Universiteit
Leiden
The Netherlands

Computer Science & Economics

Accelerating Value Propagation

Interpolation for Large Satellite Images

Dean van Laar

Supervisors:
Mitra Baratchi
Laurens Arp

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

June 28, 2024

Abstract

Value propagation-based spatial interpolation (VPint) is an interpolation algorithm created with Python for spatial missing data, with one of its applications being cloud removal from satellite images. However, compared to other methods, such as neural networks and moving average methods, VPint has a much longer running time. To illustrate, using a grid size of 40000, a moving average model has a running time of 100 seconds, while VPint has a running time of 1000 seconds. We propose two variations of the VPint algorithm to remedy the relatively long running time of VPint using parallel computing: a GPU accelerated method and a multiprocessing method. The GPU accelerated method uses the CuPy library and CUDA in order to exploit the computational power of the GPU to speed up the matrix computations of VPint. The multiprocessing method makes use of the structure of a satellite image and parallelises this structure into thirteen concurrent processes, as the satellite images used in this research contain thirteen sub-arrays that act as input of VPint. Our experimental results show a speedup of 1.65 using the GPU accelerated method compared to the original VPint algorithm, while the multiprocessing method shows a speedup of 5.58 compared to the original algorithm, thus concluding that both of our proposed methods are faster than the original VPint method, with the multiprocessing method achieving a higher speedup than the GPU accelerated method.

Acknowledgements

I would like to thank Laurens Arp for his guidance, explanations and feedback throughout this thesis. In addition to this, I would like to sincerely thank Laurens and Dr. Mitra Baratchi for supervising this thesis and helping guide me towards its completion. Finally, I want to express my gratitude towards my family and friends for not only their support, but their interest in this project as well.

Contents

1 Introduction **1**

2 Background **3**

3 Theoretical Framework **5**
3.1 CUDA and CUDA Architecture 5
3.2 GPU Computing 5
3.3 Amdahl's Law 6
3.4 Multiprocessing 7
3.5 Suitable acceleration methods for VPint 8
3.5.1 GPU Acceleration 8
3.5.2 Multiprocessing 9

4 Related Work **10**
4.1 GPU acceleration 10
4.2 Multiprocessing 11

5 Methods **12**
5.1 Baseline 12
5.2 GPU acceleration 12
5.3 Multiprocessing 13

6 Experiments **14**
6.1 Research questions 14
6.2 Dataset 15

7 Results **17**
7.1 Interpretation 18
7.1.1 Error metrics: MAE and MSE 18
7.1.2 Running times 20

8 Conclusions and Further Research **22**

9 Code Availability **23**

References **27**

Appendices **28**

A Appendix: Error Metrics per Scene **28**

B Appendix: Running Times per Scene **30**

C Appendix: VPint Patches and Cloudless Patches per Scene **32**

1 Introduction

Interpolation is a type of estimation or mathematical prediction that is applied to missing data points within a dataset. In spatial data, interpolation techniques can be used to fill in missing data due to sparsity or obstruction of measurements. Based on known data, missing values can be predicted to try and create an accurate reconstruction of measurements without missing data.

VPint, a value propagation-based spatial(-temporal) interpolation method proposed by Arp et al. (2022), addresses real-world problems concerning missing data. For example, cloud coverage in satellite images can be remedied using this method, as well as filling in the gaps in sparse weather predictions. VPint proposes two methods, both based on Markov reward processes (MRP): static-discount MRP (SD-MRP) and data-driven weight prediction MRP (WP-MRP). Compared to other common methods such as autoregressive moving average (ARMA) models and convolutional neural networks (CNNs), VPint, particularly WP-MRP, achieves better results with regard to structural similarity and mean absolute error [Arp et al., 2022b].

Cloud removal, which is the application of VPint in this research, is a relevant problem in satellite imagery, as on average, 55% of the land on Earth is covered by clouds at any given time [King et al., 2013]. Cloud coverage limits the use of the affected images and obstructs missions, including Copernicus' aims, to provide clean data (meaning free of noise or missing data) consistently [Ebel et al., 2021]. Spatial interpolation provides a solution for this problem, as well as other problems regarding meteorology, remote sensing and environmental science.

VPint is implemented with Python. Currently, VPint runs on a single thread on the Central Processing Unit (CPU), due to its use of Python packages that are limited to CPU usage. Furthermore, native Python code is restricted to the use of a single thread on the CPU. In the case of VPint, NumPy is the main library used for computations, which is not compatible with GPU computing, nor does it enable Python code to exploit several threads on a CPU. In other words, VPint lacks any parallel processing, which results in VPint having a much longer running time than regression methods [Arp et al., 2022b], such as moving average (MA) [Haining, 1978] and spatial autoregressive (SAR) models [Anselin, 2013].

To be more specific, considering clustered missing data which clouds in satellite images are an example of, it can be stated that both VPint methods, SD-MRP and WP-MRP, have a longer running time than earlier mentioned methods [Arp et al., 2022b]. VPint takes a chunk of an entire satellite image as input. The satellite images used in the experiments are all from the Sentinel-2 mission and can be found in the Copernicus Browser [Copernicus, 2024]. This chunk is called a patch, of which a visualisation can be found in Figure 1. These patches are also the form the input of VPint for the experiments.

Figure 1: A Python visualisation of a 256×256 patch from the urban Madrid satellite image from the Copernicus Browser [Copernicus, 2024]. The data is described in more detail in the experiments section.

In this research, a patch is 256×256 pixels with each pixel containing 13 bands. The various bands contain different information regarding the pixel, such as the visible colours, but also chance of cloud coverage. This patch structure results in a grid size far greater than 40000 and, as shown in the research by Arp et al. (2022), a grid with a size of 40000 leads to a running time of approximately 1000 seconds, as opposed to some competitors, such as the earlier mentioned MA [Haining, 1978] and SAR [Anselin, 2013] models, who have a running time of about 100 seconds using the same grid size. It is crucial to remedy this difference for VPint to gain a competitive advantage over other spatial(-temporal) interpolation methods.

This leads to the following research question:
What are the effects of different acceleration methods in value propagation-based spatial interpolation with regard to running time?

The main contributions in this work are two accelerated versions of the VPint algorithm used on spatially clustered real-world data, i.e., clouds in satellite images. Both proposed methods use a form of parallel computing to achieve its acceleration. These accelerated versions are a GPU accelerated method and a method using multiprocessing. We performed experiments on a dataset consisting of satellite images of 19 different regions, which resulted in a speedup of 1.65 and 5.58 respectively compared to the original VPint algorithm.

To introduce the structure of this paper and the content: This chapter contains the introduction; Section 2 introduces relevant background information; Section 3 includes the theoretical framework and definitions; Section 4 discusses related work; Section 5 contains a description of the methods; Section 6 describes the experiments; Section 7 contains the results of the experiments and its interpretation; Section 8 concludes.

2 Background

Optical remote sensing, which concerns visible and infrared light data that is typically measured by satellites, has many applications. For instance, object detection, which deals with identifying objects of interest in satellite images [Cheng and Han, 2016] and change detection, which aims at detecting change in use of land and land cover [Peng et al., 2021]. However, optical remote sensing is not without challenges, one of which is cloud coverage. In some cases, such as weather forecasting and water research, clouds are useful to keep [Li et al., 2019], however, in many other cases, such as object and change detection, cloud cover is considered as noise or missing data. Enhancing the effectiveness of applications of remote sensing that are affected by cloud coverage creates the need for cloud removal techniques. A common technique for handling cloud coverage is (spatial) interpolation of the missing pixels in satellite images as a result of clouds [Cheng et al., 2014].

VPint is an iterative interpolation method that utilises an estimation grid and that can be applied to the cloud removal problem [Arp et al., 2022b]. To summarise the VPint method, when an element is unknown (missing data), its value is updated to the estimated value, which is based on the estimated values of its neighbours. The estimated values of these neighbours are, in turn, based on the estimated values of their neighbours, thus propagating known values through these series of estimated values of neighbours. Updating the estimation grid is based on Markov reward processes, meaning that rewards are associated with updating an estimated value in the estimation grid, making certain changes more desirable than others. On top of that, future rewards are also incorporated in the update rule of the estimation grid, making the desirability of change in the estimation grid dependent on both the direct and indirect reward of that change. The iterative updating of the estimation grid continues until it reaches an equilibrium, meaning that the mean change in values in the estimation grid is smaller than pre-determined value. If this equilibrium is not reached, there is a limit to the number of iterations set by VPint. As mentioned in Section 1, VPint has two variants: SD-MRP and WP-MRP. WP-MRP, which is used in our experiments, allows for neighbour-specific weights, meaning that the effect one neighbour has on the predicted value of an element might differ from the effect that another neighbour has on the same element. A weight prediction model predicts the weight between a neighbour pair based on spatial data. On the other hand, SD-MRP does assume isotropy, meaning that all neighbours have the same weights and thus the same spatial effect on each other.

