α. Any other curve passing through the point and having the same tangent vector direction will produce the same ratio. For any point there are scales in every direction; and we can always construct a curve passing through the point in any given direction and use that curve to find the scale. At any point there are four scales of special interest. It is common practice to use the letters **h** and **k** to represent scales along the meridian and parallel curves, respectively. In other words, **h** denotes the scale in the north-south directions, and **k** denotes the scale in the east-west directions. Since at any point there is a scale in every direction, it is conventional to use the letters **a** and **b** to denote the maximum and minimum scales at the point, respectively. For a large group of projections called Normal Aspect Projections, the values **h**, **k**, **a**, and **b** have a very important relation: as sets, **{h, k} = {a, b}**. Stating this another way, if we look at the north-south scale **h** and at the east-west scale **k**, one of them will be the maximum scale **a**, and the other will the minimum scale **b**. Note that at a point, **a** and **b** do not have to be different. If the maximum scale **a** at a point equals the minimum scale **b** at that point, then all scales in every direction at that point are the same: **a = h = k = b**.

One important class of Normal Aspect Projections that minimize distortion are the Equidistant Normal Aspect Projections.

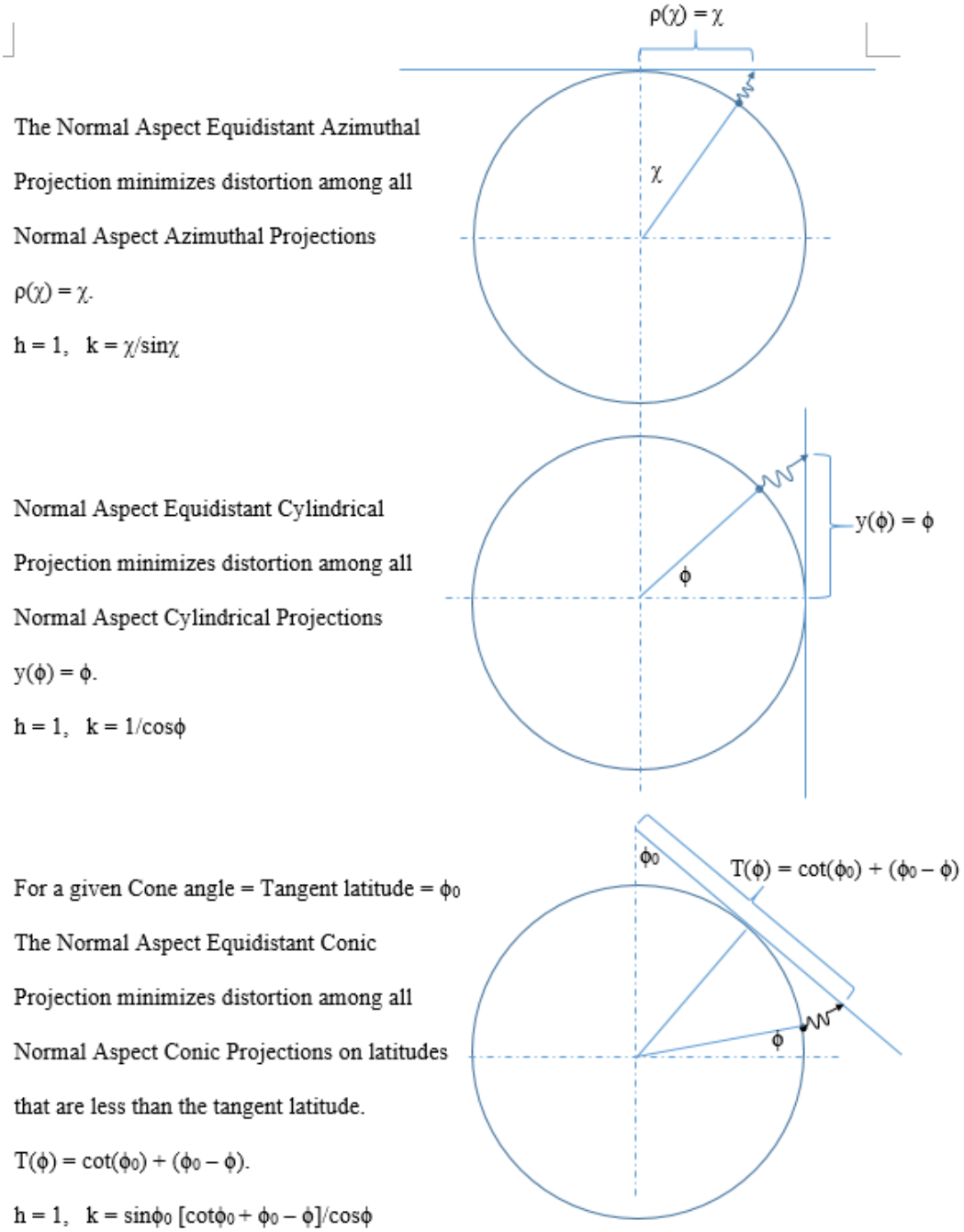


Figure 2. Equidistant normal aspect projections' scales.

2.1.3 Comparison of Normal Aspect Projection VS. Oblique Aspect

For every oblique projection of the sphere, there is a point P_0 on the sphere (at latitude φ_0 and at longitude λ_0) that corresponds to the North/South Pole at (0, 0, ±R) for Normal Aspect Projections.

Table 1. Comparison of normal aspect projection VS. oblique aspect.

	Normal Aspect Projection	**Oblique Aspect Projection**
Orientation point (OP)	North/South Pole: NP = (0, 0, ±R)	P_0 = (R cosφ_0 cosλ_0, R cosφ_0 sinλ_0, R sinφ_0)
Projection surface axis	Polar axis	Line passing through P_0 and -P_0
Geodesics from OP	**M**=Meridian curves	**Γ**=Great (semi-)circles starting at P_0
Points at same distance to OP	**Π**=Parallels of latitude	**Λ**=Lesser circles with centers on line (P_0, -P_0)
Curves in principal directions	Graticule = **M** **Π**	**Γ** **Λ**

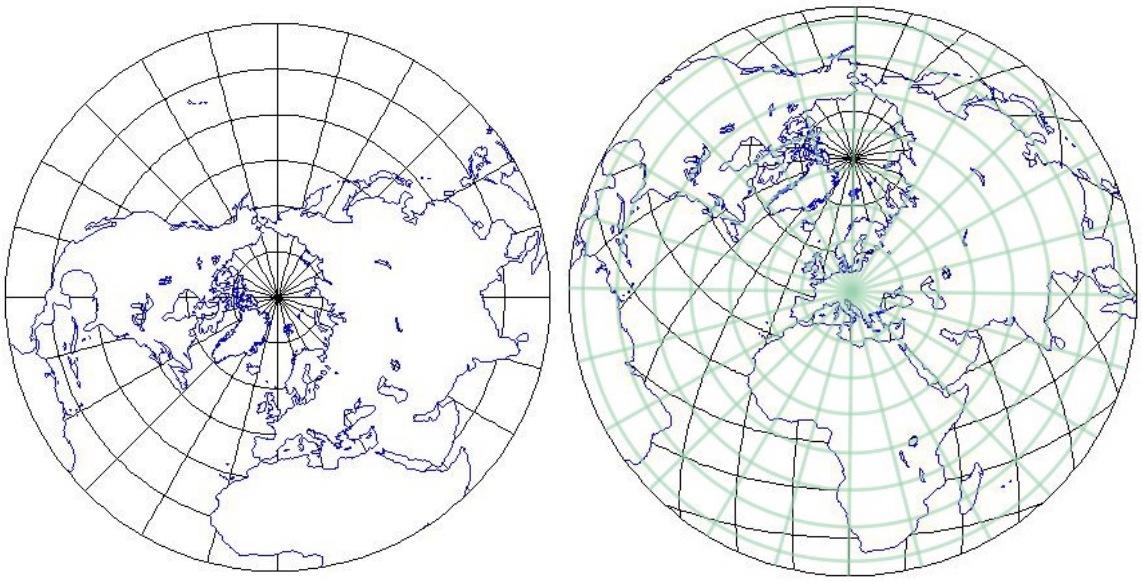


Figure 3. Normal VS. oblique aspect azimuthal projected map (P_0 = (16° E, 48° N)).

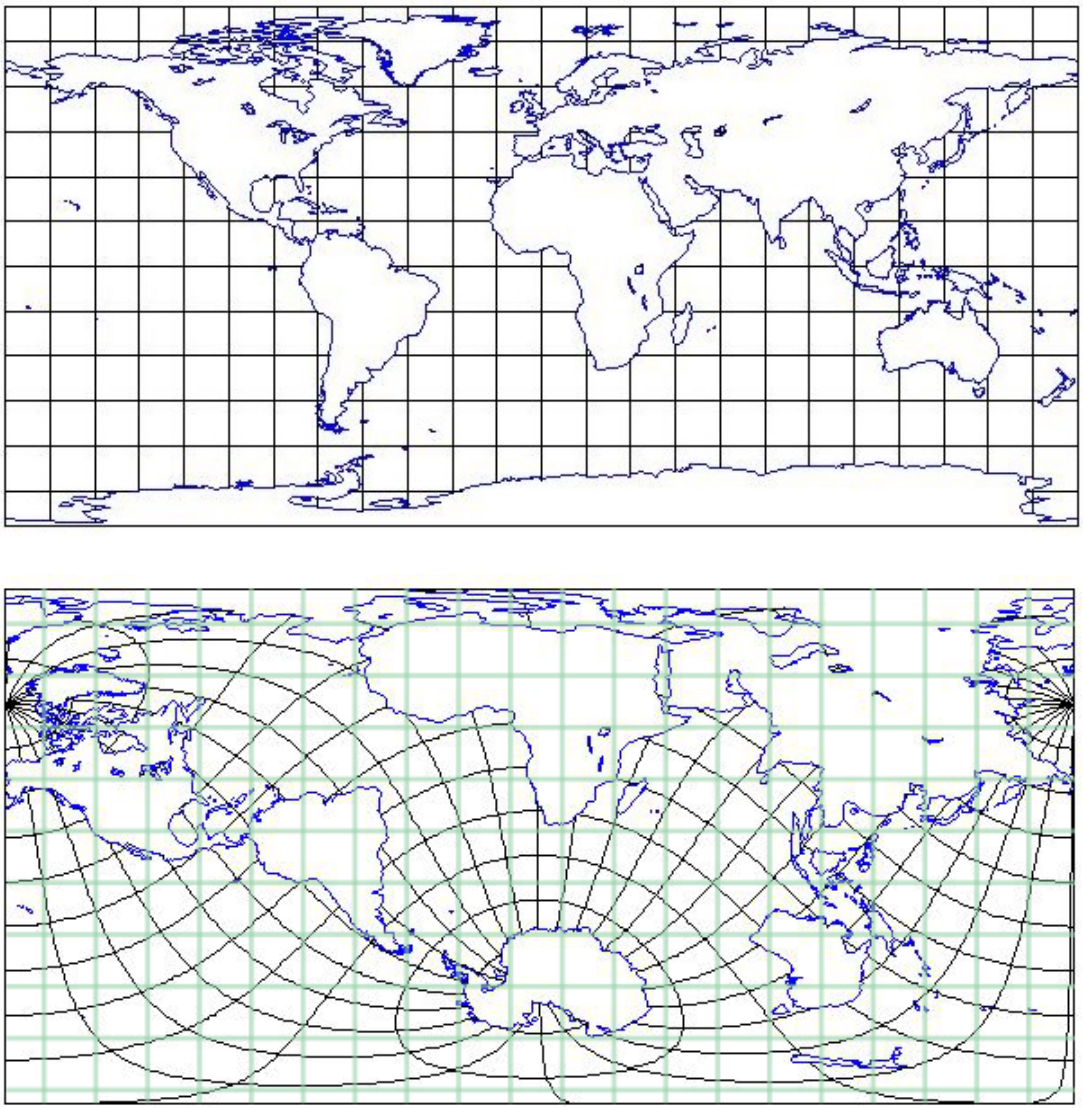


Figure 4. Normal VS. oblique aspect cylindrical projected map (P_0 = (16° E, 48° N)).

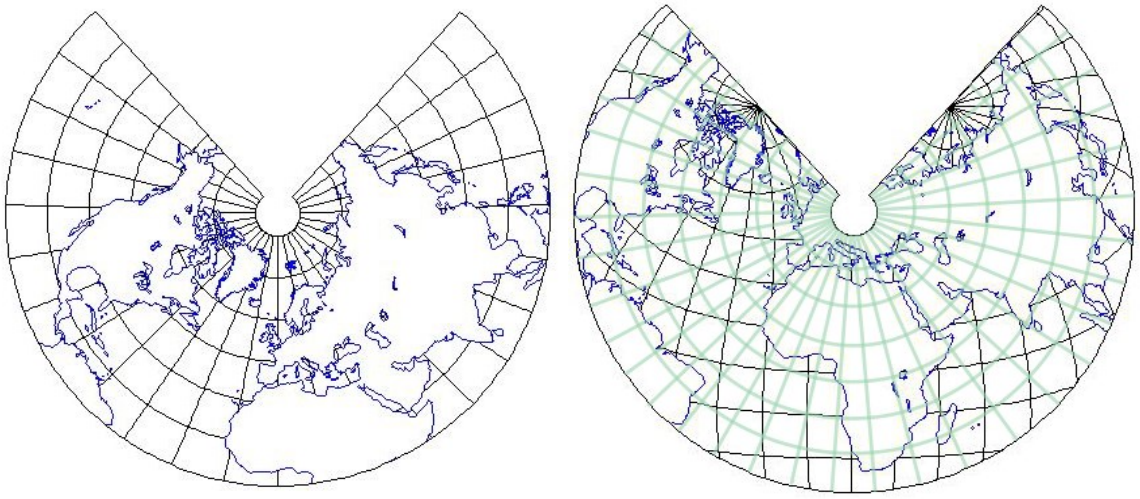


Figure 5. Normal VS. oblique aspect conic projected map (P_0 = (16° E, 48° N)).

Figure 3, Figure 4, and Figure 5 provide us a direct and intuitive way to understand how the meridians and parallels look on different projections. The upper layer light mint green graticules represent the projected great circles and lesser circles passing through the orientation point P_0, following the same pattern of normal aspect projection's graticules on the left. The bottom layer black graticules are deformed traditional meridians and parallels under oblique projections, not necessarily perpendicular to each other anymore. Among three different oblique projections above, the deformation of the oblique cylindrical projection is largest as one moves the same distance away from the tangent point or circle.

2.2 Parallel Computing

Parallel computing is a type of high-performance computation in which many calculations or execution of processes are carried out simultaneously. Typically, a parallelizable computational task is divided into several very similar subtasks that can be processed independently and then combined after completed. It can be classified as bit-level, instruction-level, data, and task parallelism.

