Asia-Pacific Journal of Science and Technology: Volume: 29. Issue: 02. Article ID.: APST-29-02-07. Research Article

APST

Asia-Pacific Journal of Science and Technology

https://www.tci-thaijo.org/index.php/APST/index

Published by the Research and Graduate Studies Division,
Khon Kaen University, Thailand

Satellite products computation with multiple GPU devices

Chaiyasit Tanchotsrinon, Wongnaret Khantuwan* and Noppadon Khiripet

Knowledge Elicitation and Archiving Laboratory (KEA), National Electronics and Computer Technology Center (NECTEC), Pathumthani 12120, Thailand
*Corresponding author: chaiyasit.tan@nectec.or.th

Received 15 October 2022
Revised 18 April 2023
Accepted 19 April 2023

Abstract

Over recent years, the number of earth observation satellites has increased dramatically. The satellite data gathering from various sources must be prepared and processed into satellite products (indexes), which takes computational time. Therefore, parallel computing techniques should be utilized in reducing time complexity. Graphic Processing Unit (GPU) devices are widely used to accelerate the computation process by massively parallel operations and, thus, are very suitable for this task. Although the index calculations were not complicated, the major drawback of the GPU process is data transfers between the host and the devices. Once data is transferred, it should be reused to calculate all related satellite indexes instead of beginning the transfer-compute cycle. In this paper, we investigate an efficacy way to produce satellite indexes and compare the computational times among many different approaches, i.e., Central Processing Unit (CPU) process executed based on NumPy and extended to multi-thread by Dask, single GPU device process performed with Numba, and multiple GPU device processes launched asynchronously. The experiments were carried out on three hardware environments, namely, the DGX workstation, the High-performance computing (HPC) High memory node, and the HPC-DGX node. The results revealed that the proposed GPU processes achieved more than ten times faster in overall. Furthermore, when compared with the CPU process, it found that its kernel computation could achieve more than 250 times faster.

Keywords: CUDA, LAI, MSAVI, NDVI, Parallel programming

1. Introduction

Over recent years, a dramatic increase in the number of earth observation satellites, image resolution, and frequency range give us unprecedented levels of collect to satellite data. A massive amount of satellite data is gathered from various sources. Those data must be prepared and calculated for satellite products (indexes), which takes computational time. Hence, the size of the satellite data that needed to be computed was huge. For instance, Sentinel-2 products are freely accessible and downloaded from Copernicus [1]. They divided the earth's surface into a grid called tiled. Each tile consists of size 100x100 km^2 ortho-images in UTM/WGS84 projection. It is approximately one hundred million pixels on each tile. In Thailand, there are over ninety tiles. Thus, more than nine thousand million pixels were required to compute in each satellite's orbit per one time series. As previously mentioned, the satellite data needed to be calculated from several sources, and, in addition, it was needed for exporting satellite index products for every source. The satellite data were generally used in various kinds of applications. Several satellite indexes have been adopted to analyze satellite data, such as Normalized Difference Vegetations Index (NDVI) [2], Normalized Difference Water Index (NDWI) [3], Leaf area index (LAI) [4], and Soil Adjusted Vegetation Index (SAVI) [5], etc. Remote sensing indices were used to represent or indicate the areas. Therefore, a computational resource should be optimized as well as possible.

Due to artificial intelligence blooming, parallel programming approaches are developed for handling computational time consumption problems by GPUs. This idea plays a significant role in parallel computing since it not only reduces the computational time, but also optimize resources and energy. In some operations, Graphic Processing Unit (GPU) computing could be performed more than one hundred times faster than Central Processing

Unit (CPU) computing. However, not all operations can be accelerated by GPU parallel computing. Although GPU parallel programming is not designed to speed up basic operations like normalized satellite indexes calculation that is not a complicated formula, it can bring great benefits when the size of satellite data that needed to be computed is huge. Thus, the product calculation should be sped up by implementing pixel-wise parallel computing.

