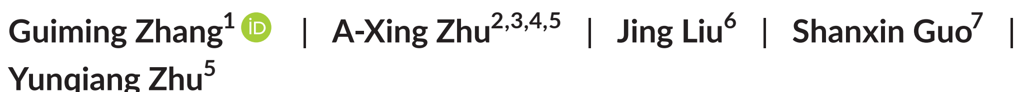
DOI: 10.1111/tgis.12730

RESEARCH ARTICLE

Transactions in GIS WILEY

PyCLiPSM: Harnessing heterogeneous computing resources on CPUs and GPUs for accelerated digital soil mapping

Guiming Zhang[1] | A-Xing Zhu[2,3,4,5] | Jing Liu[6] | Shanxin Guo[7] | Yunqiang Zhu[5]



[1]Department of Geography and the Environment, University of Denver, Denver, CO, USA

[2]Department of Geography, University of Wisconsin-Madison, Madison, WI, USA

[3]Jiangsu Center for Collaborative Innovation in Geographical Information Resource Development and Application, Nanjing, China

[4]School of Geography, Nanjing Normal University, Nanjing, China

[5]State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing, China

[6]Earth Science Department, Santa Monica College, Santa Monica, CA, USA

[7]Center for Geo-Spatial Information, Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China

Correspondence
Guiming Zhang, Geography and the Environment, University of Denver, Denver, CO, USA.
Email: guiming.zhang@du.edu

Funding information
University of Denver; National Natural Science Foundation of China, Grant/Award Number: 41601212 and 41871300

Abstract

Digital soil mapping (DSM) at high spatial resolutions over large areas often demands considerable computing power. This study aims to harness the heterogeneous computing resources on multi-core central processing units (CPUs) and graphics processing units (GPUs) to accelerate DSM by implementing PyCLiPSM, a parallel version of the iPSM (individual predictive soil mapping) algorithm which represents the type of geospatial algorithms that is data- and compute-intensive and highly parallelizable. PyCLiPSM was implemented in Python based on the PyOpenCL parallel programming library, which runs on any operating system and exploits the computing power of both CPUs and GPUs. Experiments show that PyCLiPSM can effectively leverage multi-core CPUs and GPUs to speed up DSM tasks. PyCLiPSM is open-source and freely available. Using PyCLiPSM as an example, we advocate implementing parallel geospatial algorithms using the PyOpenCL framework to harness the heterogeneous computing resources available to researchers and practitioners for accelerated geospatial analysis.

 © 2021 John Wiley & Sons Ltd

1 | INTRODUCTION

Soil maps characterizing spatial variations of soil properties (e.g., soil carbon) or soil classes inform environmental decision-making and are used as inputs to environmental models such as land surface models and hydrological models (Chaney et al., 2019; Zhu & Mackay, 2001). Digital soil mapping (DSM) algorithms are widely used for modeling soil–environment relationships and predicting soil maps based on soil sample data and environmental covariate data (McBratney, Mendonça Santos, & Minasny, 2003; Minasny & McBratney, 2016; Zhu et al., 2015). With the great advancements in earth observation technologies such as remote sensing, environmental covariates used for DSM are of increasingly fine spatial resolution (Chaney et al., 2016). Initiatives have also been started to compile and share soil sample databases across the globe (Batjes et al., 2017). These developments have made it possible to conduct fine-resolution DSM at national, regional, continental, or even global scales (Arrouays et al., 2014; Chaney et al., 2016; Hengl et al., 2017; Li, Malyshhev, & Sheviliakova, 2016; Ramcharan et al., 2018).

DSM at fine spatial resolutions over large areas involving large numbers of covariates and soil samples using sophisticated DSM algorithms demands high computing power. Some efforts have been made to address the computational challenges facing the DSM community. Padarian, Minasny, and McBratney (2015) explored Google's cloud-based platform (i.e., Google Earth Engine, GEE) for DSM. GEE allows access to Google's cloud computing resources, Earth observation data (e.g., *Landsat* imagery) hosted on the cloud, along with built-in algorithms that take advantage of the parallel cloud computing resources. As a result, GEE is good for pre-processing the massive remote sensing imagery to derive environmental covariates for DSM. However, it is difficult to implement highly specialized DSM algorithms entirely using GEE's built-in algorithms (Padarian et al., 2015). Besides, GEE imposes various limits on users (e.g., the number of soil samples a user can upload). Researchers have also employed high-performance computing (HPC) to accelerate DSM. For example, Jiang et al. (2016) developed a cyber platform for DSM using an HPC server as the computing back-end where the DSM algorithms were implemented using OpenMP (Open Multi-Processing) and run on multi-core central processing units (CPUs). In mapping soil series at 30-m resolution over the contiguous United States, Chaney et al. (2016) divided the large mapping area into smaller non-overlapping sub-areas and dispatched DSM tasks in each sub-area to the computation nodes of a supercomputer. However, not every DSM researcher or practitioner has access to HPC facilities (e.g., high-end computing server, supercomputer). For instance, early-career faculty and graduate researchers may not have the funds to build dedicated HPC facilities or pay for HPC services provided by third parties (e.g., cloud computing).

In recent years, graphics processing units (GPUs) have been widely used to accelerate spatial analyses (Tang, Feng, & Jia, 2015; Zhang, Zhu, & Huang, 2017). While multi-core CPUs place more emphasis on high performance on a small number of threads, many-core GPUs contain a large number of simpler processor cores (thousands or more) designed for a high degree of parallel processing. By exploiting data or task parallelism, GPUs can effectively speed up geoscience algorithms by a factor of tens or even hundreds compared to serial CPU implementations (Tang et al., 2015; Zhang et al., 2017).

There are only few applications of GPU computing for DSM. Many DSM algorithms (for reviews see Grunwald, 2009; McBratney et al., 2003) are generic statistical or machine learning algorithms (e.g., multiple linear regression, decision tree) that are often available in software packages running on CPUs. For example, the R statistical software (R Core Team, 2013) supports parallel code execution through the *foreach* library (Calaway & Weston, 2017). The Scikit-learn machine learning package supports model training utilizing multiple CPUs (Pedregosa et al., 2012). Kriging and geographically weighted regression (GWR) are often used for DSM (McBratney et al., 2003; Minasny & McBratney, 2016). Nonetheless, parallel implementations of the GWR algorithm utilizing multiple CPUs (Li, Fotheringham, Li, & Oshan, 2019) and kriging algorithms utilizing GPUs (Cheng, 2013; Gutiérrez de Ravé, Jiménez-Hornero, Ariza-Villaverde, & Gómez-López, 2014; Shi & Ye, 2013) have mostly been applied in other domains. To the best of our knowledge, there are few if any studies using parallel GWR or kriging algorithms for DSM. Recently, deep learning neural networks (e.g., convolutional neural networks), which have been gaining recognition across many different fields, have also been used for DSM (Behrens, Schmidt, MacMillan, & Viscarra

Rossel, 2018; Wadoux, Padarian, & Minasny, 2018). Training deep learning models is computationally intensive and thus parallel computing resources on multi-CPUs or GPUs are exploited to speed up model training, for example, using Google's TensorFlow (Abadi et al., 2016). Lastly, as aforementioned, some specialized DSM algorithms have been parallelized over multi-core CPUs (e.g., Jiang et al., 2016) or over supercomputer computing nodes (Chaney et al., 2016). Overall, there is a lack of implementations of DSM algorithms exploiting the massively parallel computing power of GPUs to speed up DSM applications.

Commodity desktop and laptop computers with multiple CPU cores and integrated and/or discrete graphics cards (i.e., GPUs) are commonplace. Dedicated HPC facilities are also often composed of computing nodes equipped with CPUs and GPUs (Guan, Shi, Huang, & Lai, 2016; Shi, Lai, Hiang, & You, 2014; Shi & Ye, 2013). Effectively harnessing the heterogeneous computing resources on CPUs and GPUs for DSM would address the pragmatic challenge facing DSM researchers who have access to only limited computing resources, and showcase an advancement toward the idea of "personal high-performance geospatial computing" (Zhang, 2010). It also informs better utilization of HPC for DSM.

This study aims to harness the heterogeneous computing resources on CPUs and GPUs for accelerated DSM. As an example, a parallel version of the iPSM (individual predictive soil mapping) algorithm (Zhu et al., 2015) representing the type of predictive soil mapping algorithms that is data- and compute-intensive and highly parallelizable was implemented in Python based on the PyOpenCL parallel programming library (Klöckner et al., 2012). Python is a high-level interpreted programming language that is dynamically typed (i.e., there is no need to specify data types for variables in the program; they are determined during run-time) and garbage-collected (Van Rossum, 1995). It features ease of use compared to low-level languages such as C/C++ (e.g., there is no need to deal with pointers and memory leaks) and high productivity, with many third-party packages available. For example, the SciPy package provides comprehensive and fast tools for mathematics, science, and engineering (Jones, Oliphant, & Peterson, 2001). OpenCL (Open Computing Language) is a parallel programming standard for heterogeneous computing systems (Stone, Gohara, & Shi, 2010). The most prominent advantage of OpenCL over other parallel programming libraries such as OpenMP, MPI (Message Passing Interface), and CUDA (Compute Unified Device Architecture) is that it provides a unified programming interface for both CPUs and GPUs. Moreover, it is hardware neutral, meaning it supports processors from virtually any vendor, while OpenMP and MPI only support CPUs and CUDA only supports NVIDIA GPUs. PyOpenCL is a Python package that wraps the OpenCL C/C++ programming interface for Python and carries out GPU run-time code generation (Klöckner et al., 2012). It allows programmers to easily program CPUs and/or GPUs directly in Python with only minimal C/C++ coding.