Optimally, the speedup from parallelization would be linear, doubling the number of processors will halve the running time. However, in reality it follows near-linear Amdahl's law when the numbers of processors are small, and then reaches a constant value for overlarge numbers of processors.

$$S_{Latency}(\mathrm{s}) = \frac{1}{1 - p + \frac{p}{s}}$$

where

- $S_{Latency}$ is the theoretical speedup in latency of the whole task
- s is the speedup in latency of the parallelizable part
- p is the proportion of execution time that the part benefiting from parallelism

From Amdahl's law, we can easily find the theoretical speedup is limited by the serial part of the task due to data dependencies. For example, if 95% of the task is parallelizable, the theoretical maximum speedup is 20 times.

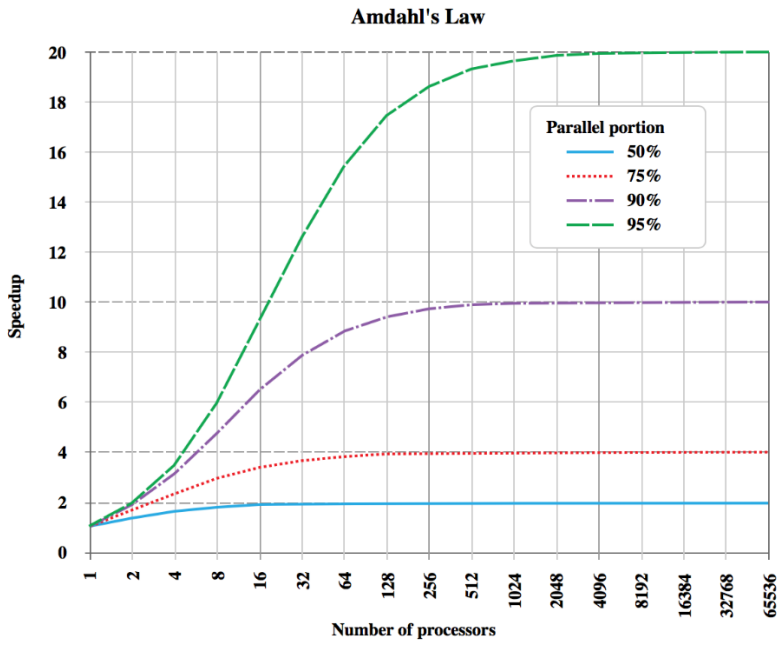


Figure 6. Amdahl's law.

2.3 GPU-Accelerated Computing

GPU-accelerated computing is the use of a massively parallel architecture GPU, consisting of thousands of smaller, more efficient cores, and a few cores CPU to accelerate original sequential executed programs. GPU-accelerated computing offloads compute-intensive portions of the program to the GPU, while the remaining code still runs on the CPU. Nowadays, it is widely used to accelerate deep learning, artificial intelligence, and other compute-intensive fields.

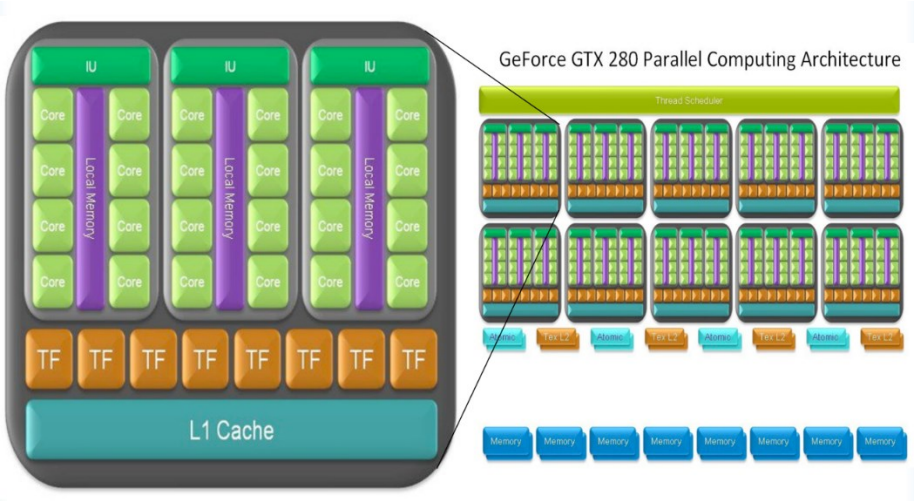


Figure 7. GPU architecture.

2.4 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic and impressive acceleration in computing performance with GPU. The following table shows how CUDA improves the performance up to 50 times radically. Started from 2006, the CUDA ecosystem had grown rapidly; included software development tools, services, and partner-based solutions. The CUDA Toolkit provides libraries, debugging and optimization tools, a compiler, and a runtime library for developers. Moreover, thousands of

applications developed with CUDA have been deployed to GPUs in embedded systems, workstations, data centers and in the cloud.

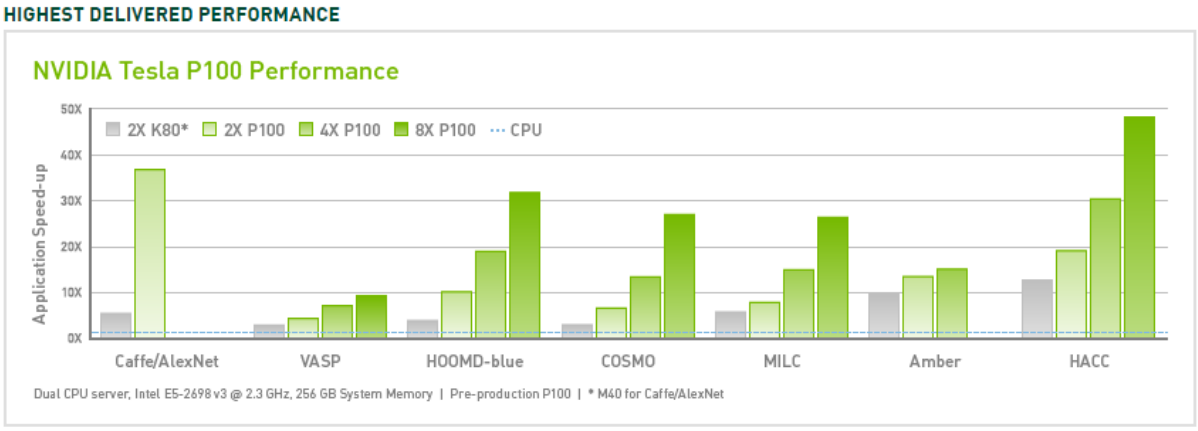


Figure 8. NVIDIA Tesla P100 performance.

The latest released CUDA 9, built for Volta GPUs, includes up to five times faster GPU-accelerated libraries, new programming model for flexible thread management with cooperative groups, and improvements to the compiler and developer tools.

2.5 Visualization of Distortions

2.5.1 Tissot's Indicatrix

Traditionally, most cartographers prefer to use Tissot's indicatrix (Tissot's ellipse), a mathematical contrivance invented by Nicolas Anguste Tissot in 1859, to characterize local distortion on a map.

Tissot's indicatrix is a geometry resulting from projecting a circle of infinitesimal radius from a curved geometric model to projected plane. Tissot proved that the projected shape is an ellipse, whose axes indicate two principle directions along which scale is maximal and minimal at the center of ellipse on the map.

A single Tissot's indicatrix describes the distortion at a particular point, thus cartographers normally place them at the intersections of meridians and parallels across a map to illustrate the spatial change in distortion.

Figure 9 is a normal aspect equidistant cylindrical projected world map. The directions of indicatrix's major and minor axis indicate that the principle directions under this projection are along meridians and parallels. Also, the lengths of the ellipses' semi minor axis are all the same, indicating the scale is fixed along the same direction. Moreover, the lengths of semi major axis are increasing non-linearly from the equatorial to polar regions, indicating the huge distortion in high latitude regions. Clearly demonstrate all the characteristics of a projection is the reason why Tissot's indicatrix is widely used for so many years.

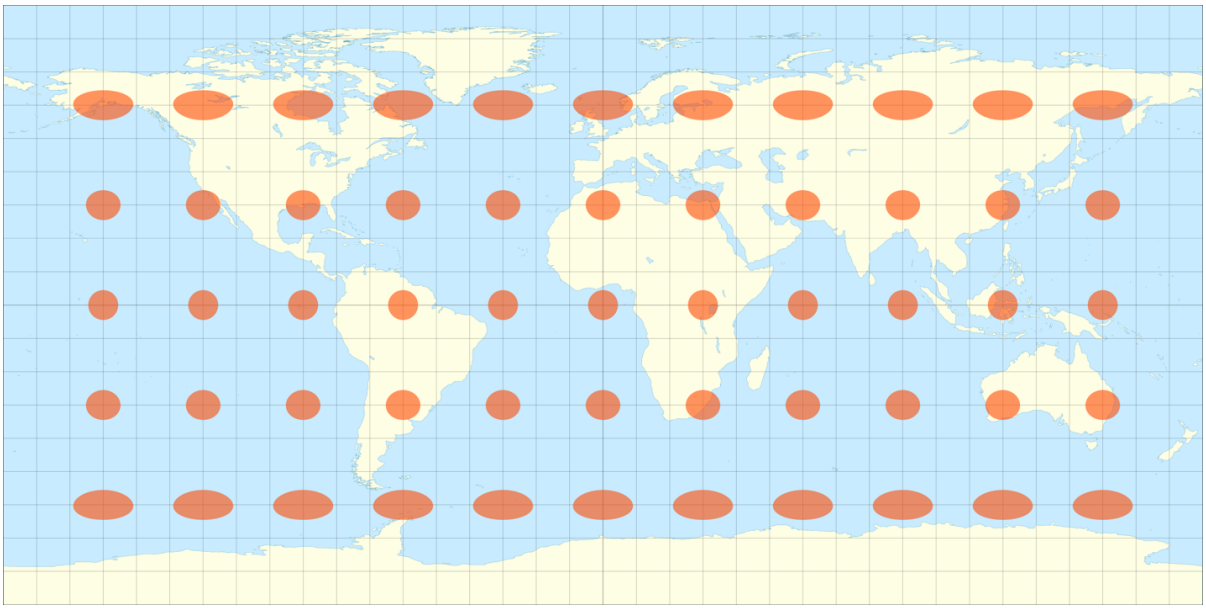


Figure 9. Tissot's indicatrix.

2.5.2 Heat Maps

A heat map is a graphical representation of data, representing the individual values of a matrix with colors or gray level. Since humans can distinguish more shades of color than they can of gray, we tend to use color schemes like rainbow color map.

Unlike Tissot's Indicatrix, heat map represents the distortion of a continuous region on the map instead of several distinct points. However, heat map can only visualize a single value in a location without any directional information, which limit its use in projection. Since this research only focus on equidistant projections, h = 1 at any point, we can merely show the scale of k on the heat map.

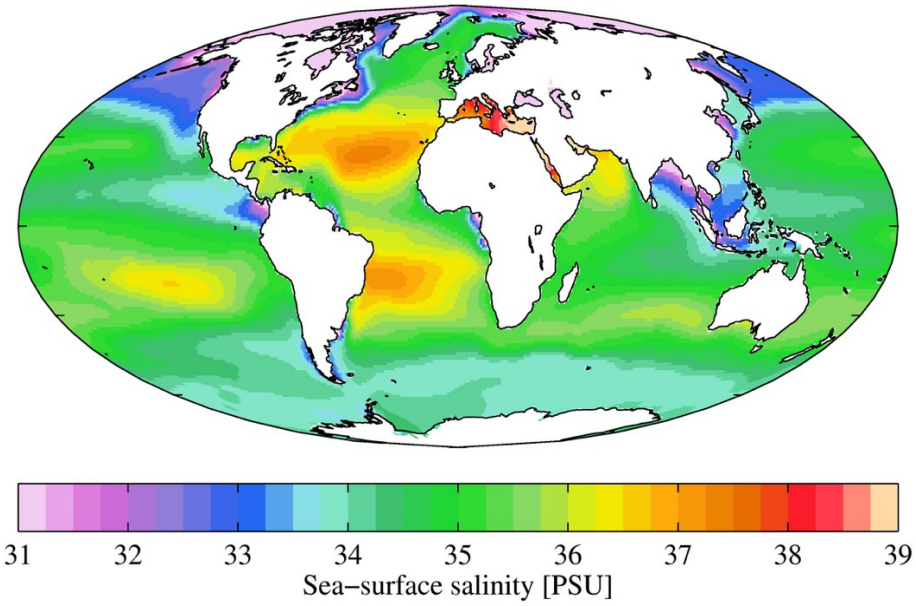


Figure 10. Heat map of sea-surface salinity.

Chapter 3. Method

3.1 Data Structure

Since the data size of input varies with the number of boundary points for a particular region, I choose vectors, dynamic length containers, to store the inputs and outputs instead arrays. It is well known that if we want to define an array, we have to define the length of it or at least the length of last dimension.

Furthermore, when recording the intermediate results in the program, we have to store the direction vector, integer, and even string variables in a single array; nevertheless it is obviously prohibited for array in C/C++. As a result, we turn to vector (vector<type>) and vector of vectors (vector<vector <type>>), which allows more flexibility to developers.

However, CUDA does not support vector container, therefore we have to transfer from vectors to arrays before we process them in GPU. We use a[][N] or a[][][N] (N must be a constant representing the column number) to define a 2D or 3D array.

As a matter of fact, we can also use an one dimension array to simulate and store a 2D array by calculating the index: a[x] can represent a[x/(length of a row)][x%(length of a row)]. To facilitate research, I choose 2D array.

3.2 Oblique Projections

3.2.1 Oblique Equidistant Azimuthal Projection

For circular regions on the sphere, Milnor (1969) has proved that the projection that minimizes distortion is the azimuthal equidistant projection tangent at the center of the

circular region [1]. According to my research, we solved this problem using vectors' dot products. Below is the pseudo-code.

Step1: Generate candidate directional vectors in a neighborhood N with density D.

Step2: For each candidate vector, calculate n dot products of all the n boundary points, and pick the minimum of those n dot products to represent that candidate vector.

Step3: Pick the vector having the largest representative minimum dot product.

Step4: If this vector improves the maximum dot product results significantly, refine N and D in Step1.

Step5: If the representative minimum stabilizes (converges), terminate and print the result.

For the first iteration candidate vectors generation, I sampled all the possible vectors in every direction, which should be in every 5 degrees along the Longitude and Latitude, over all 2592 vectors. Since the Earth is a continuous surface and the dot product is a continuous function in both variables, the change of the maximum dot product should be small when we move the candidate vector slightly. In other words, from the second iteration the new generated candidate vectors taken with greater density D from a small neighborhood N around the best vector from the last iteration will home in on a vector that maximizes its minimum dot product step by step. Under our pre-set tolerance, we will finally get a vector maximizing dot product.