Some other methods in the field of cloud removal are deep learning approaches that make use of convolutional neural networks [Meraner et al., 2020, Ma et al., 2023] and regression methods, such as Gaussian process regression that automatically optimise hyperparameters [Park and Park, 2022] and spatially and temporally weighted regression [Chen et al., 2017].

The satellite images that are used in the experiments of this research are from the Copernicus Sentinel-2 mission, which is a unit of the European Union's space programme

regarding Earth observations [ESA, nd]. The images provided by Sentinel-2 consist of thirteen bands that contain a variety of information. For example three bands contain true colour data, meaning the colours how humans would see it [GISGeography, 2024]. Other bands contain short wave infrared measurements, while others measure visible and near infrared. All bands differ in the central wavelength of which the measurements are taken and their resolutions differ from 10 meters to 60 meters, which represents the distance between independent measurements The used data is from the bottom of the atmosphere, meaning it is Sentinel-2 L2A data, which is top of the atmosphere L1C data that has undergone atmospheric corrections [PlanetLabs, nd]. All these characteristics can differ in different satellites, for example, the Sentinel-3 satellite provides data of 21 bands, all with a resolution of 300 meters [Qian, 2015].

3 Theoretical Framework

In this section, we introduce relevant topics regarding our methods. This includes NVIDIA's CUDA, which we use to enable GPU computing in our experiments, after which we expand on the topic of GPU computing, Amdahl's law and multiprocessing, with a focus on the Python programming language.

3.1 CUDA and CUDA Architecture

Python code typically runs on the CPU and there is no option in the default Python installation that enables a piece of code to use the GPU instead of the CPU. For this purpose, CUDA, formerly an acronym for Compute Unified Device Architecture, however now just known as CUDA as its own definition, will be used [Tuomanen, 2018]. CUDA is a general purpose GPU processing framework developed by NVIDIA, a firm specialised in GPU production. CUDA enables code, including Python code, to run on the GPU [Holm et al., 2020]. CUDA assigns the GPU as the device, meaning that it executes (parallel) tasks, while assigning the CPU as the host, which manages general system operation.

Regarding the concept of threads (essentially, a thread is a sequence of instructions that is given to a GPU or CPU [Ramuglia, 2023]), CUDA arranges threads into thread blocks, that are executed on the same core of a GPU. In turn, thread blocks are arranged into grids that run on the GPU [Chen et al., 2023].

GPU computing requires CUDA availability on an NVIDIA graphics card in order to be possible. There are some alternatives to CUDA for GPU computing, for example, ROCm using an AMD graphics card [Shafie Khorassani et al., 2021] and OpenCL [Karimi et al., 2010], which can be used on NVIDIA GPUs. However, CUDA will be utilised in this research, since the GPU accelerated method of VPint runs on an NVIDIA graphics card in the experiments, and OpenCL has a minor performance disadvantage compared to CUDA [Su et al., 2012].

3.2 GPU Computing

GPU acceleration utilises the computing power of the GPU, which often contains thousands of cores [Buber and Diri, 2018]. More cores lead to more computations that can be done simultaneously. CPUs often only have four or eight cores, with one CPU core typically containing two threads and a thread being the virtual component that processes sequences of instructions. This means that a typical CPU can process eight to sixteen sequences of instructions (as it contains eight to sixteen threads). This leads to CPUs having considerably less availability for concurrent computations than GPUs. The result is the possibility of a significant volume of parallel processing using GPU compared to a CPU, which potentially increases the operating speed of a program when run on a GPU [Buber and Diri, 2018]. A comparison can be made with roads, where a GPU represents a wide road with many lanes designed for many slow moving cars, while a CPU repre-

sents a narrow road with one or a few lanes on which the cars drive fast [Tuomanen, 2018].

Another important difference between CPUs and GPUs is the focus of the architecture. A CPU focuses on minimising latency [Team, 2022], meaning that the time between an instruction and the finishing of the processing is diminished. On the other hand, a GPU prioritises a high throughput, which means that it focuses on being able to do more computations simultaneously, indicating a specialisation in parallel computing [Owens et al., 2008, Tuomanen, 2018]. Thus, CPUs (low throughput and low latency) and GPUs (high throughput and high latency) contrast each other in the latency-throughput trade-off, which has always been a fundamental consideration in computer science [Subhlok and Vondran, 1996].

The advantage of parallel processing is only achieved when processing a high number of independent computations, which is why it is typically applied in 3D visuals, video editing and in the scientific field, as these applications can contain a lot of data and/or processes that can be computed independently [Grochowski et al., 2004]. While scalar processes (processes with low intrinsic parallelism), such as I/O bound processes [Engbrecht, 2008] and stack based breadth-first search algorithms [Greenlaw, 1992], customarily use the CPU as it is often as efficient as (if not more efficient than) the GPU for these tasks, utilising the GPU can enhance performance significantly for larger sets of tasks, such as matrix computations [Chrzeszczyk and Chrzeszczyk, 2013] and video-editing (which requires large-scale rendering) [Sharma, 2023]. This does require the tasks or computations to be independent, as is the case in matrix computations, where partial results can be calculated separately from the rest of a given matrix calculation without interfering in or needing the results of other partial computations.

In short, GPU computing is characterised by three aspects [Owens et al., 2008]:

1. Extensive computational requirements.
2. A high level of parallelism.
3. Throughput is prioritised over latency.

3.3 Amdahl's Law

An estimation can be made of the speedup achieved with GPU or parallel computing using *Amdahl's law* [Tuomanen, 2018]:

$$speedup = \frac{1}{(1-p) + \frac{p}{N}} \tag{1}$$

This equation gives a very rough estimation of how much parallelisation would speed up an algorithm based on the proportion of the running time or code that benefits from the parallelisation (p) and the number of cores (N) [Tuomanen, 2018]. N can also be defined as the speedup of the part of the task that can be parallelised [Bryant and O'Hallaron, 2015]. The denominator contains $1 - p$ due to the speedup

being dependent on how much of the algorithm can be sped up. It could be the case that certain operations cannot be done in parallel but should stay serial in order for the algorithm to function properly, meaning that it cannot be sped up with GPU acceleration. Parallel computing requires the operations that run in parallel to be independent from each other and thus be computable without the (partial) result or interference of another parallel computation. The maximum theoretical speedup is therefore:

$$maximum\ theoretical\ speedup = \frac{1}{1-p} \tag{2}$$

However, achieving the maximum theoretical speedup would mean that the parallelised portion of the code is finished instantly. As this is not the case, $\frac{p}{N}$ is added to $1-p$ in the denominator. This signifies how much the parallelisable portion benefits from the parallelisation. This is comparable to constructing a desk in two hours with one individual, or in one hour with two individuals.

There are some limitations regarding Amdahl's law. Firstly, it ignores the parallelisation overhead that occurs in parallelised tasks. There are several types of overhead, such as memory and instruction latency, which relate to the initialisation and allocation of resources for the parallel computing task [Bhattacharjee et al., 2011]. Furthermore, the proportion of code that benefits from parallelisation is an approximation of the proportion of the running time that benefits from parallelisation. Looking at the proportion of the code, it depends on coding style and manner of implementation, which does not necessarily translate directly to the proportion of running time. Alternatively, looking at the proportion of the running time, Amdahl's law can be calculated accurately, but this requires the running time using parallel computing to be known already, rendering the equation useless. These factors stress the fact that Amdahl's law is a rough estimation and not an exact calculation.

3.4 Multiprocessing

Another method of speeding up a program is multiprocessing. This method enables multiple tasks to be executed concurrently on different threads of the CPU [Aziz et al., 2021]. Alternatively, multiprocessing enables a program to run on several CPUs simultaneously. Like GPU acceleration, multiprocessing requires the parallel tasks to be independent from each other. On top of that, tasks can finish in a different order than when the code is run in series, meaning that the result of the multiprocessing task has to be properly rearranged to get the desired output. In Python, multiprocessing is enabled by the multiprocessing Python library and the multiprocessing method is applied to one CPU, using multiple threads. An advantage of this method is that, like GPU acceleration, it enhances throughput [Aziz et al., 2021], partially alleviating the CPU's throughput restriction. Especially in Python, which has a Global Interpreter Lock (GIL) that enables only one thread to be in a state of execution concurrently, because of which CPU usage is limited as well [Ajitsaria, 2018]. The throughput is increased in a multiprocessing method by utilising unused CPU cores for computations.

3.5 Suitable acceleration methods for VPint

After discussing GPU computing and multiprocessing in general, we will now elaborate on why these parallelisation methods are suitable for VPint.

3.5.1 GPU Acceleration

VPint is a suitable candidate for GPU acceleration as it matches the three characteristics mentioned by Owens et al. [Owens et al., 2008]:

1. VPint can have a large computational requirement, due to its application of interpolation in cloudy satellite images. As mentioned before, a patch of a satellite image from the Sentinel-2 satellites contains 256 by 256 pixels, each pixel containing 13 bands that determine the characteristics of the pixel such as the colour and cloud coverage. An example of a satellite image is the Madrid urban area image found in the Copernicus Browser [Copernicus, 2024] taken with the Sentinel-2 satellite (details of the data can be found in Section 6). The entire image consists of 441 patches, meaning that the matrix operations on the 256×256 matrices have to be carried out 441 times, which could be defined as a large computational requirement.