The parallel implementation of iPSM using PyOpenCL, which is named PyCLiPSM in this study, runs on any multi-core CPUs and GPUs that support the OpenCL standard, no matter whether they are standard commodity computers or HPC nodes, or whether they are manufactured by Intel, AMD, or NVIDIA. PyCLiPSM can effectively accommodate the diversity and heterogeneity of computing resources available to DSM researchers and practitioners. Using PyCLiPSM as an example, we advocate implementing more parallel DSM algorithms using the PyOpenCL framework for accelerated DSM.

2 | MATERIALS AND METHODS

2.1 | The iPSM algorithm for DSM

iPSM (Zhu et al., 2015) is an algorithm specially designed for DSM and has been used in a wide range of DSM studies (An et al., 2018; Yang, Qi, Zhu, Shi, & An, 2016; Zeng et al., 2016; Zhang & Zhu, 2019; S.-J. Zhang, Zhu, et al., 2016). iPSM represents the type of geospatial algorithms which is data- and compute-intensive and highly parallelizable.

2.1.1 | Overview of iPSM

The inputs to iPSM are a set of environmental covariate layers and a set of geo-referenced soil samples with measured soil property values or observed soil class labels. The outputs are a soil property or class map and a prediction uncertainty map. iPSM uses the soil–environment relationship at each individual soil sample location to predict soils at unsampled locations, assuming that locations of similar environment conditions have similar soils. It is an application of the principles of the Third Law of Geography (Zhu, Lu, Liu, Qin, & Zhou, 2018) for soil spatial prediction. It imposes no specific requirements on sample size or the spatial configuration of soil samples (Liu, Zhu, Rossiter, Du, & Burt, 2020; Zhu et al., 2015). An overview of iPSM is provided below. Readers interested in full details of the algorithm are referred to Zhu et al. (2015). iPSM predicts soil property or soil class at a prediction location (raster cell) following a two-step procedure: computation of environmental similarity, followed by prediction of soil property or class and quantification of prediction uncertainty.

At the first step, the environmental similarity between the prediction location and a sample location is computed for each individual environmental covariate. The overall similarity between the two locations on all covariates is then determined based on individual similarities. The environmental similarity between prediction location j and sample location i on the lth covariate (continuous; ratio or interval), $S_{i,j}^l$, is calculated as:

$$S_{i,j}^l = \exp\left[-\frac{\left(V_i^l - V_j^l\right)^2}{2\times\left(\frac{SD^l}{SD_i^l}\times SD^l\right)^2}\right] \quad (1)$$

where V_i^l and V_j^l denote the value of the lth covariate at sample location i and prediction location j, respectively. SD^l is the standard deviation of the covariate. SD_i^l is the pseudo "standard deviation" of the covariate values from V_i^l (not from the mean) and is computed by:

$$SD_i^l = \sqrt{\frac{\sum_{p=1}^{m}\left(V_p^l - V_i^l\right)^2}{m}} \quad (2)$$

where V_p^l is the value of the covariate at raster cell p, and m is the total number of raster cells in the mapping area. For categorical covariates (discrete; nominal or ordinal), the similarity is computed as:

$$S_{i,j}^l = \begin{cases} 0, & \text{if } V_i^l \neq V_j^l \\ 1, & \text{if } V_i^l = V_j^l \end{cases} \quad (3)$$

The overall environmental similarity between prediction location j and sample location i with respect to all L covariates, $S_{i,j}$, is then determined by taking the minimum of the environmental similarities on individual covariates:

$$S_{i,j} = \min\left(S_{i,j}^1, S_{i,j}^2, \ldots, S_{i,j}^L\right) \quad (4)$$

$S_{i,j}$ values lie between 0 and 1, with a higher value indicating higher environmental similarity.

The environmental similarity between prediction location j and each of the n sample locations can be computed following Equations (1–4).

At the second step, the soil property value or soil class at the prediction location is determined based on its environmental similarities to the n sample locations. The following equation is used for predicting soil property:

$$\hat{T}_j = \frac{\sum_{i=1}^{n} S_{i,j}\times T_i}{\sum_{i=1}^{n} S_{i,j}} \quad (5)$$

where $\hat{T}_j$ is the predicted soil property value at location j, and T_i the measured soil property value at sample location i. For soil class prediction, the predicted soil class at location j, $\hat{C}_j$, is assigned the observed soil class of the sample location whose environmental condition is most similar to location j:

$$\hat{C}_j = C_{\underset{i \in [1,n]}{\operatorname{argmax}} \, S_{i,j}}, \tag{6}$$

where C_i is the observed soil class at sample location i. Finally, for both soil property and class prediction, the prediction uncertainty at location j, U_j, is quantified as:

$$U_j = 1 - \max\left(S_{1,j}, S_{2,j}, \ldots, S_{n,j}\right) \tag{7}$$

U_j ranges from 0 to 1, with higher values indicating higher levels of prediction uncertainty.

Applying the above two-step procedure for soil prediction at every raster cell in the mapping area (m cells) results in a soil property or class map. iPSM has been implemented in the SoLIMSolutions software package (solim.geography.wisc.edu), which runs on a single CPU thread, and in the CyberSoLIM online platform (Jiang et al., 2016), which utilizes multi-CPU threads for computation.

2.1.2 | Complexity analysis

iPSM is data- and compute-intensive. If Equations (1) and (2) were followed literally, computing the environmental similarity between a prediction location and a sample location on one covariate would take $O(m)$ steps, because computing the standard deviation and pseudo "standard deviation" (deviation from covariate value at the sample location) takes $O(m)$ steps, where m is the number of raster cells in the covariate layer. As a result, the complexity of computing the overall similarity between the two locations on all L covariates (Equation 4) would be $O(mL)$. Therefore, predicting soil property or class (and quantifying prediction uncertainty) at one raster cell based on n soil samples (Equation 5) would take $O(mnL)$ steps. Finally, the complexity of predicting soil property or class at all m raster cells in the mapping area would be $O\left(m^2 nL\right)$. Thus, the run-time of the iPSM algorithm would be quadratic in the number of raster cells in the mapping area, and linear in the number of soil samples and the number of covariates.

2.1.3 | Optimization and parallelization opportunities

The above complexity analysis was for a "naïve" implementation of the iPSM algorithm. Note that the standard deviations and pseudo "standard deviations" of covariates are invariant to prediction locations; repetitively computing them at every prediction location is unnecessary. Optimizations can be adopted to avoid redundant calculations.

First, the standard deviations and means of the covariates, invariant to prediction locations, can be computed upfront in $O(mL)$ steps. They need only be computed once over the course of the algorithm and can be treated as constants in further calculations. Second, the pseudo "standard deviation" of a covariate on one soil sample location can be computed based on the computed standard deviation and mean of that covariate, as demonstrated below:

$$\begin{aligned}\left(SD_i^l\right)^2 &= \frac{\sum_{p=1}^m \left(V_p^l - V_i^l\right)^2}{m} = \frac{\sum_{p=1}^m \left[\left(V_p^l - \overline{V}^l\right) + \left(\overline{V}^l - V_i^l\right)\right]^2}{m} \\ &= \frac{\sum_{p=1}^m \left(V_p^l - \overline{V}^l\right)^2}{m} + \frac{2\left(\overline{V}^l - V_i^l\right)\sum_{p=1}^m \left(V_p^l - \overline{V}^l\right)}{m} + \frac{\sum_{p=1}^m \left(\overline{V}^l - V_i^l\right)^2}{m} \\ &= \left(SD^l\right)^2 + \left(\overline{V}^l - V_i^l\right)^2,\end{aligned} \tag{8}$$

where SD^l and $\overline{V}^l$ are the standard deviation and mean of the covariate. Therefore,

$$SD_i^l = \sqrt{\left(SD^l\right)^2 + \left(\overline{V}^l - V_i^l\right)^2} \tag{9}$$

Using Equation (9) to compute SD_i^l takes only $O(1)$ steps given the computed SD^l and $\overline{V}^l$ values. As a result, computing the pseudo "standard deviations" of all L covariates on all n soil sample locations takes $O(nL)$ steps. The pseudo "standard deviations" are also invariant to prediction locations and need only be computed once.

Utilizing the above optimizations, the complexity of computing environmental similarity following Equation (1) reduces from $O(m)$ to $O(1)$ given the computed SD^l and SD_i^l values. As a result, the complexity of predicting soil for all m raster cells reduces from $O\left(m^2nL\right)$ to $O(mnL)$. Altogether, the complexity of soil mapping using iPSM is $O(mL) + O(nL) + O(mnL) = O(mnL)$, where the first two terms represent the complexity of computing the standard deviations, means, and pseudo "standard deviations" of covariates. The overall complexity is linear in the number of cells, the number of soil samples and the number of covariates.

iPSM is also highly parallelizable. Data and computation parallelism can be exploited to speed up the algorithm. Obviously, predicting soil at a prediction location is independent of predicting soils at other locations. As a result, the two-step procedure in iPSM for soil prediction can be conducted in parallel at all raster cells.