In the research, we suppose the Earth is a sphere and set its radius to 6371.00 Kilometers. All the Geographic and Cartesian coordinates of any point on the Earth are based on those 2 assumptions.

3.2.2 Oblique Equidistant Cylindrical Projection

For oblique cylindrical projection, the equidistant and the Mercator are two extremes of minimal distortion projections. For the equidistant, h = 1; for the Mercator, h = k = $1/\cos\phi$; and in either case, the minimum scale will be 1 and the maximum scale will be $1/\cos\phi_{max}$. If we choose the equidistant projection, we can preserve the distance along meridians; but if we choose Mercator, we can reduce distortion within small subregions away from the equator. For the sake of convenience, we decide to research on equidistant cylindrical projections. Below is the pseudo-code.

Step1: Generate candidate directional vectors in a neighborhood N with density D.

Step2: For each candidate vector, calculate n dot products of all the n boundary points, and use the projected range (maximum subtracts minimum of those n dot products) to represent that candidate vector.

Step3: Sort the representative projected range of each candidate vector in a nondecreasing order.

Step4: Pick the vector having the minimum projected range under the constraint that the mid of it is as close to 0 as possible (less than preset tolerance).

Step5: If this vector improves the minimum projected range significantly, refine N and D, then return to Step1.

Step6: If the directional vector stabilizes (converges), terminate and print the result.

Here, we follow the same strategies of oblique azimuthal projection to generate candidate vectors.

3.2.3 Oblique Equidistant Conic Projection

The normal aspect conic projection tangent at latitude ϕ_0 that has minimum distortion is consisted of two different projections glued together at the tangent parallel: the minimum

distortion projection is the equidistant conic projection for latitudes below ϕ_0, and is the conformal conic projection for latitudes greater than ϕ_0. Similarly，as a matter of convenience, we focus our research on oblique equidistant conic projection. Below is the pseudo-code.

Step1: Generate candidate directional vectors in a neighborhood N with density D.

Step2: For each candidate vector, calculate n distortions of all the n boundary points, and pick the maximum of those n distortions to represent that candidate vector.

Step3: Pick the vector having the smallest representative maximum distortion.

Step4: If this vector improves the maximum distortion results significantly, refine N and D, then return to Step1.

Step5: If the representative maximum stabilizes (converges), terminate and print the result.

Here, we follow the same strategies of oblique azimuthal projection to generate candidate vectors.

3.3 Realization in CUDA

Read ArcGIS 10.3 preprocessed boundary points' Longitude/Latitude information with C++ standard Input/Output.

```
//Read Data
ifstream i_stream;
i_stream.open("C:\\Users\\Zhang\\iCloudDrive\\Projection Research\\CudaData\\Cuda_China_Bd.txt", ios::in);
assert(i_stream.is_open());
vector<double> Lam;
vector<double> Phi;
string s;
while (getline(i_stream, s))
{
    //cout << s << endl;
    vector <string> eachline;
    stringstream sstr(s);
    string token;
    while (getline(sstr, token, ','))
    {
        eachline.push_back(token);
    }
    Lam.push_back(atof(eachline[1].c_str()));
    Phi.push_back(atof(eachline[2].c_str()));
}
i_stream.close();
```

Figure 11. Read data from the input file.

Then create vector of vectors to save all the Cartesian coordinates of boundary points transferred from geodetic coordinates.

```
//Transfer to x,y,z coordinates Bd=[x,y,z]
vector <vector<double> >Bd;

for (int i = 0; i < Lam.size(); i++)
{
    vector<double> tmp;
    tmp.push_back(R*cos(Phi[i])*cos(Lam[i]));
    tmp.push_back(R*cos(Phi[i])*sin(Lam[i]));
    tmp.push_back(R*sin(Phi[i]));
    Bd.push_back(tmp);
    tmp.clear();
}
```

Figure 12. Coordinates transformation.

After applying our algorithms, we can get the directional vector for the best oblique projection for the input country. To realize a heat map, we need two steps. First, project all the boundary points according to corresponding projection theories and formulas from the sphere to the projected surface, geodetic to Cartesian coordinates. Then calculate the

distortion value, the heat map value in that position, for all the sampled points under this projection. Before using distortion calculation formulas, we should first transfer from Cartesian coordinates on the projected plane to geodetic coordinates on the sphere.

In the program, I define variable-length pointer arrays with fixed column numbers to save the results, use cudaMalloc() to allocate memory space on device (GPU), and cudaMemcpy() to copy data from host memory (CPU) to allocated corresponding device memories. And the last parameter marks the direction of memory copy, cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost.

```
//Result matrix
double(*host_BLamPhiNew)[2] = new double[HeatLam.size()][2];
double(*host_BXYDist)[3] = new double[HeatLam.size()][3];

//printf("Start to malloc device memory...\n");
double *dev_HeatLam, *dev_HeatPhi, (*dev_BLamPhiNew)[2], (*dev_BXYDist)[3];
cudaMalloc((void **)&dev_HeatLam, HeatLam.size() * sizeof(double));
cudaMalloc((void **)&dev_HeatPhi, HeatPhi.size() * sizeof(double));
cudaMalloc((void **)&dev_BLamPhiNew, HeatLam.size() * 2 * sizeof(double));
cudaMalloc((void **)&dev_BXYDist, HeatLam.size() * 3 * sizeof(double));

//printf("Start to copy host memory data to device memory...\n");
cudaMemcpy(dev_HeatLam, &HeatLam[0], HeatLam.size() * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(dev_HeatPhi, &HeatPhi[0], HeatPhi.size() * sizeof(double), cudaMemcpyHostToDevice);
```

Figure 13. Allocate memory space on GPU.

Since the access time of the memories on the same block is much faster than in other blocks, I allocate more threads in every single block instead of increasing block numbers. And the block number is up to 16 according to the density of input data.

```
int threadPerBlock = 256;
int blockPerGrid = MIN(16, (HeatLam.size() + threadPerBlock - 1) / threadPerBlock);
```

Figure 14. Program for threads and blocks settings.

The qualifier __global__ and __device__ before the functions' name is used to alert the compiler that they should be compiled to run on a device (GPU) instead of on the host (CPU), while qualifier __device__ functions will be callable only from other __device__ functions or from __global__ functions. For example, CoorRotateGPU() is used to conduct coordinates rotation to get the new geodetic coordinates under new best fitting directional vector, and ProjRangeGPU(), called by other global functions frequently, is used to calculate the projected distance through dot products.

```
__global__ void CoorRotateGPU(double Lam[], int LamSize, double F_Lam, double Phi[], double F_Phi, const double R,double GeoCoord[][2])
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    const int offset = gridDim.x * blockDim.x;

    while (index < LamSize)
    {
        double F_Theta = M_PI / 2 - F_Phi;

        double x_old = R*cos(Lam[index])*cos(Phi[index]);
        double y_old = R*sin(Lam[index])*cos(Phi[index]);
        double z_old = R*sin(Phi[index]);

        double Gama = -M_PI / 2 + F_Lam;
        double Alfa = -F_Theta;

        double x_new = x_old*cos(Gama) + y_old*sin(Gama);
        double y_new = -x_old*cos(Alfa)*sin(Gama) + y_old*cos(Alfa)*cos(Gama) + z_old*sin(Alfa);
        double z_new = x_old*sin(Alfa)*sin(Gama) - y_old*sin(Alfa)*cos(Gama) + z_old*cos(Alfa);

        double lam_new = atan2(y_new, x_new);
        double phi_new = asin(z_new / R);

        GeoCoord[index][0] = lam_new;
        GeoCoord[index][1] = phi_new;


        index += offset;
    }


}
```

```
__device__ double ProjRangeGPU(double V[5], double Bd[][3], int BdSize, double R)
{
    double Low_B = (Bd[0][0] * V[0] + Bd[0][1] * V[1] + Bd[0][2] * V[2]) / R;
    double Upper_B = (Bd[0][0] * V[0] + Bd[0][1] * V[1] + Bd[0][2] * V[2]) / R;
    double range_result = 0.0;
    for (int i = 0; i < BdSize; i++)
    {
        double dotpro = (Bd[i][0] * V[0] + Bd[i][1] * V[1] + Bd[i][2] * V[2]) / R;
        if (dotpro < Low_B) Low_B = dotpro;
        else if (dotpro > Upper_B) Upper_B = dotpro;
    }
    //range_result = Upper_B - Low_B;

    return Low_B;
}
```

Figure 15. Functions running on GPU.

After finishing all the calculation on GPU, still with cudaMemcpy(), we can copy the results from device memories to host memories. For the convenience of the following computations, I still restore them as vectors.

```
cudaMemcpy(host_BLamPhiNew, dev_BLamPhiNew, HeatLam.size() * 2 * sizeof(double), cudaMemcpyDeviceToHost);

vector<double> BLam_New, BPhi_New;
for (int i = 0; i < HeatLam.size(); i++)
{
    BLam_New.push_back(host_BLamPhiNew[i][0]);
    BPhi_New.push_back(host_BLamPhiNew[i][1]);
}
```

Figure 16. Copy results to CPU memory.

Last step is to free all the used allocated memories on the device as we did on the host.

```
delete[]host_HeatVal;
cudaFree(dev_HeatSample); cudaFree(dev_HeatVal);
```

Figure 17. Free allocated GPU memories.

For performance evaluation, although we could use CPU or operation system timers, this will include latency and variation from many sources, like operation system thread scheduling, availability of high-precision CPU timers. Moreover, when the GPU kernel runs, we might be asynchronously performing computation on the CPU at the same time, which will be included by the CPU timber mistakenly. Thus, CUDA provides event API to conduct performance evaluation efficiently and reliably.

To time a block of code, cudaEventCreate() and cudaEventRecord() can create start and stop events together as time stamps to record the running time on GPU. For the aim of synchronization, cudaEventSynchronize() instructs the CPU to synchronize on an event

and makes sure the runtime to block further instructions on the CPU until the GPU has reached the finishing time stamp. After it completed the process, cudaEventElapsedTime() is used to compute the elapsed time between two recorded time events and cudaEventDestroy() to free used allocated CUDA events.

```
cudaEvent_t HMStart, HMFinish;
float HMCostTime;
cudaEventCreate(&HMStart);
cudaEventCreate(&HMFinish);
cudaEventRecord(HMStart, 0);

cudaEventRecord(HMFinish, 0);
cudaEventSynchronize(HMFinish);
cudaEventElapsedTime(&HMCostTime, HMStart, HMFinish);
printf("CUDA Running Time of Calculating Heat Map in Program_HeatMapCon: %f (ms)\n\n", HMCostTime);
HM_Con_RunTime.push_back(HMCostTime);
cudaEventDestroy(HMStart);
cudaEventDestroy(HMFinish);
```

Figure 18. Event functions for time recording.

To guarantee the accuracy and reliability of the evaluation process, I warm up programs by iterating 20 times before start recording the running time of each iteration, and then take the average value as the final running time for that program.

```
//Warm up the program before testing
for (int i = 0; i < Loop; i++)
{
    vector < vector<double>> warmup_FV = Dynamic_Conic(fPath, R, Con_Phi0LoopNum, Con_Thre, 256, 16);
}

////////////////////////////////////////////////////////////////////////////////////////////
//Eval F_V
//ThreadsNum: 256   BlocksNum: 16
double Sum = 0.0;
for (int i = 0; i < Loop; i++)
{
    vector<vector<double>> fvRes = Dynamic_Conic(fPath, R, Con_Phi0LoopNum, Con_Thre, 256, 16);
    Sum += fvRes[1][fvRes[1].size() - 1];
}
printf("Average Running Time with %d ThreadsPerBlock and %d BlocksPerGrid Is: %f (ms)\n\n", 256, 16, Sum / Loop);
```

Figure 19. Functions for running time evaluation.

I also test all the combinations of resolutions of heat map, numbers of threads per block, and numbers of blocks per grid to find their impacts on the performance.

```
//Changing HeatMap Resolution, BlocksNum, and ThreadsNum
vector<double> All_HMRunTime;
for (int k = 0; k < sizeof(Resolution) / sizeof(double); k++)
{
    for (int m = 0; m < sizeof(Blocks_Num) / sizeof(int); m++)
    {
        for (int n = 0; n < sizeof(Threads_Num) / sizeof(int); n++)
        {
            double Sum = 0.0;
            for (int i = 0; i < Loop; i++)
            {
                vector<vector<double>> hmRes = HM_Con(fPath, FVforHM[0], R, MScale, MeriGridWid, Resolution[k], Threads_Num[n], Blocks_Num[m]);
                Sum += hmRes[0][hmRes[0].size() - 1];
            }
            All_HMRunTime.push_back(Sum / Loop);
        }
    }
}
```

Figure 20. Functions for threads/blocks number evaluation.

Finally, we should customize the code to make sure it suits for different oblique projections following the same procedures and similar key functions.

Chapter 4. Results and Discussions

4.1 Correctness

4.1.1 Oblique Equidistant Azimuthal Projection

All the programs are running on the same computer with CPU i5-6300HQ, GPU GTX960M, Python 2.7 32bits, and Windows 10 Home 64bits. The testing data is the simplified boundary of mainland China. For CUDA version programs, all of them are set to maximum 256 threads per block and 16 blocks per grid.

Programs for finding the best fitting directional vector:

1. Python version running results.

```
Input Data: China_Bd.txt
--------------------------------------------------------------------------------
Python Version Code for Finding the Best Fitting Oblique Equidistant Azimuthal Proj:

1th Iteration for Finding Direcional Vector...
Running time for Calculate Projected Range: 200.0 (ms)
Best Directional Vector Is: [-1263.1583643075858, 4714.171193601506, 4095.1998613129417, 1.8325957145940461, 0.6981317007977318]
Maximum Lower Bound: [5819.31359313, 6368.34579891]  (KM)

2th Iteration for Finding Direcional Vector...
Running time for Calculate Projected Range: 15.0 (ms)
Best Directional Vector Is: [-1163.2221001622574, 4665.429020798304, 4179.752073698522, 1.8151424220741028, 0.7155849933317675]
Maximum Lower Bound: [5856.71625706, 6370.33802433]  (KM)

3th Iteration for Finding Direcional Vector...
Running time for Calculate Projected Range: 16.0 (ms)
Best Directional Vector Is: [-1150.4028563729844, 4683.601457299649, 4162.942668020962, 1.8116517635701144, 0.7120943348136863]
Maximum Lower Bound: [5863.51273794, 6369.92123392]  (KM)

Final Directional Vector's Long/Lat: [103.8, 40.8]
Final Directional Vector Is: [-1150.4028563729844, 4683.601457299649, 4162.942668020962, 1.8116517635701144, 0.7120943348136863]
Final Lower/Upper Bound(Positive): [5863.51273794, 6369.92123392]  (KM)


Fianl Distortion Range: [ 0%, 2.74278905074% ]

Maximum Angle of the Boundary Point is: 23.0235500849 degrees
The Tangent Ponit: ID:0 [109.8410034, 18.3680992]

Overall running time for Program_Azi is: 279.0 (ms)
```

Figure 21. Running results for finding directional vector under oblique azimuthal projection (Python).