Juan et al. [6] presented the NDVI algorithm calculated based on Compute Unified Device Architecture (CUDA) [7]. The CUDA kernel launches in several image block sizes. Guerrouj et al. [8] aimed to calculate satellite indexes (NDVI and NDWI) with real-time streaming data. CPU-GPU heterogeneous has been adopted to solve these problems. Zou et al. [9] showed different techniques for implementing the NDVI index. The method is based on the OpenAcc platform and claims to be 5.3 times faster than traditional serial programming without losing accuracy.

For this paper, we would like to provide an investigation of the efficient way for exporting satellite products. The following sections will compare the CPU and GPU with three hardware environments. The hardware environments have been set to cover data processing types. The hardware environments can be divided roughly into three main categories by capabilities of processing levels which are described further in the next section. The proposed method was implemented based on the Python programming language. Python is a well-known scientific programming that provides rich scientific libraries with free of charge. Four Python libraries have been chosen to compare CPU and GPU time consumption which are NumPy [10], Dask [11, 12], Numba [13], and concurrent futures [14]. The NumPy was designed for array programming and numerical computing. By the way, NumPy is represented as a CPU process. CPU process with Multi thread has been executed by combinations of NumPy and Dask. Dask has been adopted as task assignment in parallel CPU multithreading process.

Here, we investigate multiple-GPU processing solutions. We suggest adding a planning stage before transmitting data for computing instead of comparing it to traditional techniques, which can assist in shortening computation times for large-scale calculations. The outcomes of this approach will be demonstrated in full detail during the discussion section. In order to speed up the execution of band ratio computation and morphological operations, Bhangale et al. [15] demonstrated using GPU by the traditional kernel launch technique executed in CUDA C. The process computes results from the GPU using the traditional kernel launch technique [16] in C/C++, while the alternative way was decided to be implemented in Python for the proposed method, a decision that we will discuss later. The results from this study demonstrate that the calculation time decreases when the batch size is divisible while somewhat increasing when the batch size is not divisible. Our experimental findings support the previously stated argument for the same reason. GPU processing has several advantages over CPU processing, for instance, when compared to CPU calculation, it is obvious that using the GPU for computation can significantly decrease computation time.

In contrast, Numba was designed to simplify GPU parallel programming with a just-in-time (JIT) compiler. Numba was involved in several studies to reduce computation time [17, 18]. Numba with @jit decorations was an automatically optimal Python code that can be executed similarly to CUDA C or Fortran. The proposed method combined concurrent.futures and Numba to launch kernel function as an asynchronous process compared with traditional kernel launch and Numba @jit (plus parallel flag). The method could reduce computational time efficiently. In addition, the results show that the proposed method can be more than ten times faster than the CPU process.

2. Materials and methods

2.1 Dataset specifications

Datasets have been divided into two categories. Firstly, synthesis data will be generated random numbers with the same range of sentinel-2 data values into an array of 32 bits floating points. This data will be used to examine elementwise operations for exporting satellite products. Secondly, sentinel-2 data (Level-2A) was implemented based on this applied data size.

2.2 Hardware environments

This study aimed to design for a large-scale data process. Multiple graphic devices have been adopted to parallelly compute each device's kernel function. NSTDA Supercomputer Center: ThaiSC [19] has provided TARA HPC Cluster for the experiments. TARA consists of compute node, high-memory node, GPU node, and DGX node. High-memory and DGX nodes have been chosen for processing large data in CPU and GPU, respectively. The proposed method has been executed only one node at a time. Message Passing Interface (MPI) [20] was not involved in these experiments. The small data size might not be properly manifest with algorithm performance and the size of HPC data. Therefore, DGX Station has been involved in investigating small data sizes as preliminary results provided by executing CPU and GPU processes. The detail of hardware specifications is shown in Table 1.

Table 1 Hardware Environment.