2.2 | Implementation of PyCLiPSM

The parallel iPSM algorithm with optimizations, namely PyCLiPSM, was implemented in Python using PyOpenCL (Klöckner et al., 2012). As argued in Section 1, PyCLiPSM effectively accommodates the diversity and heterogeneity of computing resources available to DSM researchers and practitioners while imposing a minimal burden of coding on them. PyCLiPSM should run on CPUs or GPUs that support the OpenCL standard. CPUs and GPUs manufactured by Intel, AMD, NVIDIA and Apple all support OpenCL. This subsection provides an overview and details of the implementation of PyCLiPSM.

2.2.1 | Implementation overview

PyCLiPSM involves the cooperation of host-side functionalities and the device-side kernel (Figure 1). The host is the CPU thread that controls the workflow of the algorithm. The device can be either CPUs or GPUs that carry out computing tasks dispatched from the host by executing kernel functions in parallel.

At the beginning of PyCLiPSM, the host reads in data (environmental covariates, soil samples) from files into its main memory. It then computes data statistics, including the standard deviations and means of the covariates and pseudo "standard deviations" of the covariates for the soil sample locations. Next, the host allocates memory on the device and copies data and statistics to device memory. After that, the host invokes a kernel function call. The device then uses data in device memory as inputs to execute the kernel function in parallel for soil prediction and writes results in device memory. Each thread is responsible for computing soil property or soil class and uncertainty at one prediction location at a time. After the kernel execution completes, the host copies the results from device memory to main memory and writes the results to files. All tasks carried out by the device are put in a command queue. The low-level task mapping and task scheduling are transparent from PyCLiPSM as they are handled automatically by the underlying PyOpenCL library. Such simplicity of parallelization is a strength of PyCLiPSM.

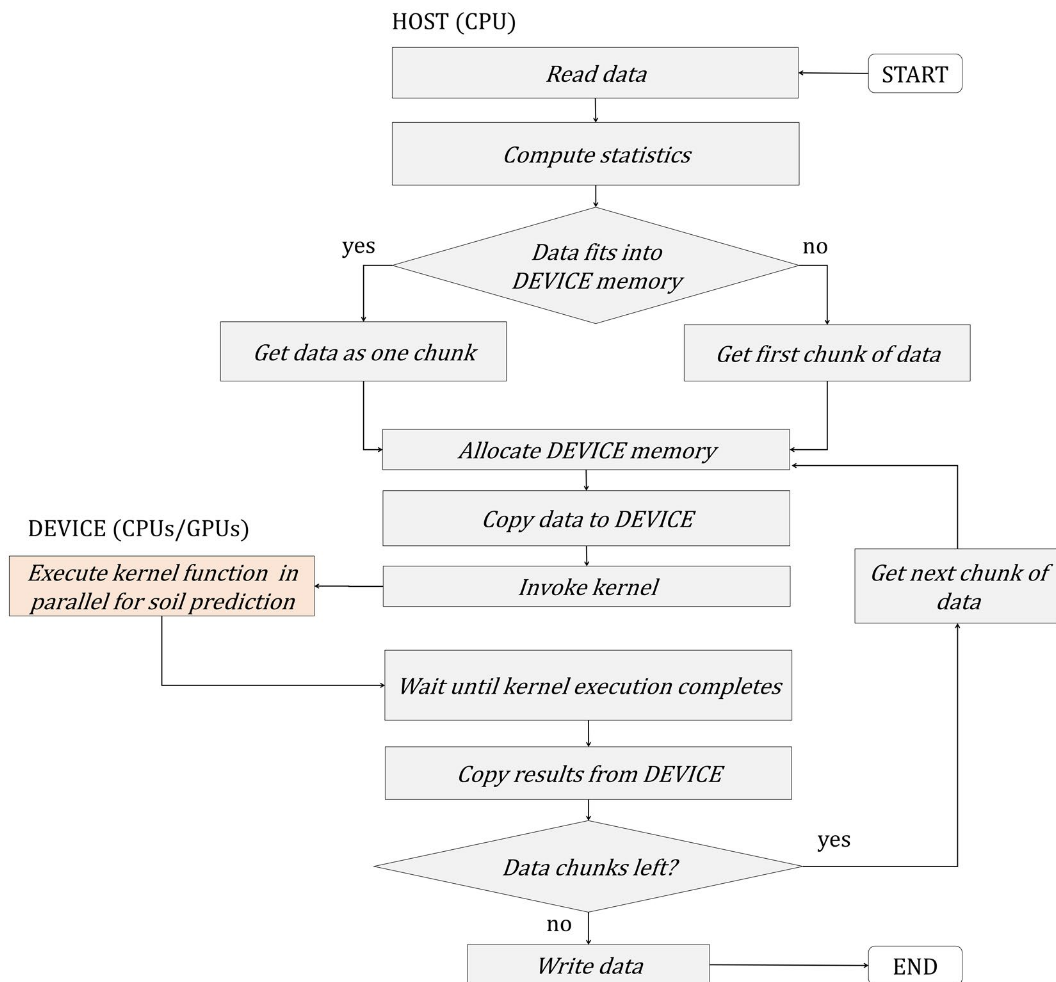


FIGURE 1 Overview of PyCLiPSM

2.2.2 | Implementation details

All the host-side functionalities were implemented in the Python programming language (version 2.7) based on the PyOpenCL package (https://pypi.org/project/pyopencl). Only the kernel function executed on the device was written in C. The source code of PyCLiPSM can be freely obtained from GitHub (https://github.com/Guiming/PyCLiPSM) under MIT license.

Data structures

The geographic coordinates and soil property values (or soil class labels) observed at soil sample locations were stored in comma-separated values (CSV) files. CSV files can be easily read and parsed using Python. Covariate raster data layers were stored in GeoTIFF files and masked to the same geographic extent. GeoTIFF files can be read and written using the Python wrapper of the Geospatial Data Abstraction Library (GDAL; https://pypi.org/project/GDAL). PyCLiPSM provides specially designed Python classes to represent and manipulate soil sample data and covariate data.

Covariate raster layers were read in and stored as two-dimensional arrays in the main (host) memory. Soil sample locations were overlaid with the covariate layers and covariate values at the sample locations were then

extracted. The extracted covariate values were stored in a 2D array where rows correspond to sample locations and columns correspond to covariates. This 2D array was then flattened to a 1D array (row major) and copied to device memory using the *Buffer* function in PyOpenCL.

Covariate layers also need to be copied to device memory for soil prediction. As the shape of the mapping area can be non-rectangular, raster cells (elements in the 2D arrays) outside of the mapping area would be filled with NODATA values where no soil prediction is made. Copying such 2D arrays with NODATA values altogether to device memory is unnecessary and would be a waste of resources. Thus, NODATA values in the 2D array of each covariate were stripped off and the remaining data values were flattened into a 1D array (row major). Note that the original 2D raster can be reconstructed based on data values in the 1D array and ancillary information (e.g., positions of NODATA values in the raster) maintained in the host-side Python class designed to manipulate raster layers. The resulting 1D arrays, each representing one covariate layer, were column-stacked to form a 2D array where rows correspond to raster cells (prediction locations) and columns correspond to covariates. Finally, this 2D array was flattened into a 1D array (row major) and copied to device memory using the PyOpenCL *Buffer* function. The array operations (e.g., stack, flatten) were performed very efficiently using functions provided by Numpy (Harris et al., 2020), a Python package that provides efficient algebraic operations on arrays and matrices.

Computing statistics

Given the 2D array containing values of covariates at all prediction locations (raster cells) (NODATA stripped off) and the 2D array containing values of covariates at soil sample locations (prior to being flattened into 1D arrays) on the host, covariate statistics (i.e., standard deviations, means, and pseudo "standard deviations" for sample locations) were computed by the host-side code using efficient Numpy built-in functions. The computed statistics were then copied to device memory using the PyOpenCL *Buffer* function.

Memory considerations

When soil mapping is conducted at fine spatial resolutions and/or over large areas (i.e., large numbers of prediction locations), the volume of covariate data may be too large to fit into host memory. In such cases, covariates for the whole mapping area cannot be read into host memory all at once. To address this challenge, the mapping area was divided into tiles and only one tile was read in at a time by the host for soil prediction. The dimension of a tile can be specified by the user or automatically determined based on the amount of host memory available (Section 3.5). A caveat is that, when reading covariates in tiles, covariate statistics (means and standard deviations) cannot be computed. Instead, the statistics were read from the metadata of GeoTIFF files, if pre-computed values were available, or computed on the fly using the *GetStatistics* function in GDAL.

The device, especially when a GPU is used, usually has even more limited memory space than the host. The data volume of covariates for the whole mapping area (or a tile of the mapping area) may exceed the device memory limit. To overcome this limitation, the 2D array containing values of covariates at all prediction locations (NODATA stripped off) were divided into chunks, each containing certain number of rows (each row represents one prediction location). Each chunk of data was then flattened into a 1D array and copied to device memory for soil prediction. This process was repeated until all chunks of data were processed, or in other words, soil was predicted at all prediction locations (Figure 1). The size of each data chunk (i.e., number of rows) was determined by considering the device memory size and the size of each row of data.