2. CUDA version running results:

```
CUDA Version Code for Finding the Best Fitting Oblique Equidistant Azimuthal Proj...

1th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 6.366208 (ms)
CUDA Best Direcional Vector Is: [-1263.158364, 4714.171194, 4095.199861, 1.832596, 0.698132]
CUDA Maximum Lower Bound: [5819.313593, 6368.345799] (KM)

2th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 2.432512 (ms)
CUDA Best Directional Vector Is: [-1163.222100, 4665.429021, 4179.752074, 1.815142, 0.715585]
CUDA Maximum Lower Bound: [5856.716257, 6370.338024] (KM)

3th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 2.454496 (ms)
CUDA Best Directional Vector Is: [-1150.402856, 4683.601457, 4162.942668, 1.811652, 0.712094]
CUDA Maximum Lower Bound: [5863.512738, 6369.921234] (KM)

CUDA Final Directional Vector's Long/Lat: [ 103.800000, 40.800000]
CUDA Final Directional Vector Is: [-1150.402856, 4683.601457, 4162.942668, 1.811652, 0.712094]
CUDA Final Maximum Lower Bound(Positive): [ 5863.512738, 6369.921234]
CUDA Overall Running Time for Finding Best Directional Vector: 157 (ms)
```

Figure 22. Running results for finding directional vector under oblique azimuthal projection (CUDA).

After comparison, I found that both intermediate running results in every iteration as well as the final directional vector's longitude and latitude are the same, which represent the correctness of CUDA version program.

Programs for generating heat map:

1. Python version running results.

```
Heat Map of Distortion Under Oblique Equidistance Azimuthal Proj:

Running time of Projecting Boundary Points in Program_HeatMapAzi: 0.0 (ms)

Running time of Projecting Meridian/Parallels Points in Program_HeatMapAzi: 0.0 (ms)

Size of HeatMap Sampled Points: 2218181

Running time of Calculating Heat Map in Program_HeatMapAzi: 2606.0 (ms)

Overall running time for Program_HeatMapAzi with Resolution 0.05 cm is: 4127.0 (ms)
```

Figure 23. Running times for heat map under oblique azimuthal projection (Python).

2. CUDA version running results.

```
CUDA Version Heat Map of Distortion Under Oblique Equidistance Azimuthal Proj...

threadPerBlock: 256, blockPerGrid: 1
CUDA Running Time of Projecting Boundary Points in Program_HeatMapAzi: 0.683040 (ms)

MeriThreadPerBlock: 256, MeriBlockPerGrid: 1
CUDA Running Time of Projecting Meridian/Parallels Points in Program_HeatMapAzi: 0.529152 (ms)

Size of HeatMap Sampled Points: 2218181

HeatThreadPerBlock: 256, HeatBlockPerGrid: 16
CUDA Running Time of Calculating Heat Map in Program_HeatMapAzi: 62.779968 (ms)

CUDA Overall Running Time for Program_HeatMapAzi with Resolution 0.050000 cm is: 875 (ms)
```

Figure 24. Running times for heat map under oblique azimuthal projection (CUDA).

Nevertheless for heat map generalization, although these two methods still provide exactly the same results in every steps, CUDA version is much faster than Python.

Visualization of heat maps produced by Python and CUDA:

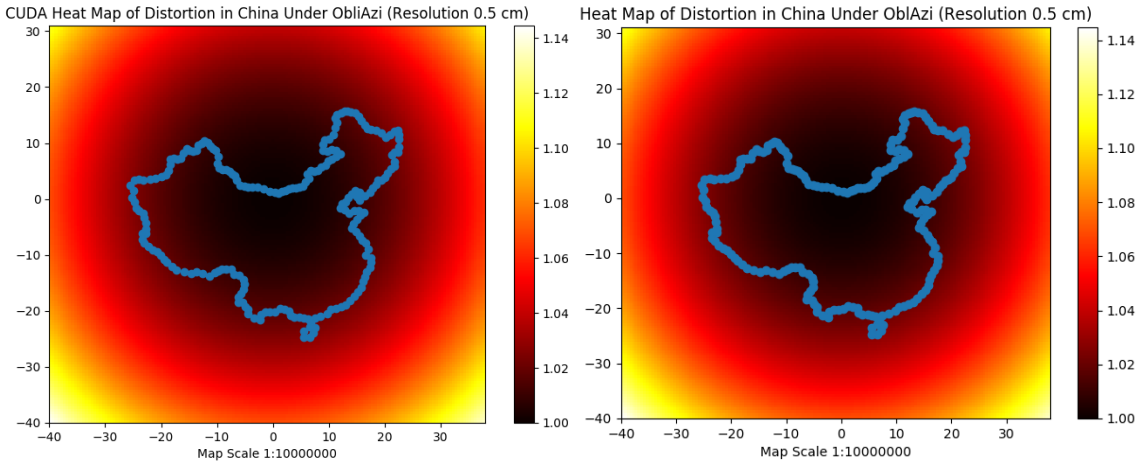


Figure 25. Heat maps under oblique azimuthal projection.

The same visualization of two heat maps directly and convincingly supports the correctness of CUDA version code.

4.1.2 Oblique Equidistant Cylindrical Projection

Programs for finding the best fitting directional vector:

1. Python version running results.

```
Python Version Dynamic Code for Finding the Best Fitting Oblique Equidistant Cylindrical Proj:

1th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 521.0 (ms)
Best Direcional Vector's Long/Lat: [-45.0, 50.0]
Best Direcional Vector Is: [2895.7435922485906, -2895.74359224859, 4880.469147111009, -0.7853981633974483, 0.8726646259971648]
Maximum Distortion: 5.20938424322%

2th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 69.0 (ms)
Best Directional Vector's Long/Lat: [-50.0, 52.0]
Best Directional Vector: [2521.2567883693573, -3004.716835391721, 5020.416511228426, -0.8726646259971648, 0.9075712110370514]
Maximum Distortion: 4.83477884471%

3th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 100.0 (ms)
Best Directional Vector's Long/Lat: [-50.8, 52.2]
Best Directional Vector: [2467.967512085577, -3026.02713786809, 5034.077583845524, -0.8866272600131193, 0.9110618695410401]
Maximum Distortion: 4.82150942478%

4th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 101.0 (ms)
Best Directional Vector's Long/Lat: [-51.0, 52.28]
Best Directional Vector: [2452.9638474842423, -3029.15808034157, 5039.524847367723, -0.890117918517108, 0.9124581329426354]
Maximum Distortion: 4.80118736224%

5th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 80.0 (ms)
Best Directional Vector's Long/Lat: [-51.032, 52.288]
Best Directional Vector: [2450.8291240088593, -3029.980485441818, 5040.069033530367, -0.8906764238777461, 0.9125977592827951]
Maximum Distortion: 4.800511949128

Final Directional Vector's Long/Lat: [-51.032, 52.288]
Final Directional Vector Is: [2450.8291240088593, -3029.980485441818, 5040.069033530367, -0.8906764238777461, 0.9125977592827951]
Fianl Distortion Range: [ 0%, 4.800511949128 ]
Projected Band Range: 3812.036781 (KM)
Projected Band Mid: -0.115914077077 (KM)
Maximum Angle: -17.408910279 (Degrees)
Maximum Tangent Points: ID:0 [109.8410034, 18.3680992]

Overall running time for Dyn_CylProgram is: 1117.0 (ms)
```

Figure 26. Running results for finding directional vector under oblique cylindrical projection (Python).

2. CUDA version running results.

```
CUDA Version Dynamic Code for Finding the Best Fitting Oblique Equidistant Cylindrical Proj...

1th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 26.452703 (ms)
CUDA Best Direcional Vector's Long/Lat: [135.000000, -50.000000]
CUDA Best Direcional Vector Is: [-2895.743592, 2895.743592, -4880.469147, 2.356194, -0.872665]
CUDA Maximum Distortion: 5.209387 %

2th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 21.134527 (ms)
CUDA Best Directional Vector's Long/Lat: [130.000000, -52.000000]
CUDA Best Directional Vector Is: [-2521.256788, 3004.716835, -5020.416511, 2.268928, -0.907571]
CUDA Maximum Distortion: 4.834779 %

3th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 21.044960 (ms)
CUDA Best Directional Vector's Long/Lat: [129.200000, -52.200000]
CUDA Best Directional Vector Is: [-2467.967512, 3026.027138, -5034.077584, 2.254965, -0.911062]
CUDA Maximum Distortion: 4.821509 %

4th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 21.183519 (ms)
CUDA Best Directional Vector's Long/Lat: [129.000000, -52.280000]
CUDA Best Directional Vector Is: [-2452.963847, 3029.158080, -5039.524847, 2.251475, -0.912458]
CUDA Maximum Distortion: 4.801191 %

5th Iteration for Finding Direcional Vector...
CUDA Running Time for Calculate Distortion: 21.051552 (ms)
CUDA Best Directional Vector's Long/Lat: [128.968000, -52.288000]
CUDA Best Directional Vector Is: [-2450.829124, 3029.980485, -5040.069034, 2.250916, -0.912598]
CUDA Maximum Distortion: 4.800510 %

CUDA Final Directional Vector's Long/Lat: [128.968000, -52.288000]
CUDA Final Directional Vector Is: [-2450.829124, 3029.980485, -5040.069034, 2.250916, -0.912598]
CUDA Final Distortion Range: [ 0, 4.800510% ]
CUDA Projected Band Range: 3812.036781 (KM)
CUDA Projected Band Mid: 0.115914 (KM)
The Opposite Direction of F_V Is: [ -51.032000, 52.288000]
CUDA Overall Running Time for Finding Best Directional Vector: 297 (ms)
```

Figure 27. Running results for finding directional vector under oblique cylindrical projection (CUDA).

Programs for generating heat map:

1. Python version running results.

```
Heat Map of Distortion Under Oblique Equidistance Cylindrical Proj:

Running time of Projecting Boundary Points in Program_HeatMapCyl: 47.0 (ms)

Running time of Projecting Meridian/Parallels Points in Program_HeatMapCyl: 17.0 (ms)

Size of HeatMap Sampled Points: 2178961

Running time of Calculating Heat Map in Program_HeatMapCyl: 1701.0 (ms)

Overall running time for Program_HeatMapCyl with Resolution 0.05 cm is: 3240.0 (ms)
```

Figure 28. Running times for heat map under oblique cylindrical projection (Python).

2. CUDA version running results.

```
CUDA Version Heat Map of Distortion Under Oblique Equidistance Cylindrical Proj...

threadPerBlock: 256, blockPerGrid: 1
CUDA Running Time of Projecting Boundary Points in Program_HeatMapCyl: 1.287072 (ms)

MeriThreadPerBlock: 256, MeriBlockPerGrid: 1
CUDA Running Time of Projecting Meridian/Parallels Points in Program_HeatMapCyl: 0.529664 (ms)

Size of HeatMap Sampled Points: 2178961

HeatThreadPerBlock: 256, HeatBlockPerGrid: 16
CUDA Running Time of Calculating Heat Map in Program_HeatMapCyl: 38.537086 (ms)

CUDA Overall Running Time for Program_HeatMapCyl with Resolution 0.050000 cm is: 828 (ms)
```

Figure 29. Running times for heat map under oblique cylindrical projection (CUDA).

After comparison, all intermediate and final results of finding directional vector and generating heat map are the same, which proves the correctness of CUDA version program.

Visualization of heat maps produced by Python and CUDA:

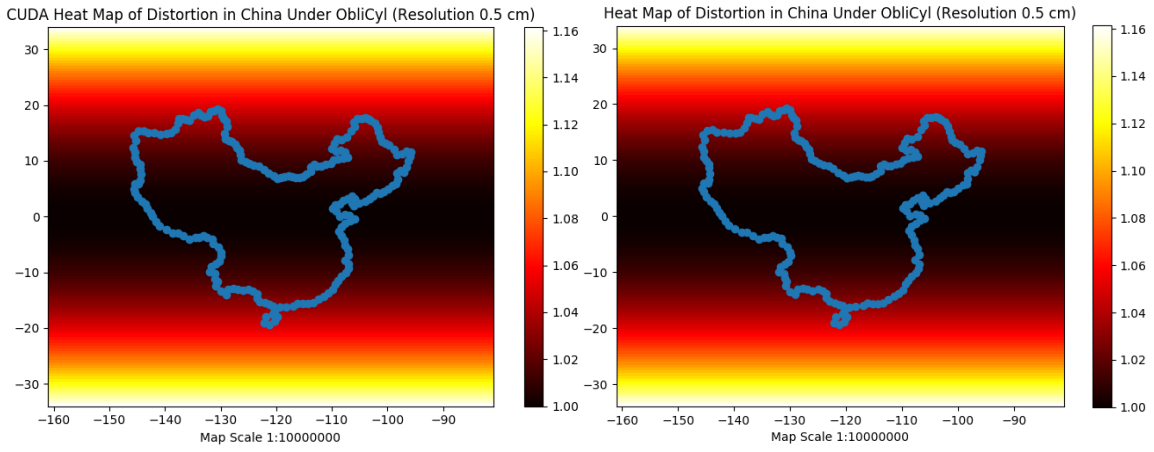


Figure 30. Heat maps under oblique cylindrical projection.

The same visualization of two heat maps directly and convincingly supports the correctness of CUDA version code.