DGX stations V100		ThaiSC HPC (TARA)	
		High-memory node	DGX node (DGX-1)
Central Processing Unit (CPU)	1 x Intel Xeon E5-2698 v4 (20 core 40 threads, 2.2 – 3.6 GHz)	4 x Intel Xeon® Gold 6148 (24 core 48 threads, 2.1 – 3.7 GHz)	1 x Intel Xeon E5-2698 v4 (20 core 40 threads, 2.2 – 3.6 GHz)
Memory (RAM)	252 GB	3 TB DDR4	512 GB DDR4
Storage types	SSD	SSD + SAS	SSD + SAS
Graphic Processing unit (GPU)	4 x Nvidia Tesla V100 (16 GB per device)	-	8 x Nvidia Tesla V100 (32 GB per device)
Operating system	Ubuntu 18.04.6 LTS	CentOS 7	CentOS 7
Dataset specifications	Synthesis Data & Sentinel-2 Data		

2.3 Software environments

Parallel computing in this experiment is specified based on CUDA. CUDA has adopted GPU to accelerate programming executions. CUDA was developed in many programming languages, such as C, C++, Fortran, Python, and MATLAB. CUDA was widely implemented based on C, C++, or Python rather than the others. Fortran was designed for parallel programming but has fewer users than other programming languages. Although MATLAB is widely used as a scientific programming platform and provides several toolboxes, it was not free of charge compared to Python.

Therefore, this method was implemented based on Python instead of C or C++ programming languages since Python is open-source software with free of charge and requires a low learning curve. Python CUDA is slightly slower than C or C++ CUDA, but it takes advantage of the automatic garbage collection of resources and provides plentiful scientific libraries. For this reason, Python is more suitable than C++ in such a way that the satellite indexes might be further used for creating classification models in artificial neural network. That is why Python programming was chosen. Besides, all operating systems were based on Linux.

This study uses asynchronous coding as a planner to launch kernel function independently operation in each GPU device. Hence, the devices were individually executing kernel functions independently. The processing of the proposed method can be illustrated as Figure 1.

2.4 Normalized data

This experiment downloaded satellite data freely from COPENICIOUS, The European Space Agency (ESA) [1]. According to Sentinel-2 user guides [21], the data must be normalized before use. Briefly, the raw data must be normalized by dividing ten thousand, but since January 25, 2022, those data must be minus with a value depending on the Extensible Markup Language (XML) file and then divided by one thousand. Those dates were restricted to the Thailand area only. Other places might be varied in date. The normalized equations are revealed as Equation (1).

$$L2A\_B0Ai = \frac{L2A_{DNi} + B0A_{ADD_{OFFSETi}}}{QUANTIFICATION_{VALUEi}} \tag{1}$$

$$NDVI = \frac{NIR - RED}{NIR + RED} \tag{2}$$

$$SAVI = \frac{(1 + L) \times (NIR - RED)}{L + NIR + RED}\ ;\ L = 0.428 \tag{3}$$

$$LAI = -\log\frac{(0.69 - SAVI)/0.59}{0.91} \tag{4}$$

$$MSAVI = \frac{\left(2 \times NIR + 1 - \sqrt{(2 \times NIR + 1)^2 - 8 \times (NIR - RED)}\right)}{2} \tag{5}$$

2.5 Normalized satellite index equations

To measure the performance of the method, synthesis data has been computed by elementwise operations with normalized satellite indexes. The indexes consist of NDVI, Modified soil adjusted vegetation index (MSAVI),

and LAI. The LAI can be calculated by several equations, so a traditional one has been chosen. Then, all the formulas used in this experiment are as shown in Equations (2) – (5).

2.6 CPU Programming

NumPy is a well-known Python library used for scientific computing multidimensional arrays. Since NumPy was executing a single thread by default. Therefore, the Dask library has been used for providing parallel computing in Python. It can be operated in both CPU and GPU processes. In this scenario, it was used with NumPy to perform the CPU multithreading process.