Kernel function

Soil prediction at one prediction location (raster cell) is independent of soil predictions at other locations. Therefore, a kernel function can be invoked by the host to execute on the device (multi-core CPU or GPU) for predicting soil at many prediction locations concurrently. Figure 2 shows the pseudo code of the kernel function implementing the iPSM algorithm for soil prediction. The actual kernel function was written in C. Large numbers of threads on

Algorithm 1: Soil prediction using the iPSM algorithm

```
1  function iPSM_Predict (ncols_X, nrows_samples, mode, msr_levels, SDs, sample_SDs,
     X, sample_X, sample_attributes, X_predictions, X_uncertainties)
   Input  : Number of covariates ncols_X
   Input  : Number of sample locations nrows_samples
   Input  : Prediction mode (0 - soil property and 1 - soil class) mode
   Input  : Covariate measurement levels (0 - nominal, 1 - ordinal, 2 - interval, and 3 - ratio) msr_levels
   Input  : Covariate standard deviations SDs
   Input  : Covariate pseudo standard deviations sample_SDs
   Input  : Covariate values at prediction locations X
   Input  : Covariate values at sample locations sample_X
   Input  : Soil property values or class labels at sample locations sample_attributes
   Output: Soil property values or class labels at prediction locations X_predictions
   Output: Prediction uncertainty at prediction locations X_uncertainties
2  i = get_global_id(0)
3  for j in 0 : nrows_samples do
4      min_sim = 9999.0
5      for k in 0 : ncols_X do
6          tmp_sim = 0.0
7          if msr_levels[k] == 0 OR msr_levels[k] == 1 then
8              if X[i * ncols_X + k] == sample_X[j * ncols_X + k] then
9                  tmp_sim = 1.0
10             else
11                 tmp_sim = 0.0
12             end
13         else
14             diff = X[i * ncols_X + k] − sample_X[j * ncols_X + k]
15             denom = SDs[k] * SDs[k]/samples_SDs[j * ncols_X + k]
16             tmp_sim = exp(−0.5 * (diff * diff)/(denom * denom))
17         end
18         if tmp_sim < min_sim then
19             min_sim = tmp_sim
20         end
21     end
22     similarities[j] = min_sim
23 end
24 if mode == 0 then
25     sum_sim = 0.0
26     max_sim = 0.0
27     sum_weighted_attribute = 0.0
28     for j in 0 : nrows_samples do
29         if similarities[j] > max_sim then
30             max_sim = similarities[j]
31         end
32         sum_sim+ = similarities[j]
33         sum_weighted_attribute+ = similarities[j] * sample_attributes[j]
34     end
35     X_predictions[i] = sum_weighted_attribute/sum_sim
36     X_uncertainties[i] = 1.0 − max_sim
37 else
38     max_sim_idx = −1
39     max_sim = 0.0
40     for j in 0 : nrows_samples do
41         if similarities[j] > max_sim then
42             max_sim = similarities[j]
43             max_sim_idx = j
44         end
45     end
46     X_predictions[i] = sample_attributes[max_sim_idx]
47     X_uncertainties[i] = 1.0 − max_sim
48 end
```

FIGURE 2 Pseudo code of the kernel function implementing the iPSM algorithm

the device execute the kernel function in parallel to perform soil prediction at prediction locations concurrently. Each thread takes data in the device memory as inputs and writes the results back to device memory.

2.3 | Computation performance evaluation

2.3.1 | Variants of the iPSM algorithm

The iPSM algorithm was implemented/configured into variants that differ in terms of the parallel programming library in use, computing device, number of computing threads, and optimization (Table 1). Besides using PyOpenCL, iPSM was also implemented using *pathos.multiprocessing*, a native Python parallel programming library for exploiting computing power on CPUs (https://pypi.org/project/pathos). The sole purpose is to compare the performance of CPU multi-threading implemented using Pathos against that using PyOpenCL. Evaluating the computation performance of these algorithms allows us to examine the effects of the proposed optimization and parallelization strategies (Section 2.1.3) and how computation performance is impacted by the underlying programming library, computing device, and computing threads involved.

2.3.2 | Computing platforms

The computation performance of the variants of iPSM (e.g., run-time) was tested on two computing platforms with distinct computing capabilities, operating systems, and hardware vendors (Table 2). One is a high-end desktop workstation running Ubuntu equipped with an NVIDIA graphics card. The other is a standard commodity laptop computer running Windows with an AMD graphics card.

2.3.3 | Performance metrics

Execution times of the algorithms were recorded as a computation performance measure. Reported execution time is the average execution time of 10 runs in all experiments. The total execution time was broken down into time spent on reading data from/writing data to disk drive (read/write), transferring data between host and

TABLE 1 Variants of the iPSM algorithm that differ in parallel programming library, computing device, number of computing threads, and optimization

Library	Device	# of threads	Optimization	Algorithm variant
PyOpenCL	GPU	Many-thread	Optimized	CL_GPU
			Naïve	CL_GPU_NAIVE
	CPU	Multi-thread	Optimized	CL_CPU
			Naïve	CL_CPU_ NAIVE
		Single-thread	Optimized	CL_CPU1
			Naïve	CL_CPU1_ NAIVE
Pathos	CPU	Multi-thread	Optimized	MP_CPU
			Naïve	MP_CPU_ NAIVE
		Single-thread	Optimized	MP_CPU1
			Naïve	MP_CPU1_ NAIVE

TABLE 2 Specifications of the two computing platforms

Platform	Model	Operating system	CPU	GPU
Desktop	Dell Precision 5820 workstation	Ubuntu 18.04 (64-bit)	Intel Xeon W-2145 CPU (16 logical processors); 3.7 GHz max clock speed; 64 GB of memory	NVIDIA Quadro P4000; 1.5 GHz max clock speed; 8 GB of memory
Laptop	Dell Inspiron 14 5000 series	Windows 8.1 (64-bit)	Intel i-7 4510U CPU (4 logical processors); 2.6 GHz max clock speed; 8 GB of memory	AMD Radeon R7 M260; 0.9 GHz max clock speed; 2 GB of memory

device (data transfer) and actual soil prediction (compute). Read/write and/or data transfer are costs common to all variants of the algorithm. Therefore, compute time was primarily compared (e.g., to compute speedup ratio) to reveal differences in computation performance among the algorithms.

The accuracy of the predicted soil map was not formally assessed in experiments in this study (except for Section 3.3). The sole purpose of this study was to evaluate the computation performance of the algorithms and how the performance was affected by various factors such as soil sample size, the number of covariates, and soil mapping spatial resolution. Evaluation of prediction accuracy of iPSM and how it may be affected by such factors is not the goal of this study. Nonetheless, correctness of the algorithms was examined by checking to ensure soil maps predicted from all variant algorithms were the same. The prediction accuracy of iPSM has been extensively examined in other studies. Readers interested in this topic are referred to Yang et al. (2016), Zeng et al. (2016), Zhang, Huang, Zhu, and Keel (2016), Zhu et al. (2018), and Zhang and Zhu (2019). In particular, An et al. (2018) documented the accuracies of mapping A-horizon soil organic matter (SOM) content in the same study area using the same data set (Section 2.3.4).

2.3.4 | Experimental data and design

DSM experiments were conducted in Anhui Province (134,000 km^2) in eastern China to evaluate the computation performance of the variants of iPSM. In the study area, there were 282 soil sample locations (Figure 3) at which soil samples were taken and the SOM content (g/kg) of A-horizon soil was measured (mean 24.69 g/kg; standard deviation 13.32 g/kg). Among the sample locations, 129 were designed based on expert knowledge and 153 following a stratified random sampling strategy (An et al., 2018). A set of 12 environmental covariates were used for mapping A-horizon SOM in the study area, including annual average temperature, annual accumulated temperature above 10°C, annual average precipitation, annual average evaporation, arid index, moisture index, slope gradient, planform curvature, profile curvature, topographic wetness index, parent material, and normalized difference vegetation index. The covariate data layers were at 90-m spatial resolution. Further details of the soil sample data and covariate data can be found in An et al. (2018).

Soil sample sets of varied sample sizes were obtained by randomly selecting samples from the 282 soil samples and were used in soil prediction experiments to examine impacts of soil sample size on the computation performance of the algorithms (Section 3.3). For some performance evaluation experiments where a larger number of soil samples were needed, synthetic soil samples were generated and used (Section 3.4). In addition, covariates of varied spatial resolution (i.e., covariate cell size) were used in soil prediction experiments to investigate the impacts of mapping spatial resolution on computation performance (Section 3.3). To this end, the covariates at 90-m spatial resolution were resampled to 1,000-, 450-, 270-, 30-, and 10-m resolutions (Table 3). Note that at 10-m resolution, the volume of the covariate data (117 GB) exceeds the amount of memory available even on the desktop workstation (64 GB). Moreover, to assess the effects of the number of covariates, subsets of the 12 covariates containing varied numbers of covariates were used in soil prediction experiments (Section 3.3).

The above experimental design is sufficient for evaluating the impacts of the various factors on the computation speed of PyCLiPSM. iPSM essentially predicts the soil property value at a prediction location as a weighted average of the soil property values at the sample locations. Thus, there are no model parameters to optimize in this process. For some other digital soil mapping algorithms (e.g., random forest) where model training involves iteratively optimizing model parameters, additional factors may also affect computation speed (minimal training accuracy, relative importance of covariates, sample representativeness, etc.) as they would influence the rate of convergence.