4.1.3 Oblique Equidistant Conic Projection

Programs for finding the best fitting directional vector:

1. Python version running results.

```
Python Version Dynamic Code for Finding the Best Fitting Oblique Equidistant Conic Proj:

1th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 1733.0 (ms)
Best Direcional Vector's Long/Lat: [95.0, 65.0]
Best Direcional Vector's Phi0: 60.3744834797 [42.3224875091, 73.446618493]
Best Direcional Vector Is: [-234.66691975818202, 2682.255166563918, 5774.0869112104965, 1.6580627893946132, 1.1344640137963142]
Maiximum Distortion: 3.90087427953%

2th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 216.0 (ms)
Best Directional Vector's Long/Lat: [98.0, 63.0]
Best Directional Vector Is: [-402.54058548904015, 2864.2250940195304, 5676.602565604091, 1.710422666954443, 1.0995574287564276]
Maximum Distortion: 3.67413316497%

3th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 218.0 (ms)
Best Directional Vector's Long/Lat: [98.6, 62.6]
Best Directional Vector Is: [-438.4275825171549, 2898.9672933344664, 5656.271818704013, 1.7208946424664087, 1.0925761117484503]
Maximum Distortion: 3.61772245906%

4th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 217.0 (ms)
Best Directional Vector's Long/Lat: [98.76, 62.52]
Best Directional Vector Is: [-447.7236496305211, 2905.534362473137, 5652.172555933634, 1.7236871692695996, 1.0911798483468547]
Maximum Distortion: 3.60551639595%

5th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 215.0 (ms)
Best Directional Vector's Long/Lat: [98.768, 62.496]
Best Directional Vector Is: [-448.49019394575237, 2907.811473396557, 5650.940627994409, 1.723826795609759, 1.090760969326376]
Maximum Distortion: 3.60410234992%

6th Iteration for Finding Direcional Vector...
Running time for Calculate Distortion: 217.0 (ms)
Best Directional Vector's Long/Lat: [98.7712, 62.4896]
Best Directional Vector Is: [-448.74884684254, 2908.4102364543005, 5650.6119464130015, 1.723882646145823, 1.0906492682542483]
Maximum Distortion: 3.60384996417%

Final Directional Vector's Long/Lat: [98.7712, 62.4896]
Final Directional Vector's Phi0(Oblique): 62.6703315024 [45.2112169367, 75.1069610561]
Final Directional Vector Is: [-448.74884684254, 2908.4102364543005, 5650.6119464130015, 1.723882646145823, 1.0906492682542483]
Final Distortion Range: [ 0%, 3.60384996417% ]

Tangent Points: ID:0 [109.8410034, 18.3680992] ID:204 [121.1080017, 53.2863007]


Overall running time for Dyn_ConProgram is: 3049.0 (ms)
```

Figure 31. Running results for finding directional vector under oblique conic projection (Python).

2. CUDA version running results.

```
CUDA Version Dynamic Code for Finding the Best Fitting Oblique Equidistant Conic Proj...

1th Iteration for Finding Direcional Vector...
threadPerBlock: 256, blockPerGrid: 11
CUDA Running Time for Calculate Distortion: 82.063744 (ms)
CUDA Best Direcional Vector's Long/Lat: [95.000000, 65.000000]
CUDA Best Direcional Vector's Phi0(Under Oblique): 60.374483, [42.322488, 73.446618]
CUDA Best Direcional Vector Is: [-234.666920, 2682.255167, 5774.086911, 1.658063, 1.134464]
CUDA Maximum Distortion: 3.900874 %

2th Iteration for Finding Direcional Vector...
threadPerBlock: 128, blockPerGrid: 1
CUDA Running Time for Calculate Distortion: 38.693890 (ms)
CUDA Best Directional Vector's Long/Lat: [98.000000, 63.000000]
CUDA Best Directional Vector's Phi0(Under Oblique): 62.258171, [44.625238, 74.818617]
CUDA Best Directional Vector Is: [-402.540585, 2864.225094, 5676.602566, 1.710423, 1.099557]
CUDA Maximum Distortion: 3.674133 %

3th Iteration for Finding Direcional Vector...
threadPerBlock: 128, blockPerGrid: 1
CUDA Running Time for Calculate Distortion: 38.487873 (ms)
CUDA Best Directional Vector's Long/Lat: [98.600000, 62.600000]
CUDA Best Directional Vector's Phi0(Under Oblique): 62.582926, [45.084075, 75.047861]
CUDA Best Directional Vector Is: [-438.427583, 2898.967293, 5656.271819, 1.720895, 1.092576]
CUDA Maximum Distortion: 3.617722 %

4th Iteration for Finding Direcional Vector...
threadPerBlock: 128, blockPerGrid: 1
CUDA Running Time for Calculate Distortion: 39.922047 (ms)
CUDA Best Directional Vector's Long/Lat: [98.760000, 62.520000]
CUDA Best Directional Vector's Phi0(Under Oblique): 62.646580, [45.180514, 75.088161]
CUDA Best Directional Vector Is: [-447.723650, 2905.534362, 5652.172556, 1.723687, 1.091180]
CUDA Maximum Distortion: 3.605516 %

5th Iteration for Finding Direcional Vector...
threadPerBlock: 128, blockPerGrid: 1
CUDA Running Time for Calculate Distortion: 38.403873 (ms)
CUDA Best Directional Vector's Long/Lat: [98.768000, 62.496000]
CUDA Best Directional Vector's Phi0(Under Oblique): 62.665090, [45.204652, 75.102662]
CUDA Best Directional Vector Is: [-448.490194, 2907.811473, 5650.940628, 1.723827, 1.090761]
CUDA Maximum Distortion: 3.604102 %

6th Iteration for Finding Direcional Vector...
threadPerBlock: 128, blockPerGrid: 1
CUDA Running Time for Calculate Distortion: 39.628159 (ms)
CUDA Best Directional Vector's Long/Lat: [98.771200, 62.489600]
CUDA Best Directional Vector's Phi0(Under Oblique): 62.670332, [45.211217, 75.106961]
CUDA Best Directional Vector Is: [-448.748847, 2908.410236, 5650.611946, 1.723883, 1.090649]
CUDA Maximum Distortion: 3.603850 %

CUDA Final Directional Vector's Long/Lat: [98.771200, 62.489600]
CUDA Final Directional Vector's Phi0(Under Oblique): 62.670332, [45.211217, 75.106961]
CUDA Final Directional Vector Is: [-448.748847, 2908.410236, 5650.611946, 1.723883, 1.090649]
CUDA Final Distortion Range: [ 0 , 3.603850 % ]
CUDA Overall Running Time for Finding Best Directional Vector: 500 (ms)

Average Running Time with 256 ThreadsPerBlock and 16 BlocksPerGrid Is: 501.600000 (ms)
```

Figure 32. Running results for finding directional vector under oblique conic projection (CUDA).

Programs for generating heat map:

1. Python version running results.

```
Heat Map of Distortion Under Oblique Equidistance Conic Proj...

Running time of Projecting Boundary Points in Program_HeatMapCon: 16.0 (ms)

Running time of Projecting Meridian/Parallels Points in Program_HeatMapCon: 0.0 (ms)

Size of HeatMap Sampled Points: 992001

Running time of Calculating Heat Map in Program_HeatMapCon: 2104.0 (ms)

Overall running time for Program_HeatMapCon with Resolution 0.05 cm is: 2820.0 (ms)
```

Figure 33. Running times for heat map under oblique conic projection (Python).

2. CUDA version running results.

```
CUDA Version Heat Map of Distortion Under Oblique Equidistance Conic Proj...

threadPerBlock: 256, blockPerGrid: 1
CUDA Running Time of Projecting Boundary Points in Program_HeatMapCon: 1.409408 (ms)

MeriThreadPerBlock: 256, MeriBlockPerGrid: 1
CUDA Running Time of Projecting Meridian/Parallels Points in Program_HeatMapCon: 0.848576 (ms)

Size of HeatMap Sampled Points: 992001

HeatThreadPerBlock: 256, HeatBlockPerGrid: 16
CUDA Running Time of Calculating Heat Map in Program_HeatMapCon: 47.457951 (ms)

CUDA Overall Running Time for Program_HeatMapCon with Resolution 0.050000 cm is: 422 (ms)
```

Figure 34. Running times for heat map under oblique conic projection (CUDA).

Still after comparison of the results, all the intermediate and final results of finding directional vector and generating heat map are the same, proving the correctness of CUDA version program.

Visualization of heat maps produced by Python and CUDA:

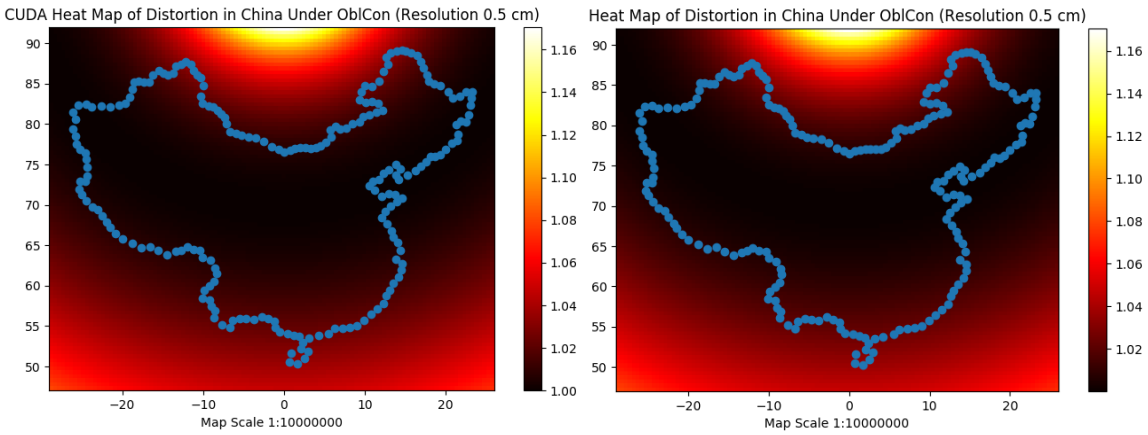


Figure 35. Heat maps under oblique conic projection.

The same visualization of two heat maps directly and convincingly supports the correctness of CUDA version code.

4.2 Efficiency

4.2.1 Oblique Equidistant Azimuthal Projection

For the numbers of blocks per grid, I tested 8, 16, 32, 64, 128, 256, 512; for the numbers of threads per block, I tested 4, 8, 16, 32, 64, 128, 256, 512; and for the resolutions of heat map, I tested 25, 15, 5, 1, 0.5, 0.1, 0.05 cm. All the following evaluations are based on those presetting values. And all the running times are averaged by iterating the programs 20 times after warming up.

Table 2. Running time for finding directional vector (azimuthal).

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
197	4	16	251.6
	8		211.0
	16		192.2
	32		181.2
	64		175.1
	128		173.4
	256		175.1
	512		173.5

It is unexpected that some of the running times of CUDA are even slightly slower than Python without any parallelism. This is actually caused by the time spent on memory copy and data format transformation when the quantity of input data set is comparatively not that high. Also, since we created too many idle threads for this program, I noticed that when we increase the numbers of threads over 64, the overall running time will not decrease anymore.

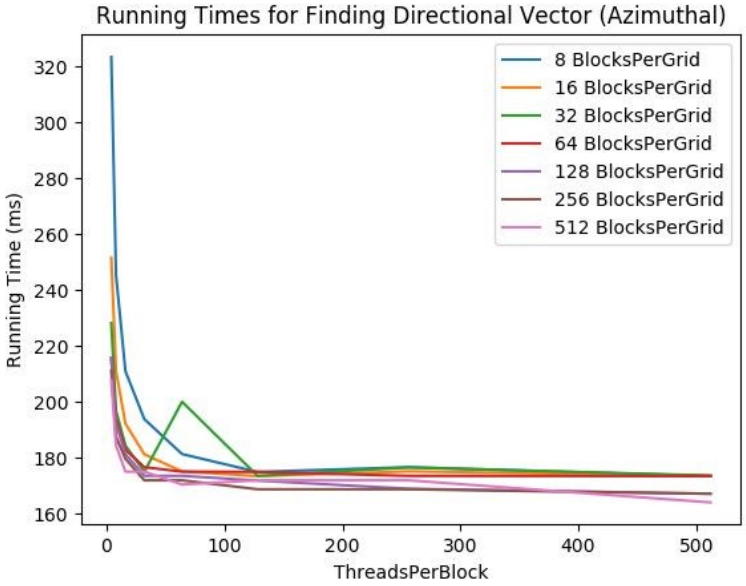


Figure 36. Running times for finding directional vector (azimuthal).

This figure shows all the possible combinations of numbers of threads and blocks, which indicates that the number of blocks will not affect the overall running time greatly when it is over 100. The sudden vibration of the broken line is caused by some unpredictable and random factors, which needed at least 128 threads for reliability.

Table 3. Running time for heat map with resolution 0.05 cm.

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
4127	4	16	1914.2
	8		1401.6
	16		1118.9
	32		1029.6
	64		981.2
	128		962.5
	256		959.7
	512		968.7
	4	256	976.6
	8		950.1
	16		949.9
	32		973.4
	64		954.7
	128		948.4
	256		942.1
	512		951.6

There are 2,218,181 sampled points in this heat map. Under this situation, CUDA will show us the power of parallelism. Even we only create 4 threads and 16 blocks, the overall running time already decreases from 4127ms to 1914ms, which is a huge improvement. After I increase the number of threads up to 256, the overall running time

keeps decreasing to 960ms, accelerating 76.7% compared to Python. Also, too many idle threads, such as 512 threads and 256 blocks, will not improve it any more.

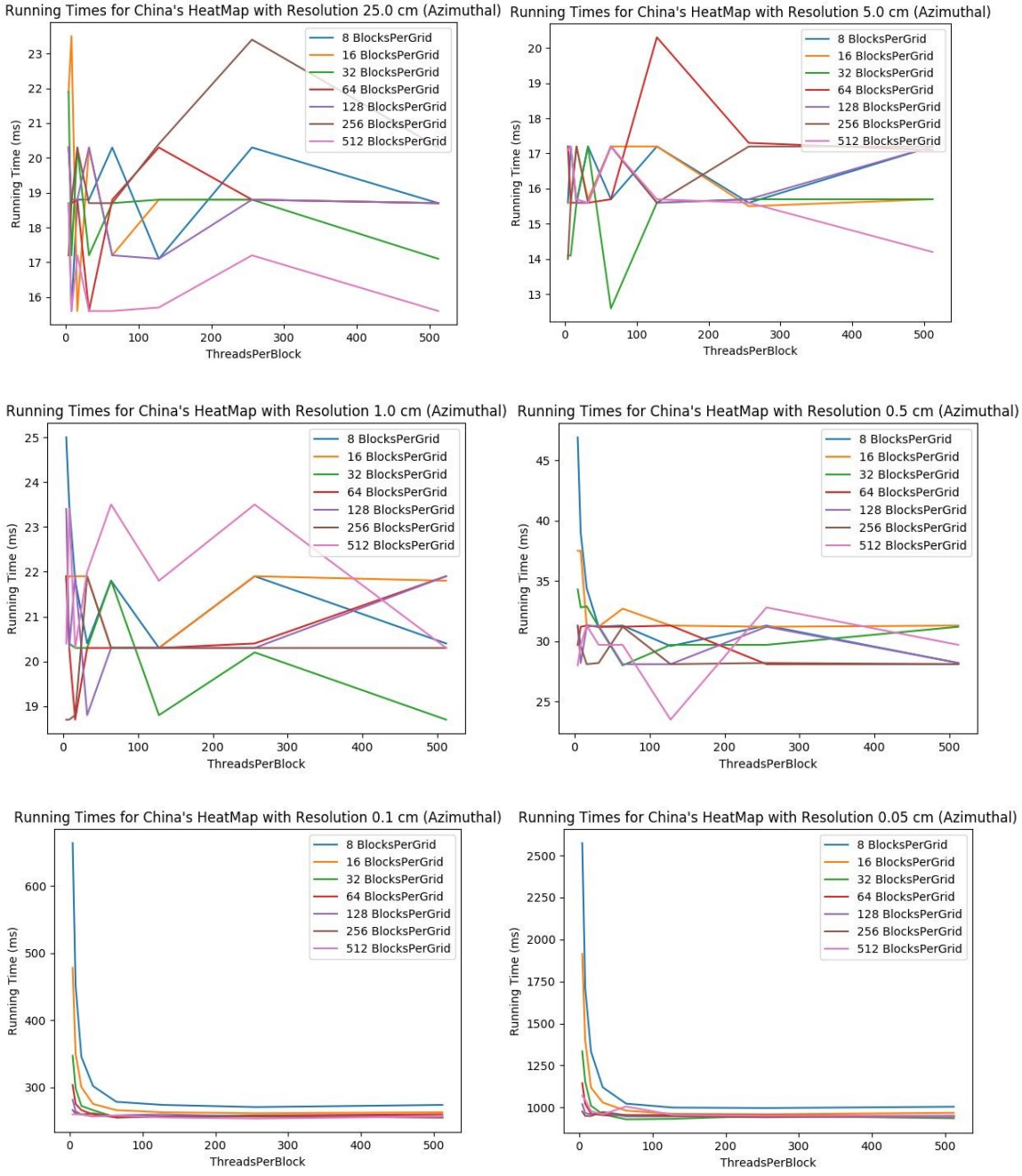


Figure 37. Running times of heat map under different resolutions.