2. Parallelism concerning GPU acceleration is achieved in VPint in matrix calculations. Using a GPU, especially large matrix calculations can benefit, as the many cores of a GPU can independently calculate partial results.

3. Latency is not as important for VPint as the throughput: the delay between the instruction and the computation is marginalised when compared to the size of the computation that needs to be done. It is more important that more computations can run in parallel, than minimising the time of a computation.

As parallelism is achieved in the matrix calculations, these are the lines of code that benefit from GPU acceleration. Lines such as creating variables, matrices and function calls are not parallelisable, meaning that a substantial part of VPint does not benefit from the GPU acceleration. Luckily, the main loop, which runs approximately 180 times more often than any other part of VPint under the conditions in the experiments, contains 10 out of 27 lines of code that benefit from the acceleration. Using a graphics card with 3584 cores, which is the number of cores that the GPU in the experiments has [NVIDIA, 2021], Amdahl's law predicts the following potential speedup:

$$speedup = \frac{1}{(1 - (10/27) + \frac{10/27}{3584}} = 1.588 \tag{3}$$

When incorporating the entire VPint code, this speedup decreases, as other parts of the code contain relatively fewer lines that benefit from GPU acceleration. The speedup of the entirety of VPint is expected to be as follows:

$$speedup = \frac{1}{(1 - (621940.5/1694300.5) + \frac{621940.5/1694300.5}{3584}} = 1.564 \tag{4}$$

Due to conditional statements and the number of occurrences of each path created by these statements, there is a .5 used in Equation 4. Overall, the expected speedup of VPint using GPU acceleration is 1.564.

3.5.2 Multiprocessing

Unlike GPU acceleration, in which the parallelisation lies within in matrix multiplication, using multiprocessing, the parallelisation lies within the bands of the satellite image. To enable multiprocessing, the code of the pre-processing of the data for VPint is adjusted in such a way that the 13 bands of the pixels are parallelisable. Then, the entire code of the VPint benefits from this. Only the pre-processing is still serialised, as the bands are still one matrix before the initiation of VPint. When considering only VPint, the entirety of the code is parallelised, meaning that p = 1. The computations are parallelised over the 13 bands, meaning that $N = 13$. Using Amdahl's law, this leads to the following estimated speedup:

$$speedup = \frac{1}{(1-1) + \frac{1}{13}} = 13 \tag{5}$$

By rough estimation, the speedup of VPint when the bands are parallelised would thus be 13, given there are enough cores to run the 13 bands in parallel.

4 Related Work

GPU computation and multiprocessing are two parallel computing methods that have been extensively explored already. Since VPint is implemented using Python, this section will focus on related work regarding Python implementations of GPU acceleration and multiprocessing.

4.1 GPU acceleration

As CuPy is the Python library that is used in the GPU accelerated version of VPint, this section will focus mainly on CuPy. The reason for using CuPy is expanded in the Section 5.
Research by Ninshino and Loomis (2017) [Nishino and Loomis, 2017] has shown a simple comparison between Numpy running on the CPU and CuPy running on the GPU. Ninshino and Loomis (2017) include basic array computations in their experiments, applying the arange, reshape, transpose and multiplication operations to a given array. The experimental results show longer computation times using CuPy for arrays with sizes 10^4 and 10^5, while from arrays with size 10^6 and onward, the computation time for the CuPy method is faster, indicating that computations with a large computational requirement benefit from using a GPU. Regarding our research, this might indicate a minimal speedup, as VPint contains matrices with an approximate size of $6 \cdot 10^5$.

CuPy has been applied to Computational Fluid Dynamics, largely depending on matrix computations [Chen et al., 2023]. The experiments in the research by Chen et al. (2023) were run using an NVIDIA Geforce RTX 2060 GPU and an Intel Core i5-9500 CPU. Some experiments resulted in a speedup by a factor of 110 when using CuPy on the GPU instead of NumPy on the CPU. This speedup was accomplished using larger two-dimensional square matrices. A matrix with shape 8192×8192 achieved a speedup of approximately 76, while a matrix with shape 256×256 achieved a speedup of about 1.7. The matrix computations of VPint are on matrices with shape 256×256 as well, thus Chen et al. (2023) provide an indication of the potential speedup of VPint based on parallel matrix computations with this particular shape.

Research by Balogh and Ruiz (2022) focuses on GPU accelerating the construction of Hadamard Matrices [Balogh and Ruiz, 2022]. CuPy is used as Python library to enable GPU acceleration. The experiments performed by Balogh and Ruiz (2022) include creating and mutating Hadamard matrices, which are square matrices consisting of 1 and -1 entries. CuPy allowed the researchers to run thousands of matrix computations, as well as populations consisting of thousands of matrices simultaneously. The speedup of their entire experiment is not discussed, but it is stated that CuPy enabled the handling of thousands of matrices and computations in parallel, which may imply that GPU accelerated matrix computations can result in a speedup in VPint using CuPy.

Research considering the effect of variables on GPU coding, including different im-

plementation packages, applies the GPU acceleration packages Numba and CuPy to a method that solves the Burgers' equation with the finite difference method (FDM) [Xu et al., 2022]. FDM is a method utilised in solving complex differential equations [Shamey and Zhao, 2014]. Xu et al. (2022) [Xu et al., 2022] have found the same error produced by the different implementations using Numba, Cupy and NumPy, with the global relative error and norm error as the error metrics. On top of that, Xu et al. (2022) have measured an improvement concerning operation speed using either Numba or Cupy when compared to the NumPy implementation. Xu et al. (2022) indicate that applying GPU computing to their problem does not result in a change in the error metrics, which is expected in our experiments as well.

4.2 Multiprocessing

Multiprocessing in Python has been applied to the Monte Carlo N-Particle (MCNP) code [Sazali et al., 2022]. MCNP is a code used for general purpose many particle transport simulations [Briesmeister et al., 2000] and is computationally heavy due to its tracking of all particles during the simulation. The experiment of Sazali et al. (2022) [Sazali et al., 2022] consisted of running an MCNP simulation on the three hardware setups and checking the running time on the setups using different numbers of logical cores. Their experiment did not result in significant changes in the average error of the experiment, while they did find a significant improvement with regard to running time. The experiments carried out by Sazali et al. (2022) produced an average speedup of approximately 7.35 when utilising twelve cores instead of one. Relating to our research, this not only indicates a significant speedup by using multiprocessing, but it indicates that the speedup is not necessarily directly proportional to the number of cores that are used.

Considering geospatial data, multiprocessing has been applied to the max-p-regions problem [Sindhu, 2018]. The max-p-regions problem concerns the clustering of geographical regions under certain constraints in order to create homogeneous areas with regard to specific characteristics [Duque et al., 2012]. Viney Sindhu (2018) [Sindhu, 2018] applied the multiprocessing Python library using a hardware setup with 16 cores. Depending on the input of the multiprocessing, the experiments resulted in a speedup of the entire problem between 3.31 and 13.94, where larger problems attained a larger speedup. Sindhu (2018) shows speedups for multiprocessing applied to geospatial data, and similarly to the research of Sindhu, our experiments concern geospatial data.

5 Methods

To understand the GPU accelerated method and the multiprocessing method, certain aspects of the original VPint algorithm need to be understood. Firstly, the input of VPint consists of two collocated patches with a resolution of 256×256 pixels. This means that only one of the thirteen bands is used as the input of VPint at a time, meaning that the bands are serialised in the original VPint algorithm. When running VPint, many matrix calculations are carried out and many matrices are created. On top of that, VPint is optionally recursive, resulting in the numbers of matrix computations and matrix creations being even larger. In short, the original VPint algorithm relies on a serial calculation of the bands that comprise a satellite image, which are calculated (partially) by use of matrix computations. All methods use the VPint algorithm with an optional random component in its calculation of the predicted image enabled, meaning that different results can be obtained while using the same algorithm and data. More details regarding the VPint algorithm can be found in the original paper by Arp et al. (2022) [Arp et al., 2022b].

5.1 Baseline

The baseline is the original VPint code created by Arp et al. (2022) [Arp et al., 2022a], which we will call the original VPint method from now on. This method only exploits the computational power of one CPU thread, which means that the method runs entirely on the CPU. This means that the bands are run in a serialised manner. Matrix computations rely on NumPy and Python's native functions. The datatype of matrices in the original VPint method (including the input and output) is a NumPy array. The datatype of the items in the matrices are floats, which is a double precision datatype and corresponds to NumPy's float64 datatype.

5.2 GPU acceleration

The GPU accelerated method utilises the GPU to accelerate parallelisable functions and methods within the VPint algorithm. This concerns matrix computations, of which partial results can be calculated independently from the other partial calculations. On top of that, the matrices in the experiments typically have a shape of 256x256, thus, a total of 65536 operations per matrix computation can be done using thousands of GPU cores, instead of just one CPU core. GPU acceleration is enabled by the Python library CuPy. CuPy is not only used due to the fact that it enables GPU computing, but also due to its high compatibility with NumPy, meaning that many functions are identical [Chen et al., 2023].