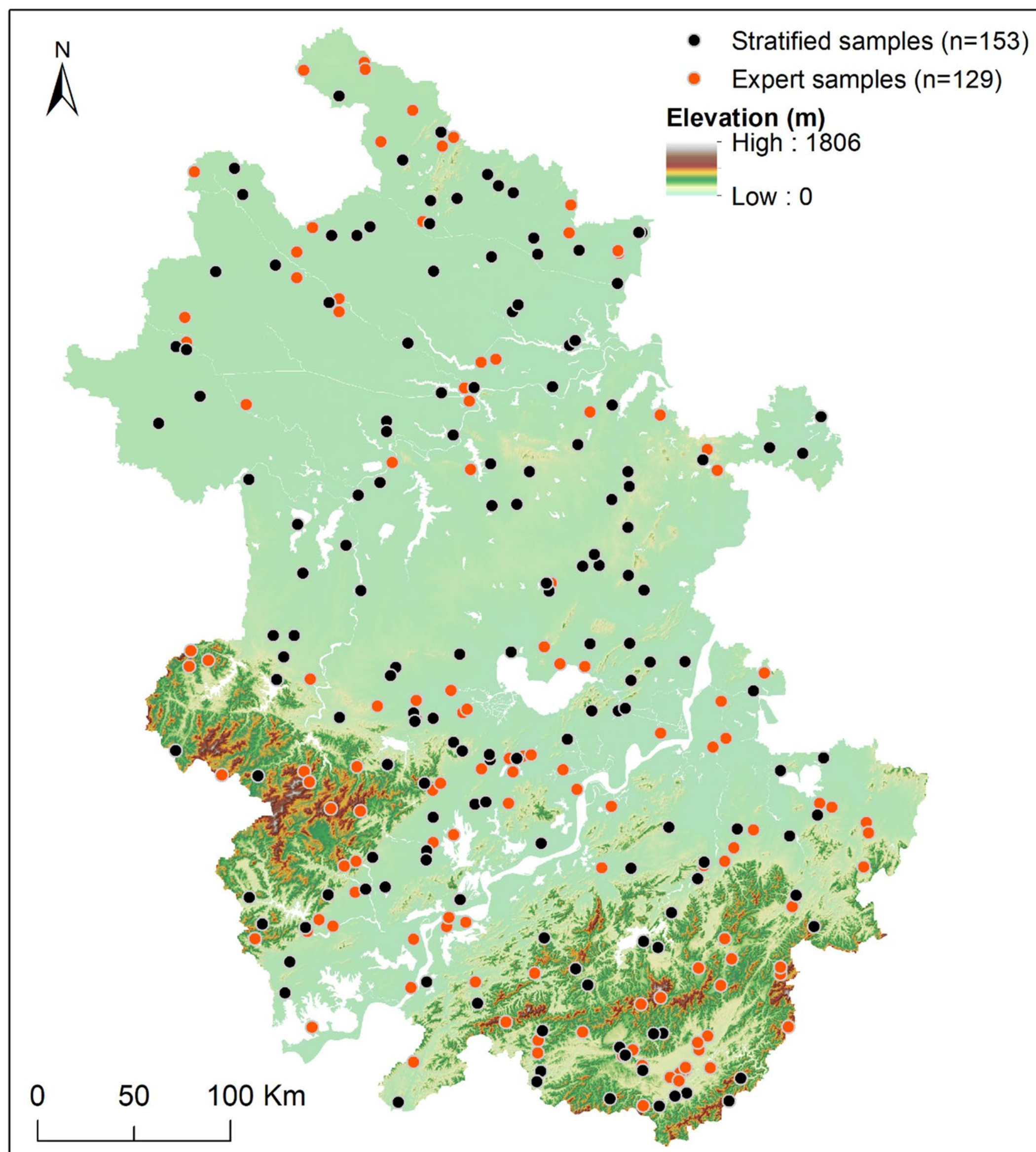


FIGURE 3 Spatial distribution of soil samples in the study area

3 | RESULTS AND DISCUSSION

3.1 | Effectiveness of optimization

Soil prediction experiments were conducted on a small data set to evaluate the effectiveness of the optimization (Section 2.1.3). The small data set was intentionally designed so that the naïve algorithms could complete within a reasonable time-frame and their execution times could be recorded. Five soil samples randomly selected from the full set of samples and covariates at 1 km resolution were used in these experiments using variants of the iPSM algorithm (Section 2.3.1) running on the desktop workstation.

TABLE 3 Covariates used for soil mapping at varying spatial resolutions

Resolution (m)	1,000	450	270	90	30	10
Rows	582	1,293	2,154	6,463	19,389	58,167
Columns	450	1,001	1,668	5,004	15,012	45,036
Files size (compressed GeoTIFF)	6 MB	26 MB	71 MB	504 MB	3 GB	18 GB
Data size in memory	12 MB	59 MB	164 MB	1.4 GB	13 GB	117 GB
Prediction cells	1.3×10^5	6.5×10^5	1.8×10^6	1.6×10^7	1.5×10^8	1.3×10^9

TABLE 4 Execution time (seconds) of variants of the iPSM algorithm. Covariates at 1 km resolution and five soil samples were used in the experiments

Algorithm	Total	Read/write	Data transfer	Compute
CL_GPU	0.48	0.16	0.00	0.32
CL_GPU_NAIVE	6.19	0.15	0.00	6.04
CL_CPU	0.76	0.14	0.00	0.62
CL_CPU_NAIVE	22.93	0.17	0.01	22.76
CL_CPU1	0.26	0.14	0.00	0.12
CL_CPU1_NAIVE	1,712.49	0.14	0.01	1,712.35
MP_CPU	12.77	0.16	0.00	12.61
MP_CPU_NAIVE	1,779.47	0.15	0.00	1,779.31
MP_CPU1	39.18	0.14	0.00	39.04
MP_CPU1_NAIVE	5,339.71	0.18	0.00	5,339.53

The proposed optimization effectively accelerated soil prediction using iPSM (Table 4). Naïve implementations of iPSM were very slow even on this small data set. For example, the PyOpenCL-based implementation without optimization running on a single CPU thread (CL_CPU1_NAIVE) on the desktop workstation took 28.5 min in total. Optimized iPSM algorithms were up to orders of magnitude faster than naïve implementations. For instance, in terms of compute time, the PyOpenCL-based algorithm with optimization running on a single CPU thread (CL_CPU1) took only 0.12 s, which was over 14,000 times faster than CL_CPU1_NAIVE. The optimized algorithm running on GPU (CL_GPU) was 18 times faster than the naïve algorithm (CL_GPU_NAIVE). The Pathos-based algorithm without optimization running on multiple CPU threads (MP_CPU) was 136 times faster than the naïve algorithm (MP_CPU_NAIVE).

It was also observed that iPSM implementations based on PyOpenCL were up to hundreds of times faster (in terms of compute time) than those based on Pathos (Table 4). The PyOpenCL-based implementation running on multiple CPU threads (CL_CPU) was 20 times faster than the implementation based on Pathos multiprocessing (MP_CPU) running on the same CPUs with the same number of threads. The PyOpenCL-based implementation running on a single CPU thread (CL_CPU1) was 325 times faster than the Pathos-based implementation (MP_CPU1).

Pathos multi-processing was much slower than PyOpenCL, possibly for several reasons. First, Python is an interpreted language, meaning it executes with the help of the interpreter instead of the compiler, which slows it down. Therefore, conducting compute-intensive tasks using Python code is inefficient. In contrast, the kernel function used in PyOpenCL was written in C/C++ and was compiled before execution. Second, when using multi-threading in Python, the global interpreter lock (GIL) can degrade performance as it protects access to Python

objects, preventing multiple threads from executing Python bytecodes at once (Beazley, 2010). Parallel computing using PyOpenCL is not susceptible to Python GIL and thus can be much more efficient.

3.2 | Effectiveness of parallelization

Soil prediction experiments were conducted to evaluate effectiveness of the parallelization of iPSM (Section 2.1.3). The 129 expert soil samples and covariates at 90 m spatial resolution were used for soil prediction using PyCLiPSM running sequentially on a single CPU thread (CL_CPU1) and running in parallel on GPU (CL_GPU) and multiple CPU threads (CL_CPU) (16 and 4 threads on the desktop and laptop, respectively). Execution times were recorded and compared. Accuracy of the predicted soil map (Figure 4) was validated by comparing predicted and observed SOM content values at the 153 stratified soil sample locations. The mean absolute error (MAE) computed based on the validation soil samples was 8.74 g/kg, which is consistent with the accuracies reported in An et al. (2018).

Parallelizing iPSM using PyOpenCL effectively accelerated the algorithm (Table 5). Soil prediction using parallelized iPSM was much faster than using the sequential implementation. On the desktop, CL_GPU and CL_CPU were 27.3 and 21.2 times faster (in terms of compute time) than CL_CPU1, respectively. On the laptop computer, CL_GPU and CL_CPU were 35.8 and 22.5 times faster than CL_CPU1, respectively.

All variants of PyCLiPSM (CL_CPU1, CL_CPU and CL_GPU) on the desktop were about 7–10 times faster (in terms of compute time) than on the laptop (Table 5). This is because the desktop has a larger number of more powerful CPU cores and a more advanced GPU. Moreover, read/write was about 7 times faster and data transfer about 6 times faster on the desktop. This was expected given that the standard commodity laptop computer does not have as fast storage and data bandwidth as the high-end desktop workstation.