When the resolution of heat map is 25 cm, it only has 16 sampled points. Under this circumstance, the unpredicted and random factors will dominate the running time instead of the settings of threads and blocks, thus the broken line graph is in a mess and vibrates a lot. However, when the density of sampled points is high enough, the trend of broken lines under different settings will become more uniform. Such as 100 threads are absolutely enough for those heat maps with 0.1 cm resolution or even lower.

4.2.2 Oblique Equidistant Cylindrical Projection

Table 4. Running time for finding directional vector (cylindrical).

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
922.4	4	16	1078.1
	8		687.5
	16		482.7
	32		392.3
	64		339.1
	128		318.7
	256		304.7
	512		328.1

When we create enough threads for it, CUDA will benefit a lot from parallelism: there will be at least 32 threads under 16 blocks. However, due to the low efficiency of dynamic cylindrical algorithm, programs take more time compared to previous azimuthal projection's algorithm.

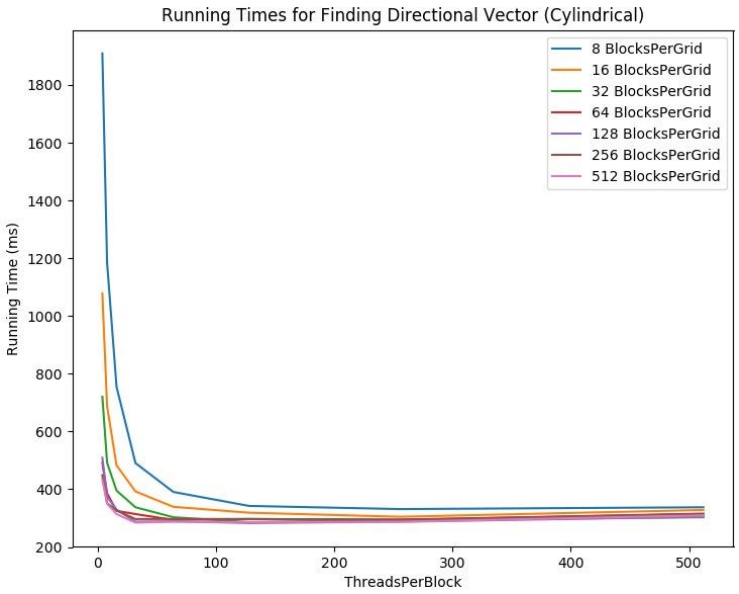


Figure 38. Running times for finding directional vector (cylindrical).

The number of blocks will seldom affect the overall running time when the number of threads is over 80. At least 128 threads are needed for reliability.

There are 2,178,961 sampled points in this heat map. Even we only create 4 threads and 16 blocks, the overall running still decreases from 3154ms to 1373ms, which is a huge performance improvement. When I increase the number of threads up to 64, the overall running time keeps decreasing to 969ms, accelerating 69.3% compared to Python. Too many idle threads and blocks will not improve the performance any further, even causing some slight vibrations. Compared with more blocks, we prefer using more threads in a single block, since all the threads in a block shares a same memory that guarantees a much faster access time. This conclusion is supported by the 967ms running time under 256 threads and 16 blocks, but 991ms under 16 threads and 256 blocks.

Table 5. Running time for heat map with resolution 0.05 cm.

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
3154.0	4	16	1373.4
	8		1112.5
	16		1032.8
	32		1048.2
	64		968.5
	128		984.3
	256		965.7
	512		979.6
	4	256	1000.1
	8		970.2
	16		990.6
	32		999.9
	64		963.9
	128		970.3
	256		1031.1
	512		1004.6

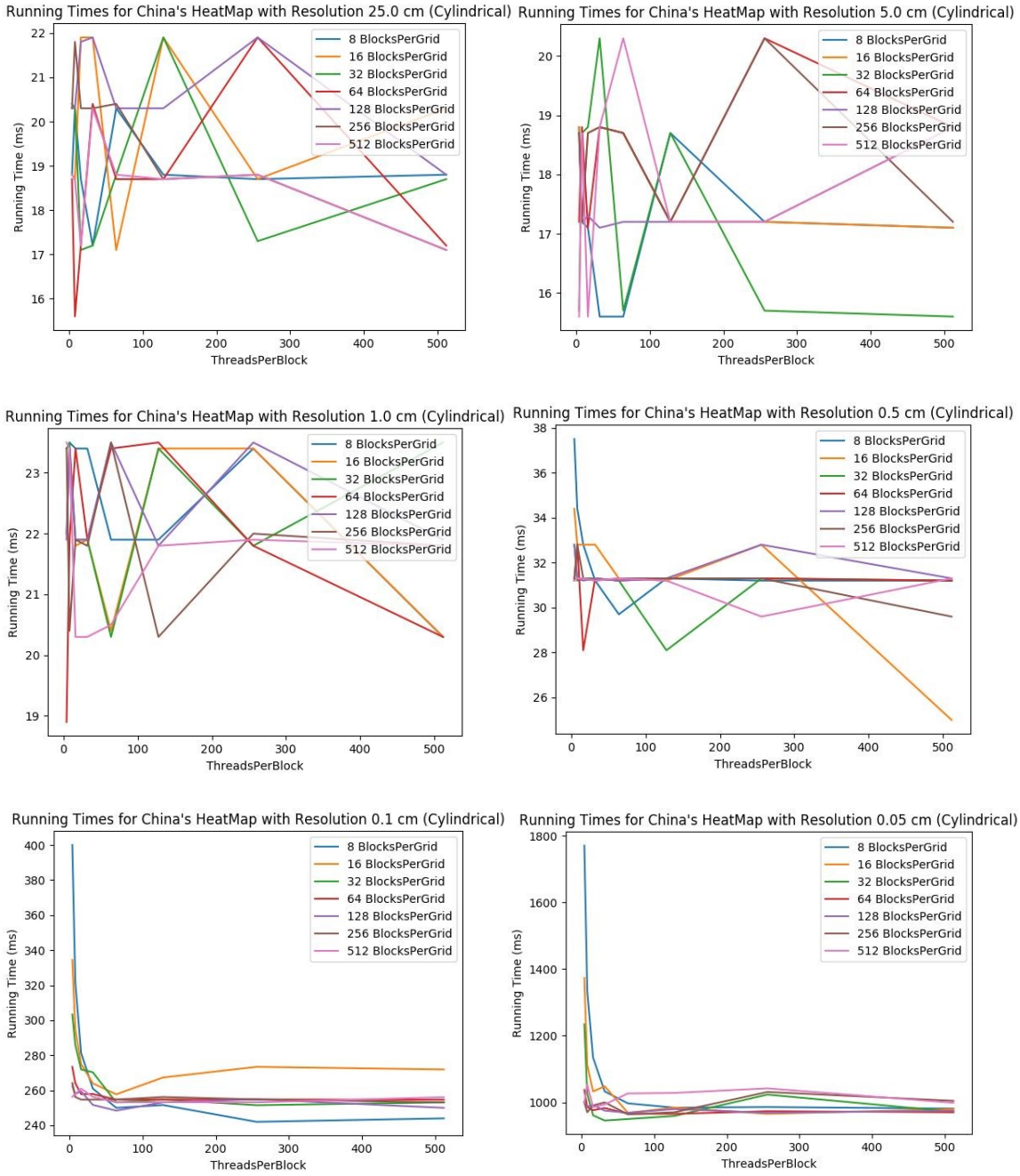


Figure 39. Running times of heat map under different resolutions.

According to the figures, we can pick reliable threads and blocks settings under each resolution: 128 threads and 16 blocks are enough for heat map with 0.05 cm resolution.

4.2.3 Oblique Equidistant Conic Projection

Table 6. Running time for finding directional vector (conic).

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
2926.2	4	16	1134.3
	8		993.8
	16		825.0
	32		686.0
	64		572.0
	128		518.7
	256		511.0
	512		504.7

Since the compute-intensive algorithm for oblique conic projection will iterate 1,000 times for each candidate directional vector. Even merely with 4 threads and 16 blocks, CUDA will accelerate the processing dramatically. After the thread number reaches 128, the running time will stabilize to 510ms approximately.

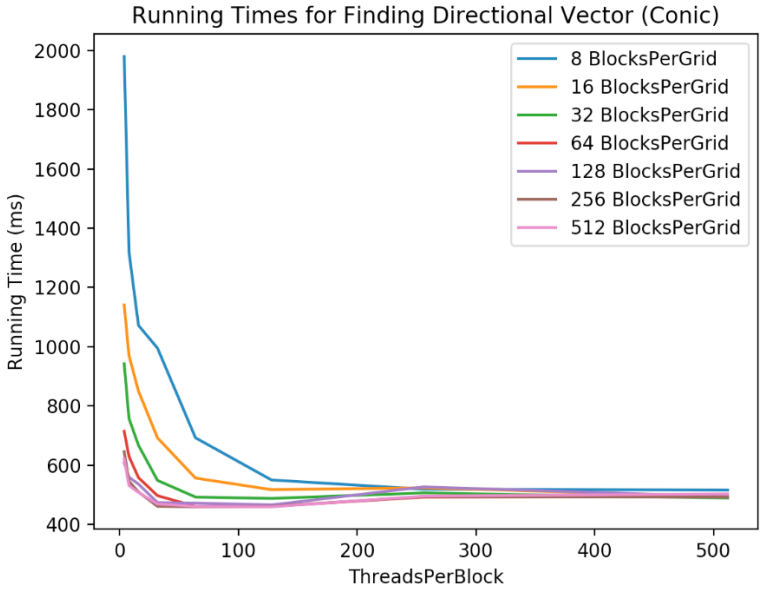


Figure 40. Running times for finding directional vector (conic).

The number of blocks will not affect the overall running time greatly when the number of threads is over 200. At least 256 threads are needed for reliability.

Table 7. Running time for heat map with resolution 0.05 cm.

Python	CUDA		
Running Time (ms)	Threads Num	Blocks Num	Running Time (ms)
2759.4	4	16	1331.3
	8		950.0
	16		638.9
	32		509.3
	64		457.7
	128		451.6
	256		451.6
	512		440.8
	4	256	475.1
	8		448.4
	16		455.0
	32		457.8
	64		443.7
	128		440.6
	256		446.9
	512		442.1

There are 992,001 sampled points in this heat map. Even we only create 4 threads and 16 blocks, the overall running already decreases to 1331ms from 2759ms, a huge performance improvement. After increasing the number of threads up to 128, the overall running time keeps decreasing to 452ms, accelerating 83.6% compared to Python. Too many idle threads and blocks will not improve the performance any further, even causing some slight vibrations. Considering the memory access time, I think 256 threads and 16 blocks are reasonable and acceptable setting.

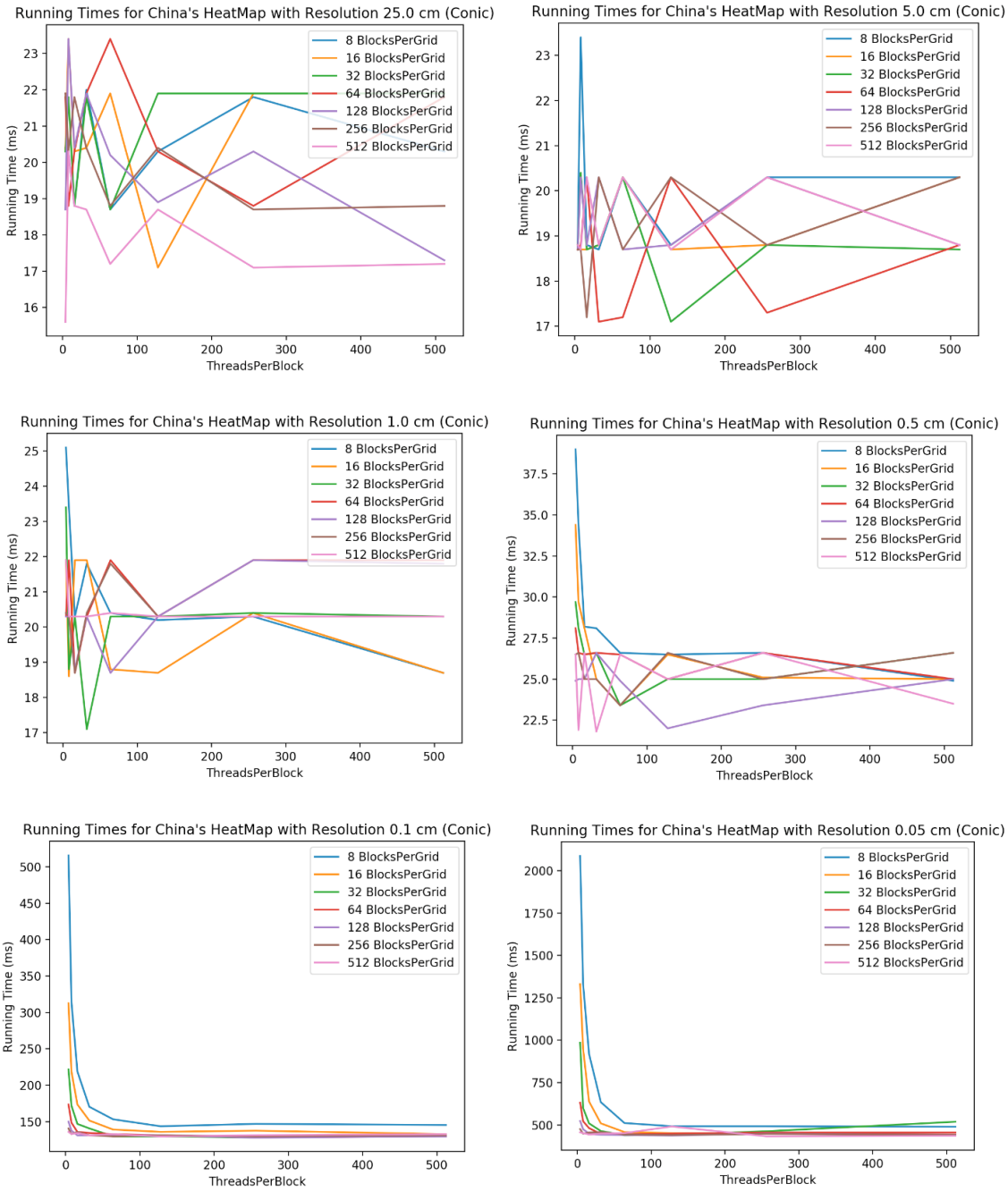


Figure 41. Running times of heat map under different resolutions.