The bands are serialised in this method. The datatype of the input matrices is a CuPy array, while the output is a NumPy array. This is done to have an identical output as the other methods with regard to datatype. The matrices that are initialised during VPint are CuPy arrays, while the datatype of the items in these arrays, as well as the

input and output arrays, are floats. This method relies on both GPU and CPU, with the matrix computations being executed on the GPU, while other tasks run on the CPU. The pre-processing is the same as the original VPint method, apart from the fact that the input of VPint is converted to a CuPy array. The VPint algorithm has been adjusted to contain CuPy functions and arrays instead of NumPy functions and arrays, although some NumPy functions have not been replaced by CuPy functions, which is due to minor incompatibilities within the VPint code. An example of this is NumPy's product function, which resulted in complications with other functions when changed to CuPy's product function. The affected functions are a minor part of the algorithm and are not relevant in enabling GPU computation for all matrix computations.

5.3 Multiprocessing

Multiprocessing makes use of the same VPint algorithm as the original VPint method, meaning that NumPy functions and arrays are used, resulting in this method running entirely on the CPU. This means that no conversion of the variables from CPU to GPU takes place, which saves time. However, the manner of input is adjusted compared to the original VPint method. Instead of calculating one of the thirteen bands of the satellite images at a time, all thirteen bands are calculated simultaneously. Using multiprocessing, the bands are all separate "jobs" that need to be carried out. These jobs are initialised in a for-loop, while assigning a different thread to each band, meaning that the bands are calculated in parallel.

The datatype of the matrices in this method, including the input, is a NumPy array and the datatype of the items in the matrices are floats. The matrix computations are done with NumPy functions. As the bands are calculated separately and simultaneously, the jobs of the bands finish in a different order than the original matrix. To handle this, the output of the jobs are saved in a dictionary with their corresponding original band index as the key. This dictionary is sorted after all jobs are finished and a NumPy array is created from this sorted dictionary in order to get an output which is similar to the other methods.

In short, the pre-processing is adjusted to handle the thirteen bands concurrently, while the VPint algorithm itself is the same as the one used for the original VPint method. The multiprocessing method utilises thirteen threads on the CPU.

6 Experiments

In this section, specifics regarding the experiments are discussed. This includes the hardware setup, the research questions that we try to answer and a detailed description of the dataset.

All methods were run on the same hardware setup:

- CPU: Intel(R) Xeon(R) CPU E5-2683 v4, 16 cores, 32 threads.
- GPU: NVIDIA Corporation GP102 [GeForce GTX 1080 Ti], 3584 CUDA cores.
- All experiments used 8GB of memory, whether this was 8GB on the CPU or GPU depends on the experiment.

The GPU method uses the CUDA toolkit 10.0, which was released in September 2018 [Yoshida et al., 2024]. The parameters of the VPint algorithm were kept constant for all methods. The code for running the methods can be found in Section 9.

6.1 Research questions

The main research question is as follows:
What are the effects of different acceleration methods in value propagation-based spatial interpolation with regard to running time?

The experiments will answer the following sub questions that will constitute an answer to the main research question:

1. Is there a significant difference between the methods regarding the mean absolute error (MAE) and mean squared error (MSE) using a two-tailed Wilcoxon signed-rank test [Wilcoxon, 1992]?
2. How do the average running times of the original VPint method, the GPU accelerated method and the multiprocessing method compare to each other?
3. How do the average running times of entire patches of the original VPint method, the GPU accelerated method and the multiprocessing method, including pre-processing and processing of the output, compare to each other?

The first sub question will give insight into whether the GPU accelerated and multiprocessing methods operate in the same manner as the original VPint method. In other words, is there a difference in the used VPint algorithms or are they equivalent for all methods with regard to the quality of the predicted image, based on the MAE and MSE? As the methods should be identical, there should not be a significant difference in the error metrics between any of the methods. The second sub question will clarify which methods are better than others in terms of running time of VPint only, excluding pre-processing and processing of the output. The last sub question will help understand whether the pre-processing and/or the processing of the output (such as the dictionary

sorting for the multiprocessing method) has a significant impact on the overall running time of a patch when using a certain method. More specifically, it gives insight into whether a given method that is significantly better than another method when looking at the second sub question, is significantly better when the pre- and post-processing of both methods are included in the running time. This sub question regards the average running times of entire patches of the three methods instead of solely the average running time of VPint.

6.2 Dataset

As mentioned in Section 2, the satellite images that are used in the experiment are Copernicus Sentinel-2 images. More specifically, bottom of the atmosphere data, called L2A data from Sentinel-2, is used. A satellite image from Sentinel-2 consists of thirteen bands, each containing data regarding a different central wavelength. Nineteen scenes, a scene being a region such as "Europe Cropland Ukraine", are used in total, the names of which can be found Appendix C, under the column "Scene name". All satellite images can be downloaded from the Copernicus Browser. [Copernicus, 2024], although the data can be downloaded from other sources as well (such as the Copernicus Space Component Data Access) [Copernicus, nd].

To execute VPint on one scene, three different satellite images of the same region are needed. Firstly, the target image, which is the true image that the predicted image (the output of VPint) should ideally resemble. Next, the feature image is needed, which is a satellite image of the same region but the feature image was taken before the target image. The used feature image is typically an image of one month before the target image, however, depending on alignment issues, this could be six months as well. A difference of one month is used since it closely, but not completely resembles the target image. An alternative would be a one-week difference, however, the image taken one week before the target is used as the target image when the original target image is misaligned with the other images. On top of that, a one-week difference between the feature and target image might not pose a challenge for VPint. Using a time difference of one month gives a balanced trade-off between the resemblance of the feature and target image and the difficulty of the task. Lastly, a mask image is used to determine which pixels are clouds in a satellite image. An extra band that is included in the entire dataset, but not included in the thirteen bands of the input images of VPint, contains the probability of a pixel being a cloud. When that probability for a pixel is higher than a pre-defined threshold, that pixel will become a cloud (missing data) in the cloudy image.

The cloudy image is initially a copy of the target image, but clouds are placed over this copy, after which this cloudy image becomes one of the satellite image inputs of VPint. The pixels that are categorised as clouds in the mask image will also be the pixels (by coordinate) that become clouds in the copy of the target image. In this cloudy image, a pixel is a cloud when the values over all bands are set to Not a Number (NaN) values, that signify missing data. Typically, a higher threshold for the cloud probabilities results in fewer pixels being categorised as clouds. The second image input of VPint is

the feature image, most often taken one month before the target (and thus the cloudy) image. This feature image is used to predict the covered values in the cloudy image.

The target, mask and feature images of each scene are used. One satellite image can be divided into 441 patches of 256x256 pixels, which applies to all scenes. It is important to mention that VPint does not necessarily run on all patches of a satellite image, as patches that do not have any clouds are skipped, as there is nothing to predict in these patches. Appendix C shows for all 19 scenes used for the experiments how many cloudy patches it contains and thus how many patches were skipped.

7 Results

In this section, we show the experimental results and discuss them afterwards. Firstly, the used significance test will be discussed, after which the error metrics, being the mean absolute error and mean squared error, of the original VPint method, GPU accelerated method and the multiprocessing method will be analysed. Finally, we will reflect on the differences of running times of the three methods.

All metrics are *averages* of the patches over all scenes. We analysed the original VPint method, the GPU accelerated method and the multiprocessing method by looking for a significant different (in the case of the error metrics) or better (in the case on running times) result achieved by the different methods. This significance is tested using a Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$. This test is used because both the errors (MAE and MSE) and running times of VPint are not normally distributed, which resulted from a normality test based on D'Agostino and Pearson's test [D'Agostino, 1971, D'Agostino and Pearson, 1973]. As we are testing a significant difference between the method regarding the error metrics, the MAE and MSE will be tested using a two-tailed Wilcoxon signed-rank test, while we are testing the running time on an improvement, meaning that a one-tailed Wilcoxon signed-rank test will be used. The same patches with extreme outliers in the error metrics have outliers in their running time for VPint, although less extreme. The method that outperforms most other methods in a particular metric has that metric printed in bold, except when none of the methods outperform each other, in which case none of the metrics are bold printed.

Table 1: The average values of the error metrics over all scenes per method are shown below. A bold printed value indicates the method that performed significantly better than most other methods, tested with a two-tailed Wilcoxon signed-rank test. There are no bold printed values in this table, meaning that there was not significant difference between any of the methods regarding the MAE and MSE.

Method	MAE	MSE
original VPint method	5.50E+17 ± 3.87e+19	5.08E+43 ± 3.74E+45
GPU acceleration	3.42E+15 ± 2.13e+17	5.03E+40 ± 2.98E+42
Multiprocessing	2.35E+16 ± 1.24e+18	5.25E+39 ± 3.74E+41

Table 2: The running times in seconds of VPint and of an entire patch over all scenes per method are shown below. A bold printed value indicates the method that performed significantly better than most other methods, tested with a one-tailed Wilcoxon signed-rank test.