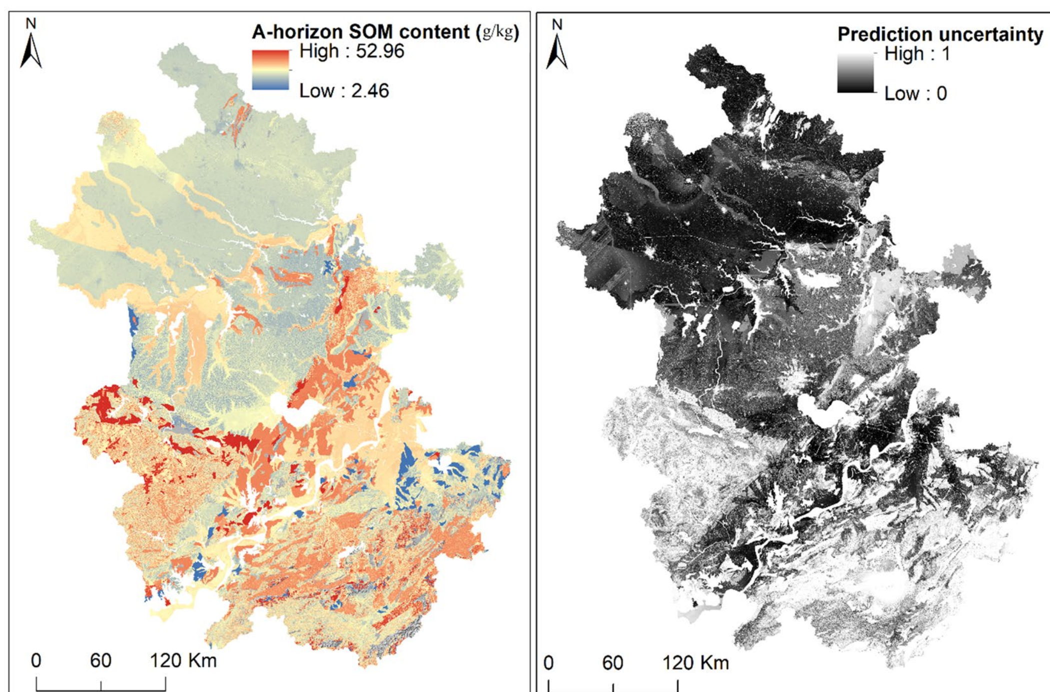


FIGURE 4 Maps of predicted A-horizon soil organic matter content and prediction uncertainty

TABLE 5 Execution time (seconds) of PyCLiPSM. Covariates at 90 m resolution and 129 soil samples were used in the experiments

		CL_GPU	CL_CPU	CL_CPU1
Desktop	Total	14.76	16.27	154.55
	Read/write	9.08	9.06	9.05
	Data transfer	0.37	0.35	0.35
	Compute	5.31	6.85	145.14
Laptop	Total	105.35	128.08	1,402.03
	Read/write	66.08	66.08	65.69
	Data transfer	2.04	2.68	2.71
	Compute	37.24	59.32	1,333.63

3.3 | Impacts of soil samples, covariates, and mapping resolution

Soil sample subsets of varied sample size and subsets of the covariates at various spatial resolutions were used in soil prediction experiments on the desktop workstation to investigate their impacts on the computation performance of PyCLiPSM. Experiments show that the responses of PyCLiPSM computation speed to these factors are consistent with the complexity analysis of the algorithm (Section 2.1.2), as detailed below.

The compute time of PyCLiPSM increased linearly as soil sample size increased (Table 6). The speedup ratio, computed as the ratio of the compute time on a single CPU thread to that on multiple CPU threads or on a GPU (G. Zhang et al., 2017; G. Zhang, Huang, et al., 2016), increased for larger sample sizes. The speedup ratio of CL_CPU increased from 13.2 to 27.7 and that of CL_GPU increased from 14.2 to 40.6 when sample size increased from 50 to 250. Note that the speedup ratio of CL_CPU at sample size 100 and beyond exceeded linear speedup (i.e., 16 on the desktop workstation with 16 CPU threads). This may be due to the different kernel functions used by CL_CPU1 and CL_CPU. The single-thread kernel function contains a nested loop (i.e., the outer loop over raster cell and the inner loop over sample locations) while the multi-thread kernel function contains only the inner loop. The latter could be turned into more efficient run-time code by PyOpenCL (Klöckner et al., 2012).

As for the impacts of the covariates, the read/write, data transfer, and compute time of PyCLiPSM increased linearly as more covariates were used for soil prediction (Table 7). Greater speedup ratios were achieved by running PyCLiPSM on multiple CPU threads and on GPUs over larger number of covariates. The speedup ratio of CL_CPU increased from 21.4 to 29.4 and speedup of CL_GPU increased from 28.3 to 49.5 when the number of covariates increased from 2 to 12.

Regarding mapping resolution, the execution time of PyCLiPSM increased approximately quadratically as mapping resolution improved (Table 8). For example, when mapping resolution improved from 90 to 30 m (by a factor of 3), the read/write, data transfer, and compute time all approximately increased by a factor of 9. This was expected, given that the total number of prediction cells has an inverse quadratic relation with cell size. The speedup ratio generally increased on finer mapping resolutions. When the cell size decreased from 1,000 to 30 m, the speedup ratio of CL_GPU and CL_CPU increased from 7.8 to 45.5 and from 13.7 to 29.0, respectively.

3.4 | Scalability to large number of soil samples

Synthetic soil samples were used to evaluate the scalability of PyCLiPSM to larger numbers of soil samples (the 282 real soil samples were too small a sample size for this evaluation). Three sets of synthetic soil samples of varied

TABLE 6 Impacts of soil sample size on compute time (seconds) and speedup ratio of PyCLiPSM. Covariates at 90 m resolution were used in the experiments

Sample size		50	100	150	200	250
Compute time	CL_GPU	4.17	4.79	5.36	6.09	6.67
	CL_CPU	4.50	5.88	7.18	8.44	9.76
	CL_CPU1	59.25	110.87	163.33	219.89	270.56
Speedup ratio	CL_GPU	14.21	23.15	30.47	36.11	40.56
	CL_CPU	13.16	18.86	22.75	26.06	27.72

TABLE 7 Impacts of the number of covariates on execution time (seconds) and speedup ratio of PyCLiPSM. Covariates at 90 m resolution and 282 soil samples were used in the experiments

Number of covariates		2	4	6	8	10	12
CL_GPU	Read/write	3.68	4.86	5.84	7.11	8.07	8.93
	Data transfer	0.07	0.13	0.19	0.25	0.31	0.37
	Compute	2.31	3.23	4.23	5.11	6.31	6.19
CL_CPU	Read/write	3.68	4.84	5.80	7.09	8.04	8.80
	Data transfer	0.07	0.13	0.18	0.24	0.30	0.35
	Compute	3.05	4.65	6.22	7.80	9.32	10.40
CL_CPU1	Read/write	3.68	4.86	5.80	7.11	8.02	8.81
	Data transfer	0.07	0.13	0.19	0.24	0.30	0.35
	Compute	65.17	118.42	170.87	222.61	277.20	306.19
Speedup ratio	CL_GPU	28.26	36.65	40.38	43.58	43.90	49.47
	CL_CPU	21.39	25.48	27.46	28.53	29.74	29.44

TABLE 8 Impacts of mapping spatial resolution on execution time (seconds) and speedup ratio of PyCLiPSM. Two hundred eighty-two soil samples were used in the experiments

Mapping resolution		1,000 m	450 m	270 m	90 m	30 m
CL_GPU	Read/write	0.22	0.84	1.08	8.93	76.98
	Data transfer	0.00	0.00	0.04	0.37	3.25
	Compute	0.34	0.51	1.00	6.19	60.71
CL_CPU	Read/write	0.22	0.84	1.08	8.80	76.93
	Data transfer	0.00	0.02	0.04	0.35	3.14
	Compute	0.19	0.46	1.18	10.43	95.10
CL_CPU1	Read/write	0.22	0.84	1.08	8.81	76.96
	Data transfer	0.00	0.02	0.04	0.35	3.17
	Compute	2.67	12.30	34.00	306.19	2,761.41
Speedup ratio	CL_GPU	7.78	24.18	34.11	49.47	45.48
	CL_CPU	13.72	26.47	28.79	29.35	29.04

TABLE 9 Scalability of PyCLiPSM to large numbers of (synthetic) soil samples

Algorithm		CL_GPU			CL_CPU			CL_CP1		
		270 m	90 m	30 m	270 m	90 m	30 m	270 m	90 m	30 m
Compute time	*n* = 1,000	2.08	16.10	143.00	3.35	29.60	266.00	122.00	1,080.00	9,650.00
	n = 2,000	3.47	28.80	256.00	6.38	56.70	510.00	247.00	2,170.00	19,600.00
	n = 5,000	7.74	66.79	599.19	15.30	137.17	1,238.66	599.00	5,390.00	48,753.73
Speedup ratio	*n* = 1,000	58.65	67.08	67.48	36.42	36.49	36.28	1.00	1.00	1.00
	n = 2,000	71.18	75.35	76.56	38.71	38.27	38.43	1.00	1.00	1.00
	n = 5,000	77.39	80.70	81.37	39.15	39.29	39.36	1.00	1.00	1.00

TABLE 10 Impact of tile dimension and using multi-thread on time (seconds) spent on reading covariates (10 m resolution) into host memory on the desktop workstation

Raster dimension	GeoTIFF file block size	Tile dimension	Single-thread	Multi-thread
45,036 × 58,167	45,036 × 128	45,036 × 3,584 (3584 = 128 × 28)	590.45	515.06
		12,704 × 12,704	1,195.51	570.92

size (n = 1,000, 2,000 and 5,000) were generated by randomly selecting locations from the study area. SOM content values at these locations were extracted from the SOM content map predicted based on the 129 expert soil samples (Figure 4) and were treated as the "observed" SOM content values. Soil prediction experiments using the synthetic samples and covariates at three different resolutions (270, 90, and 30 m) were conducted on the desktop workstation. The compute times for PyCLiPSM running on a GPU (CL_GPU), multiple CPU threads (CL_CPU), and a single CPU thread (CL_CPU1) were recorded, based on which the speedup ratios were computed and compared.