2.7 GPU Programming

2.7.1 Python – Numba

Numba was chosen as an open-source Python library that automatically translates Python code into optimized machine code. Numba has provided several Python decorator functions such as @jit (Just-In-Time), @vectorize, @guvectorize (generalized universal function), @stencil, etc. JIT decorator is used with flags, no-Python, and parallel modes. These flags can be automatically executed by Python code with a contactless Python interpreter and automatically perform GPU parallel execution. The decorator has been taken as primitive parallel programming. Therefore, Python users do not need to step out of the Python comfort zone. @jit decorator with the parallel flag was used as a baseline for automatic parallel computing. However, the limitation of Numba is that some Python functions or libraries cannot perform as GPU Parallelize.

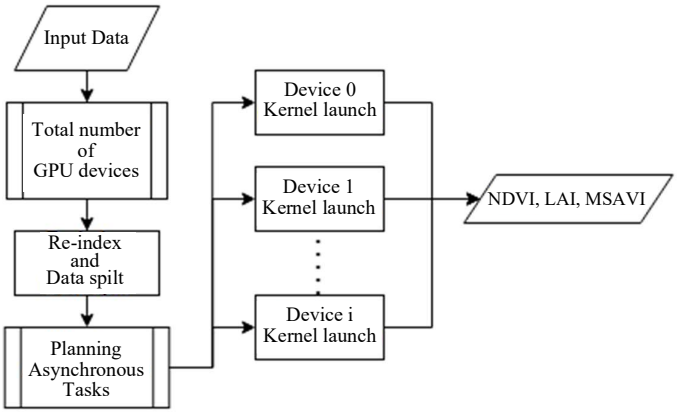

Figure 1 Flowchart of the proposed method.

2.7.2 Kernel functions

CUDA programming was composed of three-main process components: Firstly, data must be transferred from host memory to GPU device memory (Host to Device). Secondly, GPU programming was loaded and then executed. Lastly, the results of GPU programming were transferred back to the host memory (Device to Host). The hierarchical structure of CUDA consists of Grids, Blocks, and Threads. A group of threads was defined as Block. Likewise, the group of blocks is represented as Grid. To launch any CUDA kernel function, those three dimensions must be assigned relating to the specification of GPU devices and data shape. The dimension can be implemented in various dimensions, such as 1D, 2D, and 3D. In this study, the One-Dimensional kernel function has been chosen based on the shape of the data and asynchronous algorithm. The data shape can be anything. Thus, 2D or 3D might not be suitable for this reason. In this case, the efficiency for designed size was selected based on divisible kernel sizes. Performance might be affected by those fractions. Hence, the Data array must be re-shaped in vector. The data has been split into groups depending on which GPU device models (GPU memory capacity) and the number of GPU devices.

2.7.3 Parallel multiple devices

Serial processing was regularly executing code in order line by line. On the other hand, some tasks do not need to wait for the earlier task to be accomplished. Elementwise operations were counted in the same categories as

well. Therefore, parallel processing stepped in to reduce computation time. Asynchronous coding techniques were widely used in parallel programming. The proposed method is used concurrent futures for wrapping and planning CUDA kernel operations in each device. The devices can independently execute operations almost at the same time. Figure 1 presents a flowchart of the proposed method. Additionally, the traditional CUDA kernel launch and the proposed method are described in pseudo-code as shown in Algorithm 1 and 2, respectively.