Method	Running time of VPint	Running time of a patch
original VPint method	217.69 ± 88.42	143.78 ± 88.52
GPU acceleration	132.21 ± 54.52	89.44 ± 54.64
Multiprocessing	**39.02 ± 1.48**	**26.41 ± 1.55**

7.1 Interpretation

Before answering the sub questions, the large MAE and MSE in Table 1 of all methods must be discussed. These can be explained by looking at the individual scenes, of which the results regarding error metrics can be found in Appendix A. Most scenes do not have errors as large as in Table 1. However, unusually large errors can be found for all three methods in the "Europe Cropland Ukraine" and "Africa Cropland Nile" scenes. The former scene has an average MAE of more than 10^{16} and an average MSE of more than 10^{40} for all methods and the latter has average MAEs ranging from 10^5 to $1.37 \cdot 10^{17}$ and MSEs ranging between orders of magnitude of 10^{17} and 10^{41}. This large error is caused due to the absence of a maximum value that a predicted value can be, which we shall refer to as a clip, in the VPint functions and pre-processing. This clip limits values in an initialised array in the VPint algorithm to a certain value, which in turn limits the MAE and MSE to a certain value. After running VPint with a clip of 10000 on all of the patches that had an MAE of over 10000 during the experiments, these patches all returned an MAE in the order of magnitude of 10^2 and an MSE in the order of magnitude of 10^6 for all methods. On top of that, these unusually large errors were rare incidents that did not occur in the majority of patches. However, the averages of the error metrics of the afflicted scenes were affected by the small number of patches with such large errors.

As these large errors were a known complication in the research of Arp et al. (2022) when clipping was disabled, and due to the fact that the clip remedied these large errors, it can be concluded that the cause of the errors was not a fault in the experiment, but rather a result of the omission of a clip.

Again, it is important to note that the large errors were a rare occurrence and most of the scenes resulted in an MAE in the order of 10^2 and an MSE between the orders of 10^4 and 10^{12}. A complete overview of the error metrics per scene can be found in Appendix A, where it can be seen that most scenes do not have unusually large errors.

7.1.1 Error metrics: MAE and MSE

Reflecting on the first sub question: Is a significant difference between the methods regarding the MAE and MSE? The results in Table 1 do not show a significant difference

between any of the methods regarding the MAE and MSE. Due to the random element of VPint being enabled in our experiments for all methods, different results occur regularly, but when looking at the results over all nineteen scenes together, there is no significant difference between the methods for the error metrics. Although not tested, disabling this random element should result in all methods having the same error metrics.

When looking at the error metrics of the scenes independently, of which the entire tables with results can be found in Appendix A, there are several scenes that show a significant difference in MAE and MSE between the methods. At first glance, this result is especially strange when comparing the original VPint method and the multiprocessing method, as they use exactly the same VPint algorithm and metric calculation. When looking into the Wilcoxon signed-rank test, this can be explained by the number of ties that occur. Taking the MAE of the original VPint method and multiprocessing method as an example, the Wilcoxon signed-rank test calculates the difference between the MAE of a certain patch of the original VPint method and the MAE of the same patch of the multiprocessing method. All the absolute differences of all patches are ranked from small to large. Once ranked, they get an increasing weight and a sign depending on the sign of the difference. For example, the smallest few differences could be: $-0.01, 0.02, 0.025, -0.03, ...$
This would lead to their signed ranks respectively being: $-1, 2, 3, -4, ...$
However, this does not take into account the patches where the difference is zero, meaning that the methods have the same error with the used precision. This is where the problem lies, as there are many such cases, called ties, in all scenes. In some cases, these ties lead to a large imbalance in the signed ranks, which, in turn, results in a significant difference when no significant difference is expected.

To partially remedy this, the precision of the error calculation can be increased. Normally, the precision of the MAE and MSE is equal to the precision of a float, or double precision, with about 15 decimals. When setting the precision of the error calculations to 25 decimals, the number of ties decreases, as fewer patches have equal values for both methods when increasing the precision. The "America Forest Mississippi" scene (which contains 170 patches on which VPint is run) has been tested with a comparison between using double precision (approximately 15 decimals) and using 25 decimals when using the original VPint method and the multiprocessing method. This resulted in the number of patches that resulted in ties decreasing from 67 to 42 out of a total of 170 patches when using a 25-decimal precision. Although the p-value was still significant due to the large remaining number of ties, this result does indicate the disturbance of ties to the significance test.

Supporting the argument that all three methods do not differ significantly regarding their MAE and MSE, is the fact that performing a Wilcoxon signed-rank test on the entire dataset, i.e. all scenes together, did not result in a significant difference between any two methods. All scenes of the original VPint method compared to all scenes of the multiprocessing method concerning MAE resulted in a p-value of 0.225, which is higher than the test statistic $\alpha = 0.05$. This means that the null hypothesis of the Wilcoxon signed-rank test, which is that the result of both groups (methods) come from the same

distribution, cannot be rejected.

In short, based on the facts that:

- The ties disturb the significance test for some scenes when the multiprocessing method is compared to the original VPint method or the GPU accelerated method concerning the MAE and MSE,
- When considering all scenes, the null hypothesis of the Wilcoxon signed-rank test cannot be rejected when comparing any of the methods based on their MAE and MSE, thus indicating no significant difference between the methods over all scenes,
- The GPU method did not yield any significant difference compared to the original VPint method regarding MAE and MSE,

It can safely be assumed that the three methods only differ significantly regarding the error metrics in specific cases due to limitations of the Wilcoxon signed-rank test and that the methods should not differ regarding the quality of their predictions.

7.1.2 Running times

Table 2 shows a significant difference in running time between all methods, with the multiprocessing method having a significantly shorter running time for all scenes than both the original VPint method and the GPU accelerated method. Furthermore, the GPU method has a significantly shorter running time than the original VPint method. Using the average running times from the same table, it can be established that the multiprocessing method leads to a speedup of VPint of approximately 5.58 when compared to the original VPint method. This is quite a bit lower than the expected speedup of 13, as was expected in Equation 5. As discussed in Section 3, the actual speedup is lower than the expected speedup partially due to the parallelisation overheads, such as the initialisation and termination overhead. On top of that, software overhead concerning the used libraries, the operating system and the parallel compiler [Balasooriya, 2017] may have an impact on the lower speedup as well. These last types of overhead include function calls, resource management and code generation. Hardware limitations may also play a role in the lesser actual speedup, as only 8GB of the total 128GB of available memory on the CPU was used for the multiprocessing experiment (as well as for the original VPint method). Only a very small portion of this lower speedup is due to having to sort the output dictionary and create a NumPy array in the multiprocessing method. On average, these extra operations only took approximately 0.03 seconds per patch.

The initialisation overhead also plays a role in the running time of entire patches. On average, for the multiprocessing method, each patch added about 1.92 seconds to the running time concerning pre- and post-processing only. For the original VPint method, this average was 1.69 seconds, meaning that the pre- and post-processing running time has increased by 0.23 seconds per patch using multiprocessing compared to the original VPint method. Thus, when comparing the average running time for entire patches, the

multiprocessing method leads to a speedup of 5.44.

Furthermore, a patch running VPint using the multiprocessing method is only as fast as its slowest band. While the calculation in Equation 5 assumes all bands take the same amount of time, this is not true. Some bands take longer to converge to the equilibrium in the mean change of the prediction grid than others. Some bands converge after a few iterations, while other bands might not converge at all (which can be seen in the case of the unusually large errors). For example, the tenth band in the L2A products only consists of zeroes as this band does not contain any bottom of the atmosphere information [Müller-Wilm et al., 2017]. A band of zeroes is calculated much faster than the band that caused the very large errors in the "Europe Cropland Ukraine" scene, as this band reaches the limit of computations set by VPint, while the band consisting of zeroes does reach the equilibrium that VPint converges to. An exaggerated illustration can be given with a case where one band takes 100 seconds to finish while the other 12 bands only take 10 seconds. The original VPint method, which calculates the bands in series, would take 220 seconds to finish all bands. The multiprocessing method, which calculates all bands in parallel, would take 100 seconds to finish all bands. Equation 5 would assume that all bands take 100 seconds, meaning 1300 seconds for the original VPint method and 100 seconds for the multiprocessing method, which is not the case. This is more nuanced in reality, but a similar situation in the experiments partially explains why the actual speedup is less than the expected speedup.

The GPU accelerated method achieves a speedup of about 1.65 compared to the original VPint method. This is quite close to the predicted speedup in Equation 4, which was 1.564. The slight difference between the expected speedup and the actual speedup might be caused by a larger proportion of the running time benefiting from GPU computing than the proportion of the code used in Equation 4 approximates. Using the proportion of code in Amdahl's law assumes that all lines of code have an equal individual running time. This is not the case, as, for example, executing a large matrix computation probably has a longer running time than creating an integer variable. Thus, the actual speedup suggests that the proportion of running time that benefits from parallel computing is larger than the proportion of running time that is estimated by the proportion of code in Equation 4. The GPU accelerated method resulted in a speedup of 1.61 compared to the original VPint method when looking at the average running time of a patch. This is slightly lower than the speedup for only the VPint algorithm because of the cloudless patches having a larger impact on the average patch running time of a higher running time, as these cloudless patches finish almost instantly.