The speedup ratio of CL_GPU varied from about 60 to 80, increasing as soil sample size increased and as mapping resolution improved (Table 9). The speedup ratio of CL_CPU was between 35 and 40, in a smaller range compared to CL_GPU. It increased as soil sample size increased but remained relatively stable as mapping resolution improved. The above observations suggest that PyCLiPSM running on GPUs is better at achieving massive speedups for soil mapping tasks at fine spatial resolutions involving large numbers of soil samples.

3.5 | Addressing input/output and memory constraints

When the data volume of the covariates exceeds available host memory, PyCLiPSM reads in covariate data one tile at a time for soil prediction (Section 2.2.2). Tile dimension greatly influences the speed of reading covariates. Covariate GeoTIFF files are stored in blocks on disk drive. Experiments show that setting tile dimension to multiples of the block size of the GeoTIFF files resulted in faster reading speed (Table 10) as the tile dimension coincides with the layout of the files on the disk drive. PyCLiPSM determines the default tile dimension based on this observation. Moreover, time spent on reading covariates is a significant portion of the total execution time of PyCLiPSM. To address the input/output (I/O) bottleneck, PyCLiPSM implemented an option for reading covariates using multiple threads concurrently where each thread reads in one covariate. Multi-thread reading can be faster than single-thread reading on large files or tiles (Table 10), but it consumes more memory space and incurs an additional overhead for thread management.

Covariate data residing in host memory for soil prediction were split into smaller chunks that can fit into device memory, and only one chunk was transferred to the device at a time for soil prediction (Section 2.2.2). Chunk size (i.e., the number of prediction locations in each chunk) is a critical parameter affecting the computation performance of PyCLiPSM. Experiments show that setting chunk size to multiples of the device maximum work item size resulted in faster computation (Table 11). With such a setting, the per-chunk workload aligns with the number of threads launched as a work group to execute kernel function concurrently; therefore, no threads are left idle waiting for other threads to finish computing. This heuristic was implemented in PyCLiPSM to determine the default chunk size.

Equipped with the above solutions addressing I/O and memory constraints, PyCLiPSM can be used for DSM over large areas at fine spatial resolutions. It was applied to conduct soil prediction using the 10 m covariates (117 GB) and all 282 soil samples on the desktop (64 GB memory) and the laptop (8 GB memory). On the desktop, it took about 24 min (total execution time) using the GPU as the computing device and 29 min using the CPU (Table 12). On the laptop, it took roughly 2 and 3.4 hr using the GPU and the CPU, respectively.

TABLE 11 Impact of data chunk size on PyCLiPSM compute time (seconds) on the desktop workstation. Covariates at 90 m resolution (16,302,679 prediction locations) and 282 soil samples were used in the experiments

		1 chunk		2 chunks		
Device	Max. work item size	Chunk size	Compute time	Chunk 1 size	Chunk 2 size	Compute time
GPU	1,024	16,302,679	86.34	16,302,080 (1,024 × 15,920)	599	6.19
CPU	8,192	16,302,679	39.30	16,302,080 (8,192 × 1,990)	599	10.43

TABLE 12 Execution time (seconds) of PyCLiPSM. Covariates at 10 m resolution and 282 soil samples were used in the experiments

		CL_GPU	CL_CPU
Desktop	Total	1,402.39	1,740.62
	Read/write	781.28	835.92
	Data transfer	30.14	28.57
	Compute	590.98	876.12
Laptop	Total	7,028.33	12,036.95
	Read/write	3,637.24	3,737.80
	Data transfer	138.31	166.01
	Compute	3,252.78	8,133.15

4 | CONCLUSIONS

As the experimental results have demonstrated, the optimization and parallelization implemented in PyCLiPSM effectively speed up DSM tasks. Compared to the non-optimized version, the optimized version was tens of thousands of times faster. The parallel version was tens of times faster than the sequential version. Moreover, parallelizing the iPSM algorithm using PyOpenCL was much more effective than using Pathos in exploiting the computing power on multi-CPUs to speed up DSM. The PyOpenCL version was hundreds of times faster than the Pathos version. DSM tasks at a finer mapping spatial resolution, over a larger mapping area, or involving a larger number of soil samples and covariates often demand more computing power. With the acceleration brought by PyCLiPSM, such DSM tasks can be conducted efficiently. It was observed in the experiments that, for large DSM computing tasks, running PyCLiPSM on GPUs can bring a higher level of acceleration than on multi-core CPUs due to the massively parallel computing power on GPUs.

Additional enhancements could be implemented to further improve PyCLiPSM. First, I/O can still be a bottleneck for PyCLiPSM, although the compute-intensive parts of the algorithm have been greatly accelerated through optimization and parallelization. When reading very large covariate data, read/write can take up over 50% of the total execution time even using multi-thread reading (Section 3.9). Thus, one area of enhancement is to improve I/O efficiency for PyCLiPSM. Second, PyCLiPSM currently can utilize heterogeneous computing environments with distinct computing resources (e.g., the kernel function runs on either multi-core CPU or GPU). It does not fully harness the heterogeneous computing resources in the same environment. Future enhancements could extend PyCLiPSM to exploit computing power on multi-core CPUs and GPUs at the same time, in a similar way to that presented in Wang, Guan, and Wu (2017).

The most prominent features of PyCLiPSM are its ease of use and its compatibility with heterogeneous computing environments. PyCLiPSM was implemented using the user-friendly and cross-platform Python language

based on the PyOpenCL parallel programming library. It can therefore run on any operating system (e.g., Windows, Linux, MacOS) and exploit computing power on multi-core CPUs and GPUs manufactured by virtually any vender provided that they support the OpenCL standard. Users can easily adapt PyCLiPSM to meet their own needs with basic Python coding knowledge and minimal C/C++ coding skills. The free and open-source PyCLiPSM provides a framework for implementing other geospatial algorithms using PyOpenCL. It offers specially designed Python classes for representing and manipulating soil samples and covariates and is extensible to accommodate various data formats. The tiled reading strategy implemented in PyCLiPSM is generally applicable to handling large volumes of covariate data. It also establishes the workflow of implementing geospatial algorithms using PyOpenCL. For implementing other algorithms, one need only analyze the parallelism within the algorithms and implement kernel functions accordingly.

As demonstrated in the experiments, PyCLiPSM can accelerate DSM tasks on high-end computing workstations as well as commodity personal computers that have distinct computing capabilities. It showcases the advocacy of "personal high-performance geospatial computing" (Zhang, 2010) in the DSM application domain. Using PyCLiPSM as an example, we advocate implementing more parallel geospatial algorithms based on the PyOpenCL framework toward harnessing heterogeneous computing resources available to researchers and practitioners for accelerated geospatial analysis.

ACKNOWLEDGMENTS

Supports to Guiming Zhang through the Faculty Start-up Funds and the Faculty Research Fund Grant at the University of Denver are greatly appreciated. Supports to Shanxin Guo and A-Xing Zhu from the National Natural Science Foundation of China (Grant No. 41601212, 41871300) and supports to A-Xing Zhu through the Vilas Associate Award, the Hammel Faculty Fellow Award, and the Manasse Chair Professorship from the University of Wisconsin-Madison are acknowledged.

CONFLICT OF INTEREST

The authors have no conflict of interest to declare.

ORCID

Guiming Zhang https://orcid.org/0000-0001-7064-2138

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Devin, M. (2016). *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*. Preprint, arXiv1603.04467.

An, Y., Yang, L., Zhu, A-X., Qin, C., & Shi, J. (2018). Identification of representative samples from existing samples for digital soil mapping. *Geoderma*, *311*, 109–119. https://doi.org/10.1016/j.geoderma.2017.03.014

Arrouays, D., Grundy, M. G., Hartemink, A. E., Hempel, J. W., Heuvelink, G. B. M., Hong, S. Y., ... Zhang, G.-L. (2014). GlobalSoilMap: Toward a fine-resolution global grid of soil properties. *Advances in Agronomy*, *125*, 93–134.