According to the figures, we can pick reliable threads and blocks settings under each resolution, such as 256 threads and 16 blocks enough for heat map with 0.05 cm resolution.

Chapter 5. Map Projection Recognition

5.1 Background

We already figured out how to map a given region with minimal distortion under different projections, however, our knowledge about those maps without projection parameters is limited. Although most of them are titled with their location, few of the authors provide detailed projection information on the map. We merely know the location, but we have no idea about what kind of projections those maps used as well as their projection parameters. Inspired by the popularity of current machine learning and image recognition, I want to apply machine learning in this field to figure out this question.

I optimized the standard distance-versus-angle signature to extract and describe the contour of target region accurately, reducing the impact of distortion and nonuniform scale. In order to get reliable results, I trained and compared the performance of several classification models, multilayer perceptron (MLP), nearest neighbors and radial basis function (RBF) kernel support vector machine (SVM). Based on the average performance of trained machine learning models, we finally pick MLP, which predicts the projection method of an unknown map with an acceptable accuracy.

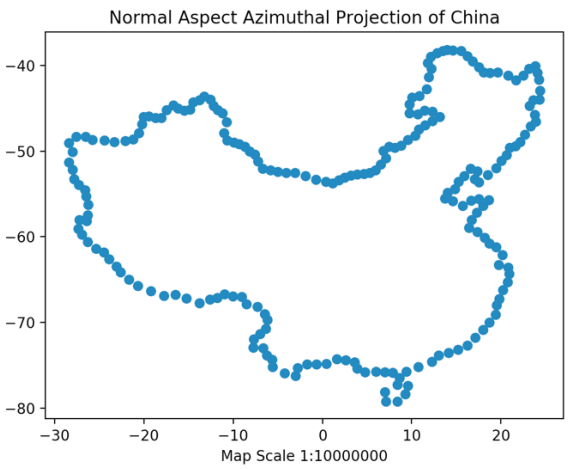
Machine learning will make a new breakthrough in traditional projection research and help us to better understand the development of map projection methods and also help computers classify unknown maps more precisely. For example, when searching

azimuthal projected maps on a web browser, at least it will not give us any unrelated cylindrical or conic projected maps.

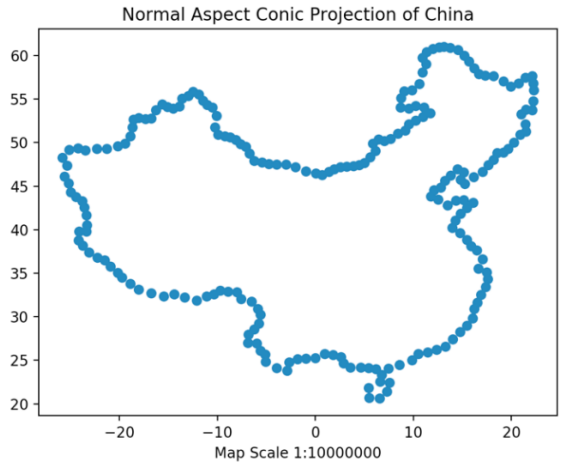
5.2 Extract and Represent Boundary

It is the projection method that determine the contour and location of projected region on the 2D map. Different projection will produce different maps, even though they might look very similar occasionally. Normally for the same region, azimuthal projection tends to generate a circular region; cylindrical projection tends to generate an east-west extended region; while conic projection tends to generate a fan-shaped region. Figure 42 are maps projected by normal aspect azimuthal, cylindrical, and conic projection. We can easily find the distinction between (a) and (c): the projected region's width is narrower on (c), and the curvature of boundaries on the east coast is also different. Normal aspect conic projection on (b) is an intermediate stage.

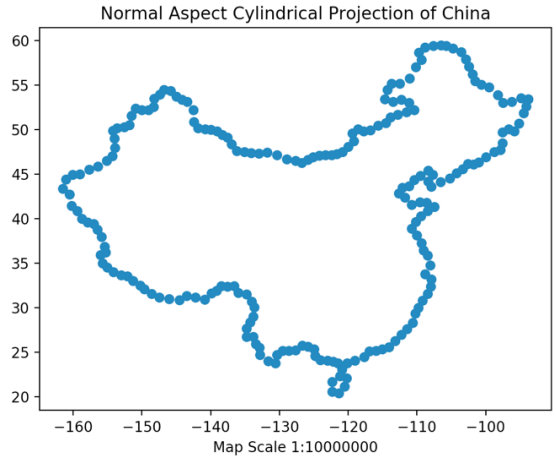
However, although projected by oblique aspect azimuthal, cylindrical, and conic projection, they look similar in Figure 43. And I believe differentiate them is the most difficult part in this task.



(a)

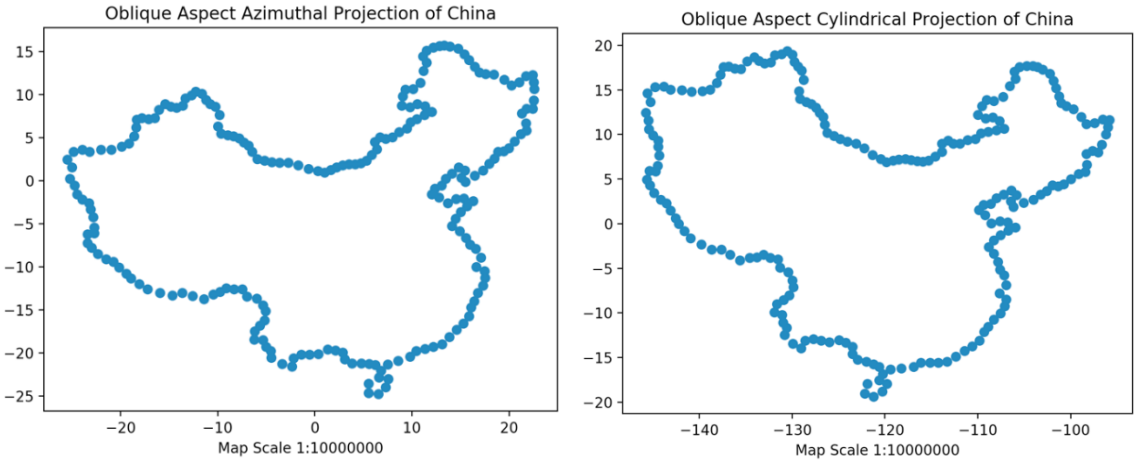


(b)

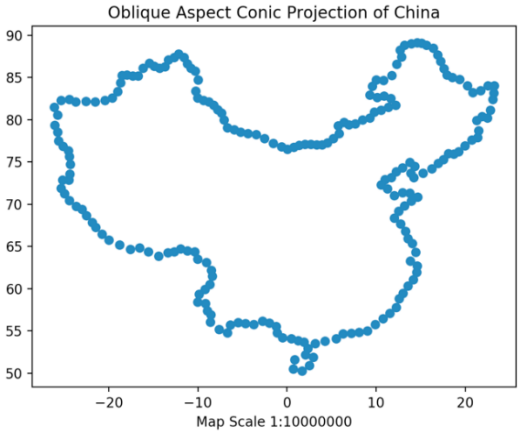


(c)

Figure 42. Normal aspect projections of China.



(a) (b)



(c)

Figure 43. Oblique aspect projections of China.

Since my research object is the contour of projected region, extracting and describing it is the first step. Given the situation that similar regions should have the same description regardless of their position or orientation on the map, good projections should be invariant to translation, rotation, and scale.

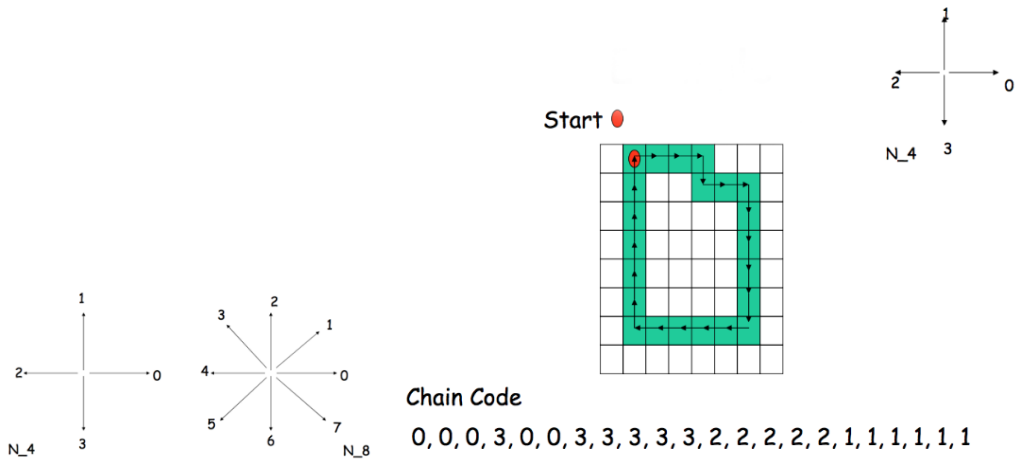


Figure 44. Chain codes.

Chain code is popular in the field of cartography and widely used to represent the boundary characteristics of a region. It uses a logically connected sequence of numbers, which include length and direction information. As Figure 44 shows, there are two options for the direction setting: eight-direction definitely will preserve the shape better, but the length of chain code is much longer than 4-direction.

But chain code is sensitive to local noise, even a small variation on the boundary might cause big problems. Besides, chain code will be very long when depicting a long and complex boundary. It is not robust to map scale too. Even the same line segment under different map scale will produce totally different chain codes.

A signature is a 1-D function used to describe a 2-D region. Pervasively, we used the distance from the centroid of the target region to the boundary as a function of angle, which is called distance-versus-angle signature. From Figure 45, we can see that different contours often has distinct signatures.

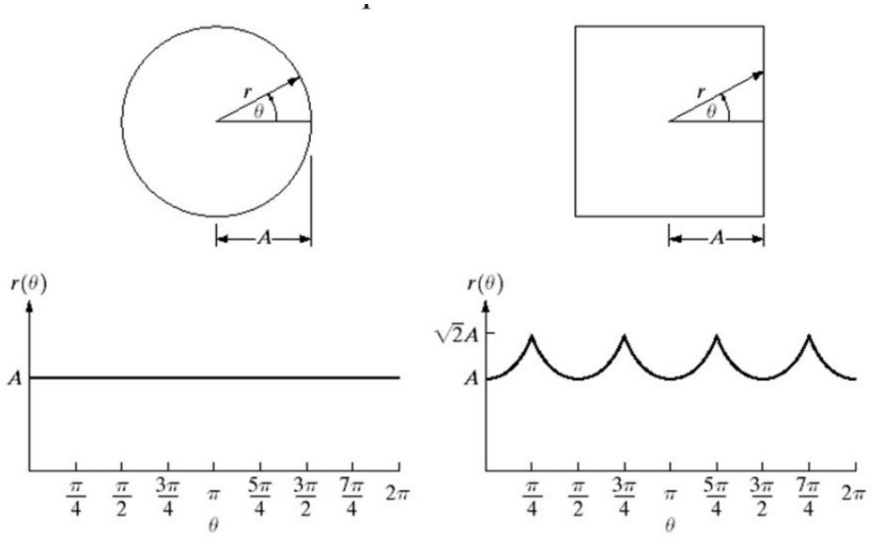


Figure 45. Distance-versus-Angle signatures.

However, caused by different resolution of maps, distance-versus-angle signature is not invariant to scale. In order to optimize, I normalized the distance to a uniform range. Moreover, it is not invariant to rotation too, which means the start point matters a lot in the final results. I decided to always start from the farthest point and picked next one clockwise, which makes sure that all signatures start from the same point. Compared to chain codes, since noise will only affect single distance value instead of the whole graph, distance-versus-angle signature is more robust to local noise.

Given its simplicity and acceptable accuracy, I finally choose to use optimized distance-versus-angle signature as my description method to represent contours of target regions. After training the machine learning models, we hope they can learn and predict the projection of an unknown map correctly, which is the object of this research.

5.3 Optimized Distance-versus-Angle Signature

Applying three different projections to the geographic coordinates of feature points on the target region's boundary, we can get the needed Cartesian coordinates on the map. The

projection axis is defined according to the definition of normal aspect and oblique aspect projections. Following the standard procedures of distance-versus-angle signature, we get three distinct signatures of normal aspect azimuthal, cylindrical, and conic projections of China as Figure 46, as well as three similar signatures of oblique aspect azimuthal, cylindrical, and conic projections as Figure 47.

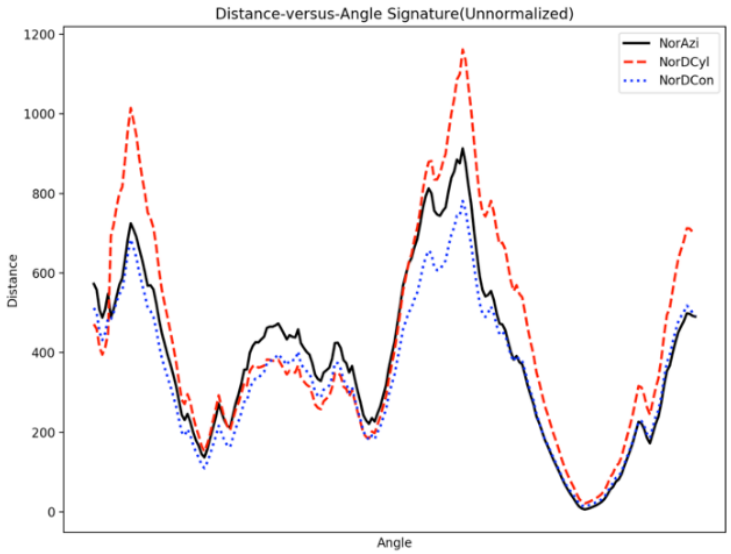


Figure 46. Unnormalized Distance-versus-Angle signatures of normal aspect projections.