Algorithm_1: Traditional multiple-device kernel launch

```
1. Input = NIR_band, Red_band
2. Output = NDVI, LAI, MSAVI
3. def kernel_function(TPB, BPG, NIR, RED, NDVI, LAI, MSAVI):
4. TPB = threadsperblock
5. BPG = blockspergrids
6. total_gpu_devices = len(cuda.gpus)
7. for i in range(total_gpu_devices):
8.     cuda.selected_device(device_i)
9.     kernel_functions[BPG, TPB](NIR, RED, NDVI, LAI, MSAVI)
```

Algorithm_2: Proposed method

```
1. Input = NIR_band, Red_band
2. Output = NDVI, LAI, MSAVI
3. # wrap traditional kernel_function with concurrent.futures
4. # Number of future objects depends on user design
5. with concurrent.futures.ThreadPoolExecutor(max_worker) as executor:
6.     for i in range (num_gpu_devices):
7.         future_obj[i] = executor.submit(kernel_function, device[i])
8.     concurrent.futures.as_completed(future_obj[all])
9. executor.shutdown(wait=True)
```

3. Results

For the experiment, various data sizes are used to investigate relations between an appropriate size and the performance of the CUDA kernel function. The data is fetched to the main memory (RAM) through numerous Python libraries such as NumPy, Xarray, Rasterio, rioxarray, Open Data Cube (ODC), etc. Once data is loaded, the data will be able to calculate all the satellite indexes, as shown in Table 2. The size of the data could be affected by computation time. Small data size might not be worthy for executing operations on GPU programming because data transferring cost was massive compared to running time on CPU. Hence, CPU computing could be completed before GPU data moving was finished in the case of small data.

Consequently, the loading operations from storage to memory will be excluded for the reason as same as writing from memory to storage. The HPC will take massive advantage of scratch space to write large-scale data into storage. However, this study still counts the data transfer times between host-memory and device-memory in the workflow of GPU computing, which is a significant drawback in GPU parallel computing.

Table 2 DGX stations (GPU single device VS. Multiple devices) – Computation time.

Method / Size of pixels, Computation time (sec)		100M pixels	200M pixels	300M pixels	400M pixels	500M pixels
CPU	NumPy	2.7473	5.4602	8.3687	10.9728	13.6795
GPU -Single device, Kernel profiling	Host to Device	0.2983	0.5565	0.9131	1.1930	1.4832
	Kernel executes (including CUDA overhead)	0.0519	0.0509	0.0525	0.0486	0.0541
	Device to Host	0.1526	0.3192	0.4718	0.6160	0.7699
	Total Consumption time	0.5029	0.9268	1.4376	1.8577	2.3073
GPU -Multiple devices	Numba @jit(parallel=True)	0.5161	0.7929	1.0820	1.3572	1.6307
	Proposed method	0.2446	0.4296	0.6089	0.7961	0.9820

4. Discussion

In comparison, Numba - @jit was automatically run in parallel and used as a baseline in this experiment. The running time of every technique was measured by an average of ten times repeating. Comparing the performance of each method has been represented in Tables 3 , Table 4 , Figure 2, and Figure 3. The result shows that single and multiple GPU processing is much faster than CPU computing. The speed-up ratio between CPU processing,

single GPU processing, and multi-GPU processing is related to the data size. The CPU will execute faster than GPU if the data size is too tiny.

A GPU device can handle its computation efficiently when all data is within its limited GPU memory. However, multiple GPU devices will be used to optimize all operations when the data size is massive. The traditional kernel launch method in multiple devices has been delayed by I/O bound operations in assigning data movement for execution. The proposed method alleviated this drawback by using concurrent.futures to wrap each kernel's instructions into future objects. The future objects can be executed independently on the assigned device at nearly the same time. The results showed that the proposed method is better at optimizing computation time than other methods.