To put the speedups of the VPint algorithm into perspective, the average total running time of VPint of all patches of one scene using the original VPint method is approximately 63000 seconds or 17.5 hours. The GPU accelerated method had an average total running time of VPint of about 39000 seconds, which is about 10.8 hours. The multiprocessing method resulted in an average total running time of 11000 seconds, or 3.1 hours.

8 Conclusions and Further Research

To conclude, both of the proposed alternative methods for VPint, GPU acceleration and multiprocessing, resulted in a significantly faster running time than the original VPint method: a speedup of 1.65 and 5.58 respectively. The multiprocessing method only differs significantly in MAE and MSE from the original VPint method and GPU accelerated method in specific cases, which is due to a limitation of the used significance test. It can be assumed that the proposed methods result in similar quality predictions as the original VPint method. The GPU accelerated method leverages the computational power of the GPU to speed up the matrix computations carried out by VPint, while the multiprocessing method exploits the data structure of the satellite images in order to reach its speedup. Overall, the multiprocessing method is found to be much faster than the original VPint method and the GPU accelerated method, leading to its recommended use for computations with VPint.

Future research might include changing the datatype of the values inside the (input) arrays of VPint. Research by Xu et al. (2022) has shown that changing the datatype from double precision (float64) to single precision (float32) reduces running time by a minimum of 20% in their experiments regarding finite difference methods [Xu et al., 2022]. Although their experiments are not related to VPint, their work does suggest a substantial decrease in running time when using a single precision datatype. Researching the exact impact that different datatypes, such as float16, float32, float64 and float128, have on the running time and on the error metrics might yield interesting results. Looking into the exact effect of these datatypes on the error metrics might be especially interesting due to the occurrence of ties in the Wilcoxon signed-rank test. The precision/running time trade-off for VPint might prove an interesting and useful research.

Furthermore, we propose two alternative methods to the original VPint code: the GPU accelerated method and the multiprocessing method. Both of these methods resulted in a significant decrease in running time compared to the original VPint method. Combining these metrics might result in an even better realised speedup and thus would be worth looking into in the future. We did not include this method in this research due to a limitation of CUDA, which does not allow processes to be forked after initialisation of CUDA. That means that this method would require a different manner of input regarding multiprocessing to avoid the CUDA limitation.

Lastly, changing the input size of the matrices of VPint could yield interesting speedups, especially for the GPU accelerated method. Research by Chen et al. (2023) [Chen et al., 2023] has shown a significantly larger speedup when using much larger matrices in a GPU accelerated method. Applying this to VPint will not only reduce the serialised part, as using a larger patch means that there will be less patches in total in a satellite image, but this could also result in an advantage for the GPU accelerated matrix computations compared to the CPU when using larger matrices.

9 Code Availability

The code that was used for running the experiments can be found at:
https://github.com/DeanvanLaar/AcceleratedVPint.

References

[Ajitsaria, 2018] Ajitsaria, A. (2018). What is the python global interpreter lock (gil)? https://realpython.com/python-gil/.

[Anselin, 2013] Anselin, L. (2013). *Spatial econometrics: methods and models*, volume 4. Springer Science & Business Media.

[Arp et al., 2022a] Arp, L., Baratchi, M., and Hoos, H. (2022a). Vpint: Github repository. https://github.com/ADA-research/VPint.

[Arp et al., 2022b] Arp, L., Baratchi, M., and Hoos, H. (2022b). Vpint: value propagation-based spatial interpolation. *Data Mining and Knowledge Discovery*, 36.

[Aziz et al., 2021] Aziz, A., Z., Naseradeen Abdulqader, D., Sallow, A. B., and Khalid Omer, H. (2021). Python parallel processing and multiprocessing: A rivew. *Academic Journal of Nawroz University*, 10(3):345–354.

[Balasooriya, 2017] Balasooriya, C. (2017). Overhead of parallelism.

[Balogh and Ruiz, 2022] Balogh, A. and Ruiz, R. (2022). A gpu accelerated genetic algorithm for the construction of hadamard matrices. *arXiv preprint arXiv:2208.14961*.

[Bhattacharjee et al., 2011] Bhattacharjee, A., Contreras, G., and Martonosi, M. (2011). Parallelization libraries: Characterizing and reducing overheads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1–29.

[Briesmeister et al., 2000] Briesmeister, J. F. et al. (2000). Mcnptm-a general monte carlo n-particle transport code. *Version 4C, LA-13709-M, Los Alamos National Laboratory*, 2.

[Bryant and O'Hallaron, 2015] Bryant, R. E. and O'Hallaron, D. R. (2015). *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition.

[Buber and Diri, 2018] Buber, E. and Diri, B. (2018). Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pages 1–6.

[Chen et al., 2017] Chen, B., Huang, B., Chen, L., and Xu, B. (2017). Spatially and temporally weighted regression: A novel method to produce continuous cloud-free landsat imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 55(1):27–37.

[Chen et al., 2023] Chen, W., Moraitis, P., Hansson, J., and Davidson, L. (2023). Gpu-accelerated computational methods using python and cuda.

[Cheng and Han, 2016] Cheng, G. and Han, J. (2016). A survey on object detection in optical remote sensing images. *ISPRS Journal of Photogrammetry and Remote Sensing*, 117:11–28.

[Cheng et al., 2014] Cheng, Q., Shen, H., Zhang, L., Yuan, Q., and Zeng, C. (2014). Cloud removal for remotely sensed images by similar pixel replacement guided with a spatio-temporal mrf model. *ISPRS Journal of Photogrammetry and Remote Sensing*, 92:54–68.

[Chrzeszczyk and Chrzeszczyk, 2013] Chrzeszczyk, A. and Chrzeszczyk, J. (2013). Matrix computations on the gpu. *CUBLAS and MAGMA by example*.

[Copernicus, nd] Copernicus (n.d.). Conventional data access hubs. https://www.copernicus.eu/en/access-data/conventional-data-access-hubs.

[Copernicus, 2024] Copernicus, B. (2024). Copernicus browser. https://browser.dataspace.copernicus.eu/.

[D'Agostino and Pearson, 1973] D'Agostino, R. and Pearson, E. S. (1973). Tests for departure from normality. empirical results for the distributions of b2 and b1. *Biometrika*, 60(3):613–622.

[D'Agostino, 1971] D'Agostino, R. B. (1971). An omnibus test of normality for moderate and large size samples. *Biometrika*, 58(2):341–348.

[Duque et al., 2012] Duque, J. C., Anselin, L., and Rey, S. J. (2012). The max-p-regions problem. *Journal of Regional Science*, 52(3):397–419.

[Ebel et al., 2021] Ebel, P., Meraner, A., Schmitt, M., and Zhu, X. X. (2021). Multisensor data fusion for cloud removal in global and all-season sentinel-2 imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 59(7):5866–5878.

[Engbrecht, 2008] Engbrecht, E. (2008). What does i/o bound really mean? https://erikengbrecht.blogspot.com/2008/06/what-does-io-bound-really-mean.html.

[ESA, nd] ESA, C. (n.d.). Esa copernicus sentinel 2. https://www.esa.int/Applications/Observing_the_Earth/Copernicus/Sentinel-2.

[GISGeography, 2024] GISGeography (2024). Sentinel 2 bands and combinations. https://gisgeography.com/sentinel-2-bands-combinations/.

[Greenlaw, 1992] Greenlaw, R. (1992). A model classifying algorithms as inherently sequential with applications to graph searching. *Information and Computation*, 97(2):133–149.

[Grochowski et al., 2004] Grochowski, E., Ronen, R., Shen, J., and Wang, H. (2004). Best of both latency and throughput. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, pages 236–243.

[Haining, 1978] Haining, R. P. (1978). The moving average model for spatial interaction. *Transactions of the Institute of British Geographers*, 3(2):202–225.

[Holm et al., 2020] Holm, H. H., Brodtkorb, A. R., and Sætra, M. L. (2020). Gpu computing with python: Performance, energy efficiency and usability. *Computation*, 8(1).

[Karimi et al., 2010] Karimi, K., Dickson, N. G., and Hamze, F. (2010). A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*.

[King et al., 2013] King, M. D., Platnick, S., Menzel, W. P., Ackerman, S. A., and Hubanks, P. A. (2013). Spatial and temporal distribution of clouds observed by modis onboard the terra and aqua satellites. *IEEE Transactions on Geoscience and Remote Sensing*, 51(7):3826–3852.

[Li et al., 2019] Li, X., Wang, L., Cheng, Q., Wu, P., Gan, W., and Fang, L. (2019). Cloud removal in remote sensing images using nonnegative matrix factorization and error correction. *ISPRS Journal of Photogrammetry and Remote Sensing*, 148:103–113.