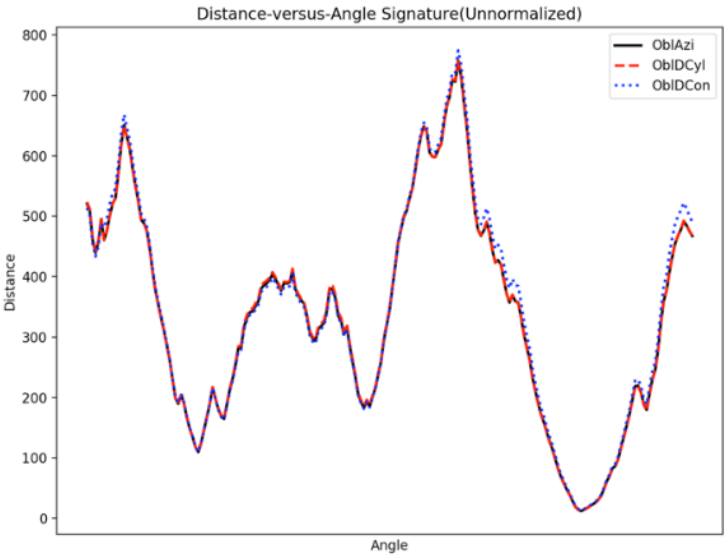


Figure 47. Unnormalized Distance-versus-Angle signatures of oblique aspect projections.

According to Figure 46, it is apparent that there is a huge distinction between normal aspect cylindrical projection and its counterparts. The difference between the rest two projections is significant in the middle, but negligible near the tail.

However, in Figure 47, except for slightly variation of oblique aspect conic projection, we find that those three oblique aspect projects produce almost the same signatures. Due to the high contact ratio of them, we have to pay much attention to oblique projections in this research.

For optimization, normalization is a simple but effective method to get rid of the bad effects of nonuniform scale. The distance ranges of those two graphs in Figure 46 and 47 demonstrate that their scale is different. To amplify the variance of different signatures, instead of traditionally from 0 to 1, we normalized it to the range from 0 to 1000. Figure 48 and 49 show the normalized signatures, which give us totally different signatures for normal aspect projections; while for oblique aspect projections, they are not affected greatly due to comparatively low distortion everywhere on the map.

To solve the problem caused by rotation, optimized distance-versus-angle signature will always start from the farthest point from the centroid of target region and find next feature point clockwise along the boundary. This guarantees that all the signatures start with the same point.

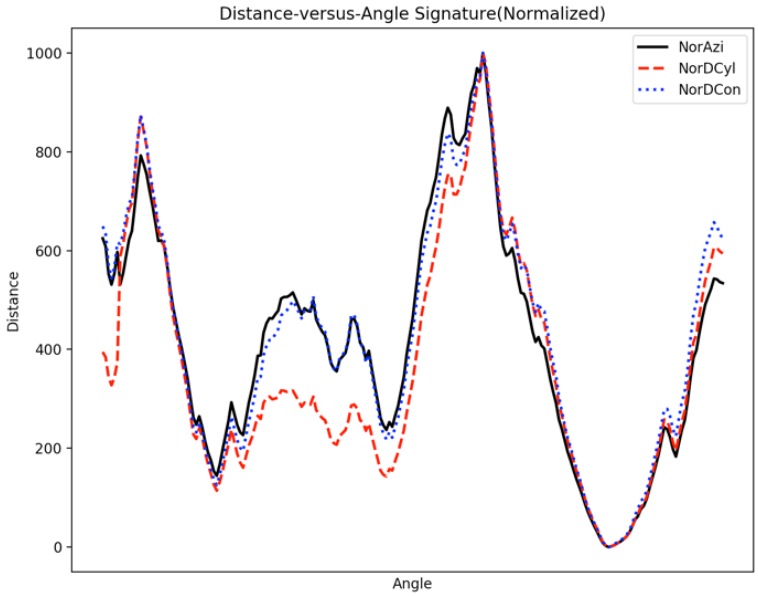


Figure 48. Normalized Distance-versus-Angle signatures of normal aspect projections.

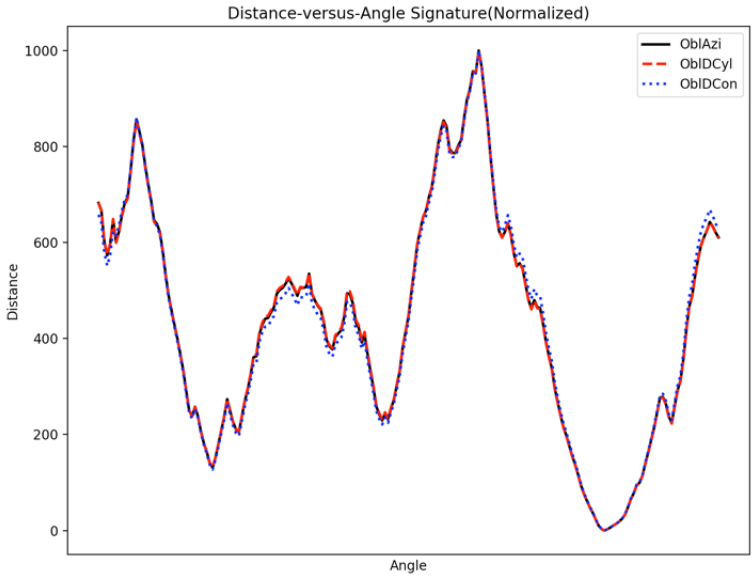


Figure 49. Normalized Distance-versus-Angle signatures of oblique aspect projections.

Still take China as our research area. In order to generate normal aspect projected training and testing data, we resampled directional vector around the traditional normal direction

and conducted projection based on this new directional vector. Following the same operations to get oblique aspect projected training and testing data, we get 3,000 training maps and 300 testing maps under each projection—overall 18,000 training maps and 1,800 testing maps.

5.4 MLP Prediction Results and Discussions

Multilayer Perceptron (MLP) is a class of feedforward artificial neural network, consisting of at least three layers of nodes. Each node is a neuron with nonlinear activation function, which helps MLP processes data that is not linearly separable, except for the input nodes. MLP uses backpropagation for training.

Scikit-learn is an open source, simple, and efficient machine learning package in Python, providing lots of functions for data mining and data analysis. For classification, it includes abundant classifiers, such as MLP, Support Vector Machine, and Decision Tree. So, all the machine learning models used in our research is from scikit-learn.

During the following experiment, we set the model to shuffle the training samples in each iteration to avoid the model learning the separation and sequence of data, which might be misleading. Also prevent over-fitting through early stop. After comparing the performance of logistic activation function, which can only reach 35% correctness, we finally choose rectified linear unit function as the activation function. Besides, we choose adam, a stochastic gradient-based optimizer, as the solver, because it works well on relatively large datasets than quasi-Newton methods.

Since the prediction performance varies a lot even under the same settings, we take ten iterations and compare the average performance.

According to Figure 50, a single hidden layer MLP's prediction correctness is limited and can only achieve up to 60% when it has 95 nodes. When the number of nodes is higher

than 50, it will not improve the prediction correctness any more. Also, more nodes mean slower learning speed, so 50 nodes are enough for a single hidden layer MLP.

Table 8. Prediction accuracy of single hidden layer MLP.

Number of Nodes (1^{st} Hidden Layer)	Accuracy (%)	Number of Nodes (1^{st} Hidden Layer)	Accuracy (%)
5	16.67	105	54.82
15	44.21	115	56.33
25	52.88	125	56.96
35	57.23	135	57.57
45	56.64	145	55.09
55	59.08	155	53.44
65	54.94	165	53.48
75	55.38	175	55.03
85	53.81	185	54.72
95	60.00	195	54.81

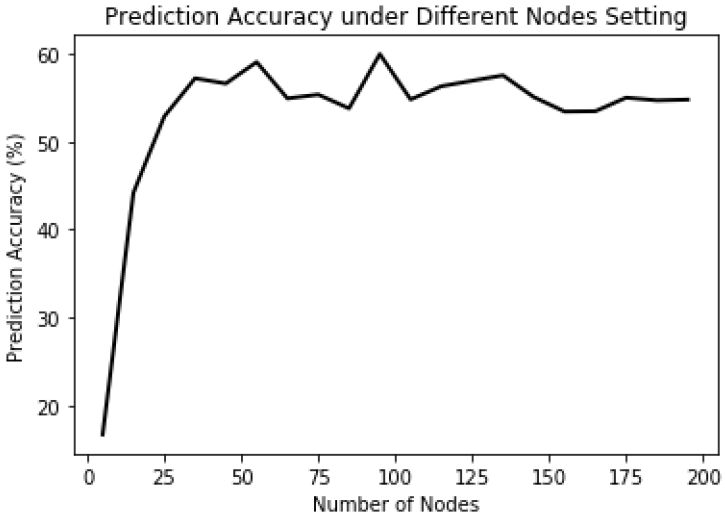


Figure 50. Prediction accuracy of single hidden layer MLP.

Table 9. Prediction accuracy of two hidden layer MLP.

Number of Nodes (1st Hidden Layer)	Number of Nodes (2nd Hidden Layer)	Correctness	Number of Nodes (1st Hidden Layer)	Number of Nodes (2nd Hidden Layer)	Correctness
40	105	56.22	60	105	67.42
	115	78.07		115	62.89
	125	88.12		125	67.38
	135	75.46		135	62.03
	145	75.6		145	66.30
50	105	65.06	70	105	70.87
	115	60.97		115	72.18
	125	74.08		125	68.32
	135	65.91		135	71.27
	145	62.6		145	76.13

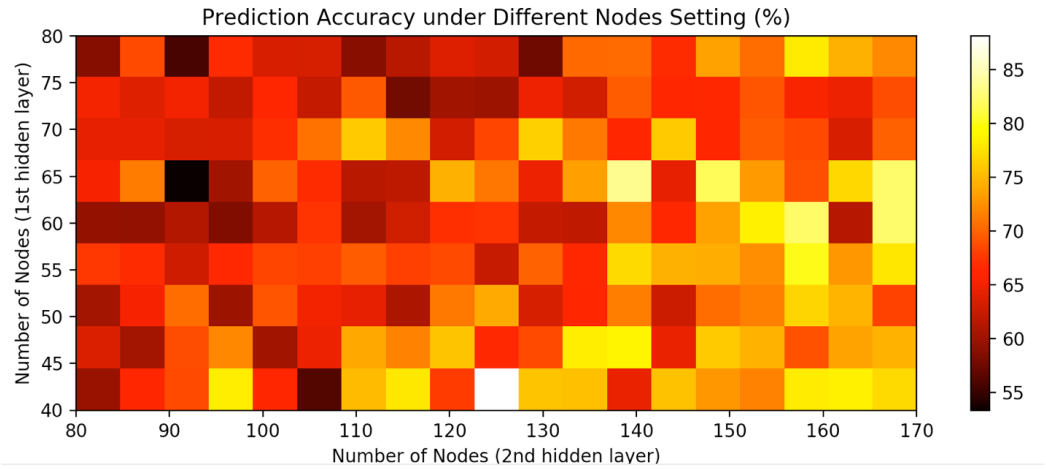


Figure 51. Prediction accuracy of two hidden layer MLP.

Two hidden layers MLP's overall performance is much better than single hidden layer MLP. According to Table 9, since more nodes did not improve the prediction results correspondingly, there is no strong positive correlation between the number of nodes in each layer and prediction accuracy. Figure 51 supports this conclusion too. Figure 51 shows the minimum prediction correctness is 53% and maximum is 88%. Although the performance fluctuates greatly even under similar settings, prediction accuracy at the right-bottom corner is normally much higher than the left-top corner, which means less nodes in the first layer and more nodes in the second layer will improve the final prediction accuracy. By setting 40 nodes on the first hidden layer and 125 nodes on the second hidden layer, we can get 88.12% prediction accuracy.

I also try three hidden layer MLP, but the overall prediction accuracy is still around 88%, and it takes much longer time to train the model.

As two hidden layers MLP is a balance between prediction accuracy and time efficiency, we finally pick it to train the model.

Chapter 6. Conclusion

6.1 Discussion on results

Since the early days of human history, people have been making all kinds of maps for various demands as well as researching on projections to minimize distortion to produce accurate maps. Limited by computational capabilities of traditional computers, cartographers always give priority to normal aspect projections since the directional vector of it is the Earth's rotation axis. Although subsequent cartographers developed revised normal aspect tangent and secant projections, and even pseudo projections, normal aspect projections are still prevalent for many years considering its simplicity, time efficiency, and acceptable accuracy.

However, our revolutionary algorithms for finding the best fitting oblique aspect azimuthal, cylindrical, and conic projections solve this age-old problem well, improving the distortion and time efficiency dramatically at the same time. Moreover, benefitting from CUDA, we are able to go much further and reach a higher place in this field. For oblique conic projection with 25 million sampled points, CUDA accelerates the running time up to 98%, from 62 seconds to merely 655 milliseconds. Besides, the results of CUDA are exactly same as sequential versions. Furthermore, machine learning gives us an opportunity to understand previous map projection's history and development. We are able to classify, learn, and finally predict the projections and even parameters of unknown maps with an accuracy up to 88% directly without checking any reference historical materials.

6.2 Discussion on future directions

CUDA provides us the opportunity to realize real-time calculation and visualization, which is crucial and revolutionary for the future of projection research. With CUDA, we can design and

realize digital maps with more unconventional projections and add various real-time interactive functions, allowing more flexibility to cartographers and users. For instance, when users move the directional vector of oblique projections with the mouse, we can calculate and visualize the projected map simultaneously, thus they can observe the continuous change of the projected region and have a better understanding on map projections. Meanwhile, we are allowed to project a complex boundary with lots of sampled points in milliseconds without simplification. This is beneficial for high resolution maps. There is no doubt that CUDA has set off a revolutionary wave of map projection's innovation and opened a door to explore the future of projection.

Besides exploring the basic projection information of unknown maps, we believe machine learning will benefit us more in the future, like get more detailed information. In this research, we merely apply MLP, a basic neural network model, but there are lots of newer, faster, and more accurate machine learning models alternatively. Also, the input data in our research are all coordinates of boundary points instead of images. So, we can explore the way of starting from images and conducting the same machine learning based research, which will make our topic easier to apply and have more practical significance, as most of the maps are images in digital form.

Bibliography

[1] Milnor, J., 1969. A problem in cartography. The American Mathematical Monthly, 76(10), pp.1101-1112.