Batjes, N. H., Ribeiro, E., Van Oostrum, A., Leenaars, J., Hengl, T., & Mendes de Jesus, J. (2017). WoSIS: Providing standardised soil profile data for the world. *Earth System Science Data*, *9*, 1–14. https://doi.org/10.5194/essd-9-1-2017

Beazley, D. (2010). *Understanding the Python GIL*. Retrieved from http://www.dabeaz.com/GIL/

Behrens, T., Schmidt, K., MacMillan, R. A., & Viscarra Rossel, R. A. (2018). Multi-scale digital soil mapping with deep learning. *Scientific Reports*, *8*, 15244. https://doi.org/10.1038/s41598-018-33516-6

Calaway, R., & Weston, S. (2017). *Package foreach*. Retrieved from http://cran.nexr.com/web/packages/foreach/index.html

Chaney, N. W., Minasny, B., Herman, J. D., Nauman, T. W., Brungard, C. W., Morgan, C. L. S., ... Yimam, Y. (2019). POLARIS soil properties: 30-m probabilistic maps of soil properties over the contiguous United States. *Water Resources Research*, *55*(4), 2916–2938. https://doi.org/10.1029/2018WR022797

Chaney, N. W., Wood, E. F., McBratney, A. B., Hempel, J. W., Nauman, T. W., Brungard, C. W., & Odgers, N. P. (2016). POLARIS: A 30-meter probabilistic soil series map of the contiguous United States. *Geoderma*, *274*, 54–67. https://doi.org/10.1016/j.geoderma.2016.03.025

Cheng, T. (2013). Accelerating universal Kriging interpolation algorithm using CUDA-enabled GPU. *Computers & Geosciences*, *54*, 178–183. https://doi.org/10.1016/j.cageo.2012.11.013

Grunwald, S. (2009). Multi-criteria characterization of recent digital soil mapping and modeling approaches. *Geoderma*, *152*, 195–207. https://doi.org/10.1016/j.geoderma.2009.06.003

Guan, Q., Shi, X., Huang, M., & Lai, C. (2016). A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures. *International Journal of Geographical Information Science*, *30*, 494–514. https://doi.org/10.1080/13658816.2015.1039538

Gutiérrez de Ravé, E., Jiménez-Hornero, F. J., Ariza-Villaverde, A. B., & Gómez-López, J. M. (2014). Using general-purpose computing on graphics processing units (GPGPU) to accelerate the ordinary kriging algorithm. *Computers & Geosciences*, *64*, 1–6. https://doi.org/10.1016/j.cageo.2013.11.004

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*, 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hengl, T., Mendes de Jesus, J., Heuvelink, G. B. M., Ruiperez Gonzalez, M., Kilibarda, M., Blagotić, A., … Kempen, B. (2017). SoilGrids250m: Global gridded soil information based on machine learning. *PLoS ONE*, *12*(2), e0169748. https://doi.org/10.1371/journal.pone.0169748

Jiang, J., Zhu, A-X., Qin, C.-Z., Zhu, T., Liu, J., Du, F., … An, Y. (2016). CyberSoLIM: A cyber platform for digital soil mapping. *Geoderma*, *263*, 234–243. https://doi.org/10.1016/j.geoderma.2015.04.018

Jones, E., Oliphant, T., & Peterson, P. (2001). *SciPy: Open source scientific tools for Python*. Retrieved from http://www.scipy.org/

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, *38*, 157–174. https://doi.org/10.1016/j.parco.2011.09.001

Li, D., Malyshev, S., & Sheviliakova, E. (2016). Mapping the global depth to bedrock for land surface modeling. *Journal of Advances in Modeling Earth Systems*, *8*, 917–935.

Li, Z., Fotheringham, A. S., Li, W., & Oshan, T. (2019). Fast Geographically Weighted Regression (FastGWR): A scalable algorithm to investigate spatial process heterogeneity in millions of observations. *International Journal of Geographical Information Science*, *31*, 155–175. https://doi.org/10.1080/13658816.2018.1521523

Liu, J., Zhu, A-X., Rossiter, D., Du, F., & Burt, J. (2020). A trustworthiness indicator to select sample points for the individual predictive soil mapping method (iPSM). *Geoderma*, *373*, 114440. https://doi.org/10.1016/j.geoderma.2020.114440

McBratney, A., Mendonça Santos, M., & Minasny, B. (2003). On digital soil mapping. *Geoderma*, *117*, 3–52. https://doi.org/10.1016/S0016-7061(03)00223-4

Minasny, B., & McBratney, A. B. (2016). Digital soil mapping: A brief history and some lessons. *Geoderma*, *264*, 301–311. https://doi.org/10.1016/j.geoderma.2015.07.017

Padarian, J., Minasny, B., & McBratney, A. B. (2015). Using Google's cloud-based platform for digital soil mapping. *Computers & Geosciences*, *83*, 80–88. https://doi.org/10.1016/j.cageo.2015.06.023

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, É. (2012). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

R Core Team. (2013). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing.

Ramcharan, A., Hengl, T., Nauman, T., Brungard, C., Waltman, S., Wills, S., & Thompson, J. (2018). Soil property and class maps of the conterminous United States at 100-meter spatial resolution. *Soil Science Society of America Journal*, *82*, 186–201. https://doi.org/10.2136/sssaj2017.04.0122

Shi, X., Lai, C., Huang, M., & You, H. (2014). Geocomputation over the emerging heterogeneous computing infrastructure. *Transactions in GIS*, *18*, 3–24. https://doi.org/10.1111/tgis.12108

Shi, X., & Ye, F. (2013). Kriging interpolation over heterogeneous computer architectures and systems. *GIScience & Remote Sensing*, *50*, 196–211. https://doi.org/10.1080/15481603.2013.793480

Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, *12*, 199–200. https://doi.org/10.1109/MCSE.2010.69

Tang, W., Feng, W., & Jia, M. (2015). Massively parallel spatial point pattern analysis: Ripley's K function accelerated using graphics processing units. *International Journal of Geographical Information Science*, *29*, 412–439. https://doi.org/10.1080/13658816.2014.976569

Van Rossum, G. (1995). *Python tutorial*. Amsterdam, the Netherlands: Centrum voor Wiskunde en Informatica.

Wadoux, A. M. J.-C., Padarian, J., & Minasny, B. (2018). Multi-source data integration for soil mapping using deep learning. *SOIL Discussions*, *2018*, 39.

Wang, H., Guan, X., & Wu, H. (2017). A hybrid parallel spatial interpolation algorithm for massive LiDAR point clouds on heterogeneous CPU-GPU systems. *ISPRS International Journal of Geo-Information*, *6*(11), 363. https://doi.org/10.3390/ijgi6110363

Yang, L., Qi, F., Zhu, A-X., Shi, J., & An, Y. (2016). Evaluation of integrative hierarchical stepwise sampling for digital soil mapping. *Soil Science Society of America Journal*, *80*, 637–651. https://doi.org/10.2136/sssaj2015.08.0285

Zeng, C., Yang, L., Zhu, A-X., Rossiter, D. G., Liu, J., ... Wang, D. (2016). Mapping soil organic matter concentration at different scales using a mixed geographically weighted regression method. *Geoderma*, *281*, 69–82. https://doi.org/10.1016/j.geoderma.2016.06.033

Zhang, G., Huang, Q., Zhu, A-X., & Keel, J. (2016). Enabling point pattern analysis on spatial big data using cloud computing: Optimizing and accelerating Ripley's *K* function. *International Journal of Geographical Information Science*, *30*, 2230–2252.

Zhang, G., & Zhu, A-X. (2019). A representativeness heuristic for mitigating spatial bias in existing soil samples for digital soil mapping. *Geoderma*, *351*, 130–143. https://doi.org/10.1016/j.geoderma.2019.05.024

Zhang, G., Zhu, A-X., & Huang, Q. (2017). A GPU-accelerated adaptive kernel density estimation approach for efficient point pattern analysis on spatial big data. *International Journal of Geographical Information Science*, *31*, 2068–2097. https://doi.org/10.1080/13658816.2017.1324975

Zhang, J. (2010). Towards personal high-performance geospatial computing (HPC-G): Perspectives and a case study. In *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, San Jose, CA (pp. 3–10). New York, NY: ACM.

Zhang, S.-J., Zhu, A-X., Liu, J., Yang, L., Qin, C.-Z., & An, Y.-M. (2016). An heuristic uncertainty directed field sampling design for digital soil mapping. *Geoderma*, *267*, 123–136. https://doi.org/10.1016/j.geoderma.2015.12.009

Zhu, A-X., Liu, J., Du, F., Zhang, S. J., Qin, C. Z., Burt, J., ... Scholten, T. (2015). Predictive soil mapping with limited sample data. *European Journal of Soil Science*, *66*, 535–547. https://doi.org/10.1111/ejss.12244

Zhu, A-X., Lu, G., Liu, J., Qin, C.-Z., & Zhou, C. (2018). Spatial prediction based on Third Law of Geography. *Annals of GIS*, *24*, 225–240. https://doi.org/10.1080/19475683.2018.1534890

Zhu, A-X., & Mackay, D. S. (2001). Effects of spatial detail of soil information on watershed modeling. *Journal of Hydrology*, *248*, 54–77. https://doi.org/10.1016/S0022-1694(01)00390-0

How to cite this article: Zhang G, Zhu A-X, Liu J, Guo S, Zhu Y. PyCLiPSM: Harnessing heterogeneous computing resources on CPUs and GPUs for accelerated digital soil mapping. *Transactions in GIS*. 2021;00:1–23. https://doi.org/10.1111/tgis.12730