[Ma et al., 2023] Ma, D., Wu, R., Xiao, D., and Sui, B. (2023). Cloud removal from satellite images using a deep learning model with the cloud-matting method. *Remote Sensing*, 15(4).

[Meraner et al., 2020] Meraner, A., Ebel, P., Zhu, X. X., and Schmitt, M. (2020). Cloud removal in sentinel-2 imagery using a deep residual neural network and sar-optical data fusion. *ISPRS Journal of Photogrammetry and Remote Sensing*, 166:333–346.

[Müller-Wilm et al., 2017] Müller-Wilm, U., Devignot, O., and Pessiot, L. (2017). S2 mpc sen2cor configuration and user manual.

[Nishino and Loomis, 2017] Nishino, R. and Loomis, S. H. C. (2017). Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7).

[NVIDIA, 2021] NVIDIA (2021). Geforce gtx 1080 ti. https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1080-ti/specifications/.

[Owens et al., 2008] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879–899.

[Park and Park, 2022] Park, S. and Park, N.-W. (2022). Cloud removal using gaussian process regression for optical image reconstruction. *Korean J. Remote Sens*, 38:327–341.

[Peng et al., 2021] Peng, X., Zhong, R., Li, Z., and Li, Q. (2021). Optical remote sensing image change detection based on attention mechanism and image difference. *IEEE Transactions on Geoscience and Remote Sensing*, 59(9):7296–7307.

[PlanetLabs, nd] PlanetLabs (n.d.). Sentinel-2 l2a. https://docs.sentinel-hub.com/api/latest/data/sentinel-2-l2a/.

[Qian, 2015] Qian, S.-E. (2015). *Optical payloads for space missions*. John Wiley & Sons.

[Ramuglia, 2023] Ramuglia, G. (2023). What are cpu threads? cores vs threads explained. https://ioflood.com/blog/what-are-cpu-threads-cores-vs-threads-explained/.

[Sazali et al., 2022] Sazali, M. A., Sarkawi, M. S., and Ali, N. S. M. (2022). Multiprocessing implementation for mcnp using python. *IOP Conference Series: Materials Science and Engineering*, 1231(1):012003.

[Shafie Khorassani et al., 2021] Shafie Khorassani, K., Hashmi, J., Chu, C.-H., Chen, C.-C., Subramoni, H., and Panda, D. K. (2021). Designing a rocm-aware mpi library for amd gpus: Early experiences. In Chamberlain, B. L., Varbanescu, A.-L., Ltaief, H., and Luszczek, P., editors, *High Performance Computing*, pages 118–136, Cham. Springer International Publishing.

[Shamey and Zhao, 2014] Shamey, R. and Zhao, X. (2014). 5 - solving dynamic equations in dye transport. In *Modelling, Simulation and Control of the Dyeing Process*, Woodhead Publishing Series in Textiles, pages 100–113. Woodhead Publishing.

[Sharma, 2023] Sharma, H. (2023). Harnessing parallelism: How gpus revolutionize computing.

[Sindhu, 2018] Sindhu, V. (2018). Exploring parallel efficiency and synergy for max-p region problem using python.

[Su et al., 2012] Su, C.-L., Chen, P.-Y., Lan, C.-C., Huang, L.-S., and Wu, K.-H. (2012). Overview and comparison of opencl and cuda technology for gpgpu. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 448–451.

[Subhlok and Vondran, 1996] Subhlok, J. and Vondran, G. (1996). Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71.

[Team, 2022] Team, I. E. (2022). Gpu vs. cpu: What's the difference? (plus other faqs). https://www.indeed.com/career-advice/career-development/gpu-vs-cpu.

[Tuomanen, 2018] Tuomanen, B. (2018). *Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA*. Packt Publishing Ltd.

[Wilcoxon, 1992] Wilcoxon, F. (1992). *Individual Comparisons by Ranking Methods*, pages 196–202. Springer New York, New York, NY.

[Xu et al., 2022] Xu, P., Sun, M.-Y., Gao, Y.-J., Du, T.-J., Hu, J.-M., and Zhang, J.-J. (2022). Influence of data amount, data type and implementation packages in gpu coding. *Array*, 16:100261.

[Yoshida et al., 2024] Yoshida, K., Miwa, S., Yamaki, H., and Honda, H. (2024). Analyzing the impact of cuda versions on gpu applications. *Parallel Computing*, 120:103081.

Appendices

A Appendix: Error Metrics per Scene

Table 3: The average mean absolute errors (MAE) of the scenes of all the methods are shown below. When methods are bold printed, it indicates a statistically significant difference between these bold printed methods using a two-tailed Wilcoxon signed-rank test with $\alpha = 0.05$

Scene name	Original VPint method average MAE	Multiprocessing average MAE	GPU average MAE
Europe Urban Madrid	2.15e+02 ± 2.60e+04	9.89e+02 ± 1.10e+04	2.88e+03 ± 5.02e+04
Europe Cropland Hungary	**2.08e+02 ± 9.59e+01**	**2.08e+02 ± 9.94e+01**	**2.07e+02 ± 9.30e+01**
Europe Cropland Ukraine	**7.95e+18 ± 1.49e+20**	**3.47e+17 ± 4.74e+18**	**5.04e+16 ± 8.16e+17**
Africa Cropland Nile	1.37e+17 ± 2.83e+18	5.24e+05 ± 1.08e+07	2.16e+06 ± 3.34e+07
America Shrubs Mexico	1.29e+03 ± 2.04e+04	5.64e+02 ± 5.27e+03	7.01e+02 ± 6.15e+03
Asia Herbaceous Mongoliaeast	**1.24e+02 ± 9.89e+01**	**1.24e+02 ± 9.90e+01**	1.24e+02 ± 9.91e+01
Asia Urban Beijing	7.46e+02 ± 7.29e+03	**9.96e+02 ± 1.12e+04**	**9.72e+02 ± 1.03e+04**
Africa Herbaceous Southafrica	**1.08e+02 ± 5.59e+01**	**1.07e+02 ± 5.52e+01**	1.08e+02 ± 5.54e+01
Australia Shrubs South	2.98e+03 ± 3.53e+04	3.60e+03 ± 4.45e+04	3.38e+03 ± 4.08e+04
Asia Cropland India	2.72e+02 ± 1.14e+02	2.71e+02 ± 1.09e+02	2.73e+02 ± 1.15e+02
America Cropland Iowa	**1.70e+02 ± 3.51e+01**	**1.69e+02 ± 3.49e+01**	**1.70e+02 ± 3.51e+01**
Asia Herbaceous Kazakhstan	**1.19e+02 ± 5.41e+01**	1.19e+02 ± 5.46e+01	**1.19e+02 ± 5.41e+01**
America Herbaceous Peru	1.24e+02 ± 4.87e+01	1.24e+02 ± 4.90e+01	1.24e+02 ± 4.87e+01
Asia Shrubs Indiapakistan	**2.66e+02 ± 3.11e+03**	**3.40e+02 ± 4.16e+03**	**1.12e+02 ± 2.05e+02**
Australia Shrubs West	3.59e+02 ± 6.45e+02	3.70e+02 ± 6.99e+02	3.54e+02 ± 6.25e+02
America Urban Atlanta	2.14e+02 ± 2.05e+02	**2.14e+02 ± 1.99e+02**	**2.15e+02 ± 2.13e+02**
Asia Cropland China	1.38e+02 ± 1.06e+02	1.32e+02 ± 8.33e+01	1.57e+02 ± 2.88e+02
America Forest Mississippi	**2.30e+02 ± 2.74e+02**	**2.30e+02 ± 2.73e+02**	**2.30e+02 ± 2.70e+02**
Africa Forest Angola	**1.14e+02 ± 5.13e+01**	**1.14e+02 ± 5.17e+01**	**1.14e+02 ± 5.14e+01**
All scenes	5.50e+17 ± 3.87e+19	2.35e+16 ± 1.24e+18	3.42e+15 ± 2.13e+17

Table 4: The average mean squared errors (MSE) of the scenes of all the methods are shown below. When methods are bold printed, it indicates a statistically significant difference between these bold printed methods using a two-tailed Wilcoxon signed-rank test with $\alpha = 0.05$.

Scene name	Original VPint method Average MSE	Multiprocessing Average MSE	GPU Average MSE
Europe Urban Madrid	7.56E+11 ± 1.54E+13	9.48E+10 ± 1.92E+12	2.14E+12 ± 4.35E+13
Europe Cropland Hungary	1.28E+05 ± 1.57E+05	1.30E+05 ± 1.73E+05	1.27E+05 ± 1.51E+05
Europe Cropland Ukraine	**7.50E+44 ± 1.44E+46**	**7.43E+41 ± 1.14E+43**	**7.74E+40 ± 1.44E+42**
Africa Cropland Nile	2.26E+41 ± 4.67E+42	1.80E+17 ± 3.73E+18	1.78E+18 ± 3.37E+19
America Shrubs Mexico	5.44E+11 ± 7.79E+12	7.98E+11 ± 1.10E+13	4.31E+12 ± 7.90E+13
Asia Herbaceous Mongoliaeast	**5.79E+04 ± 1.86E+05**	**5.81E+04 ± 1.86E+05**	5.82E+04 ± 1.86E+05
Asia Urban Beijing	5.09E+11 ± 7.77E+12	**1.17E+12 ± 1.78E+13**	**9.04E+11 ± 1.38E+13**
Africa Herbaceous Southafrica	**3.75E+04 ± 4.24E+04**	**3.69E+04 ± 4.15E+04**	3.71E+04 ± 4.13E+04
Australia Shrubs South	2.68E+11 ± 3.80E+12	4.57E+11 ± 6.56E+12	3.72E+11 ± 5.32E+12
Asia Cropland India	**2.35E+05 ± 3.42E+05**	**2.30E+05 ± 3.39E+05**	**2.37E+05 ± 3.42E+05**
America Cropland Iowa	8.15E+04 ± 3.59E+04	8.13E+04 ± 3.59E+04	8.16E+04 ± 3.60E+04
Asia Herbaceous Kazakhstan	**3.46E+04 ± 4.50E+04**	**3.47E+04 ± 4.54E+04**	**3.46E+04 ± 4.52E+04**
America Herbaceous Peru	4.79E+04 ± 7.23E+04	4.79E+04 ± 7.21E+04	4.79E+04 ± 7.25E+04
Asia Shrubs Indiapakistan	1.86E+09 ± 3.76E+10	3.26E+09 ± 6.53E+10	1.01E+06 ± 1.98E+07
Australia Shrubs West	9.57E+08 ± 4.41E+09	1.09E+09 ± 5.08E+09	8.68E+08 ± 3.95E+09
America Urban Atlanta	1.05E+08 ± 1.02E+09	**9.49E+07 ± 9.63E+08**	**1.10E+08 ± 1.11E+09**
Asia Cropland China	4.36E+07 ± 2.56E+08	2.69E+07 ± 1.57E+08	5.24E+08 ± 4.90E+09
America Forest Mississippi	**2.36E+05 ± 8.15E+05**	**2.31E+05 ± 7.93E+05**	**2.29E+05 ± 7.88E+05**
Africa Forest Angola	**4.05E+04 ± 3.63E+04**	**4.09E+04 ± 3.67E+04**	**4.05E+04 ± 3.62E+04**
All data	5.08E+43 ± 3.74E+45	5.03E+40 ± 2.98E+42	5.25E+39 ± 3.74E+41

B Appendix: Running Times per Scene

Table 5: The average running time in seconds of VPint of the scenes of all methods are presented below. A bold printed value indicates the method that performed significantly better than the most other methods using a one-tailed Wilcoxon signed-rank test with $\alpha = 0.05$.

Scene name	Original VPint method average running time	Multiprocessing VPint average running time	GPU VPint average running time
Europe Urban Madrid	184.67 ± 50.42	**38.44 ± 0.85**	112.23 ± 33.28
Europe Cropland Hungary	202.36 ± 65.03	**39.54 ± 1.46**	124.87 ± 39.25
Europe Cropland Ukraine	406.56 ± 44.47	**41.41 ± 1.00**	265.44 ± 29.69
Africa Cropland Nile	351.98 ± 95.07	**40.54 ± 1.30**	213.22 ± 59.49
America Shrubs Mexico	181.95 ± 19.55	**38.49 ± 0.37**	103.69 ± 12.92
Asia Herbaceous Mongoliaeast	167.74 ± 11.74	**38.45 ± 0.27**	102.57 ± 8.45
Asia Urban Beijing	246.95 ± 104.36	**39.19 ± 1.45**	161.85 ± 66.74
Africa Herbaceous Southafrica	169.82 ± 23.19	**38.53 ± 0.34**	104.10 ± 14.58
Australia Shrubs South	164.14 ± 7.04	**38.40 ± 0.30**	99.53 ± 6.17
Asia Cropland India	268.64 ± 84.43	**40.37 ± 1.56**	164.22 ± 54.03
America Cropland Iowa	186.09 ± 28.16	**38.77 ± 0.96**	113.91 ± 17.97
Asia Herbaceous Kazakhstan	177.16 ± 21.18	**38.20 ± 0.54**	108.75 ± 14.81
America Herbaceous Peru	180.57 ± 22.09	**38.10 ± 0.34**	110.77 ± 14.70
Asia Shrubs Indiapakistan	229.09 ± 57.73	**37.95 ± 0.90**	148.25 ± 38.22
Australia Shrubs West	173.53 ± 29.34	**38.66 ± 0.64**	106.45 ± 21.73
America Urban Atlanta	164.97 ± 16.31	**38.44 ± 0.73**	103.06 ± 9.61
Asia Cropland China	197.78 ± 67.58	**38.60 ± 0.94**	122.48 ± 43.04
America Forest Mississippi	166.43 ± 8.76	**38.50 ± 0.42**	101.40 ± 6.52
Africa Forest Angola	176.34 ± 28.97	**38.27 ± 0.47**	106.78 ± 19.45
All scenes	217.69 ± 88.42	**39.02 ± 1.48**	132.21 ± 54.52

Table 6: The average running time in seconds for calculating a patch (including pre- and post-processing) of the scenes of all methods are presented below. A bold printed value indicates the method that performed significantly better than the most other methods using a one-tailed Wilcoxon signed-rank test with $\alpha = 0.05$.

Scene name	Original VPint method patch average running time	Multiprocessing patch average running time	GPU patch average running time
Europe Urban Madrid	174.73 ± 50.52	**37.22 ± 0.94**	106.71 ± 33.37
Europe Cropland Hungary	68.61 ± 65.07	**13.95 ± 1.51**	42.66 ± 39.31
Europe Cropland Ukraine	363.43 ± 44.52	**38.14 ± 0.95**	237.79 ± 29.78
Africa Cropland Nile	353.24 ± 95.22	**41.95 ± 1.39**	214.56 ± 59.71
America Shrubs Mexico	171.83 ± 19.58	**37.21 ± 0.41**	98.37 ± 12.95
Asia Herbaceous Mongoliaeast	55.75 ± 11.78	**13.31 ± 0.31**	34.38 ± 8.50
Asia Urban Beijing	135.19 ± 104.50	**22.25 ± 1.59**	88.96 ± 66.95
Africa Herbaceous Southafrica	90.47 ± 23.27	**21.18 ± 0.39**	55.82 ± 14.66
Australia Shrubs South	80.32 ± 7.05	**19.33 ± 0.37**	49.02 ± 6.18
Asia Cropland India	92.64 ± 84.56	**14.56 ± 1.72**	56.95 ± 54.20
America Cropland Iowa	186.73 ± 28.20	**39.90 ± 1.01**	114.82 ± 18.02
Asia Herbaceous Kazakhstan	168.58 ± 21.24	**37.32 ± 0.58**	103.99 ± 14.90
America Herbaceous Peru	175.11 ± 22.15	**37.99 ± 0.33**	107.96 ± 14.79
Asia Shrubs Indiapakistan	220.97 ± 57.84	**37.88 ± 0.97**	143.60 ± 38.36
Australia Shrubs West	12.68 ± 29.32	**3.30 ± 2.35**	8.01 ± 21.72
America Urban Atlanta	93.55 ± 16.31	**22.51 ± 0.67**	58.80 ± 9.60
Asia Cropland China	85.95 ± 67.70	**17.47 ± 1.06**	53.58 ± 43.19
America Forest Mississippi	64.83 ± 8.82	**15.62 ± 0.44**	39.80 ± 6.58
Africa Forest Angola	137.29 ± 29.07	**30.67 ± 0.50**	83.60 ± 19.58
All scenes	143.78 ± 88.52	**26.41 ± 1.55**	89.44 ± 54.64

C Appendix: VPint Patches and Cloudless Patches per Scene

Table 7: The number of patches that VPint runs on per scene, the number of patches that do not have clouds, and thus do not run VPint and the total number of patches per scene can be found in the table below.

Scene name	Number of patches running VPint	Number of cloudless patches	Total patches
Europe Urban Madrid	415	26	441
Europe Cropland Hungary	148	293	441
Europe Cropland Ukraine	393	48	441
Africa Cropland Nile	441	0	441
America Shrubs Mexico	414	27	441
Asia Herbaceous Mongoliaeast	145	296	441
Asia Urban Beijing	240	201	441
Africa Herbaceous Southafrica	233	208	441
Australia Shrubs South	214	227	441
Asia Cropland India	151	290	441
America Cropland Iowa	440	1	441
Asia Herbaceous Kazakhstan	417	24	441
America Herbaceous Peru	425	16	441
Asia Shrubs Indiapakistan	423	18	441
Australia Shrubs West	31	410	441
America Urban Atlanta	248	193	441
Asia Cropland China	190	251	441
America Forest Mississippi	170	271	441
Africa Forest Angola	341	100	441