Moreover, the speed-up ratios between CPU processing and GPU processing performance are shown in Figures 3 (a-b). For the DGX station experiment, the ratios between CPU and GPU were not increased due to the limitation that the data size could not be increased by more than 252 GB of memory space. Furthermore, the speed-up ratio seems to grow slightly when the data size can be divisible by kernel size, as shown in Figure 3 (a). However, the speed-up ratios do not significantly increase when the data size was larger than two thousand million pixels. The amount of data that affects computation time is related to divisible data size with kernel size. The optimum point is when data size equals GPU memory space. The speed-up of the HPC DGX node experiment can be grown in Figure 3 (b). As shown in Tables 3 and 4, the computation time of the same data size of 1,000 to 3,000 million pixels on the HPC-DGX nodes was slightly slower when compared to the DGX station. This might be caused by the data size being too small compared to the hardware specifications of HPC. The computation time on the HPC-DGX node might be slower than the DGX Station machine because of cluster memory architecture, data transferring between nodes (front-end node to high memory node and HPC-DGX node), and overhead from the SLURM workload manager. However, that does not mean the HPC-node performance is worse than the DGX Station. Since the DGX Station has only four GPU devices with 16GB of memory for each device, on the other hand, DGX-node consists of eight GPUs with 32GB of memory for each device; this made the DGX-node able to handle the larger data than DGX Station. In real situations, many operations are also included to finish the entire workflow, such as reading, writing, etc. Overall, it made the computation on HPC-DGX faster than DGX Stations.

Table 3 DGX stations (Multiple devices) – Computation time.

Method / Total number of pixels in millions (Time consumption - secs)	2 M	4 M	6 M	8 M	10 M	20 M	40 M	60 M	80 M	100 M	200 M	400 M	600 M	800 M	1000 M	2000 M	3000 M
CPU – NumPy single thread	0.04	0.09	0.14	0.19	0.28	0.55	1.12	1.67	2.19	2.74	5.45	11.20	16.29	21.93	27.70	55.70	84.69
GPU - traditional kernel launch	0.11	0.11	0.13	0.14	0.14	0.19	0.27	0.36	0.45	0.53	0.95	1.79	2.57	3.37	4.21	8.49	13.07
GPU - Numba @jit (parallel=True)	0.22	0.23	0.24	0.24	0.25	0.29	0.34	0.39	0.43	0.49	0.76	1.30	1.88	2.41	3.45	6.26	9.19
GPU - Proposed method – NDVI	0.10	0.10	0.10	0.10	0.10	0.10	0.14	0.15	0.17	0.20	0.30	0.53	0.76	0.97	1.21	2.29	3.33
GPU - Proposed method LAI	0.10	0.10	0.10	0.10	0.10	0.11	0.14	0.16	0.17	0.21	0.32	0.53	0.76	0.99	1.22	2.39	3.44
GPU - Proposed method MSAVI	0.11	0.11	0.11	0.12	0.11	0.12	0.15	0.16	0.18	0.22	0.32	0.54	0.77	0.97	1.21	2.31	3.44
GPU - Proposed method (Sum time computation of NDVI, LAI, MSAVI)	0.31	0.31	0.31	0.31	0.31	0.34	0.43	0.47	0.53	0.63	0.94	1.60	2.28	2.93	3.64	6.99	10.21
GPU - Proposed method (calculating 3 products at the same time)	0.12	0.11	0.10	0.10	0.11	0.14	0.17	0.19	0.23	0.29	0.47	0.86	1.22	1.56	1.99	3.86	6.05

Table 4 ThaiSC (CPU – High memory node, GPU – DGX node) – Computation time.

Method /	1,000 M pixels	2,000 M pixels	3,000 M pixels	6,000 M pixels	9,000 M pixels	12,000 M pixels

Total number of pixels in millions (Computation time - sec)						
CPU - NumPy single thread	34.5240	68.9926	108.5673	206.8101	366.5507	656.4981
CPU - Numba + Dask Multi-threads	10.9850	20.1078	28.6280	68.3279	102.7995	132.2834
GPU - traditional kernel launch	6.6022	13.8750	21.0548	48.1403	75.9102	95.2698
GPU - Numba jit (parallel=True)	3.2901	6.2925	9.4978	17.7719	27.3729	36.3783
GPU - Proposed method (3 products at the same time)	2.6602	4.5844	7.2360	14.0662	22.3275	29.7365

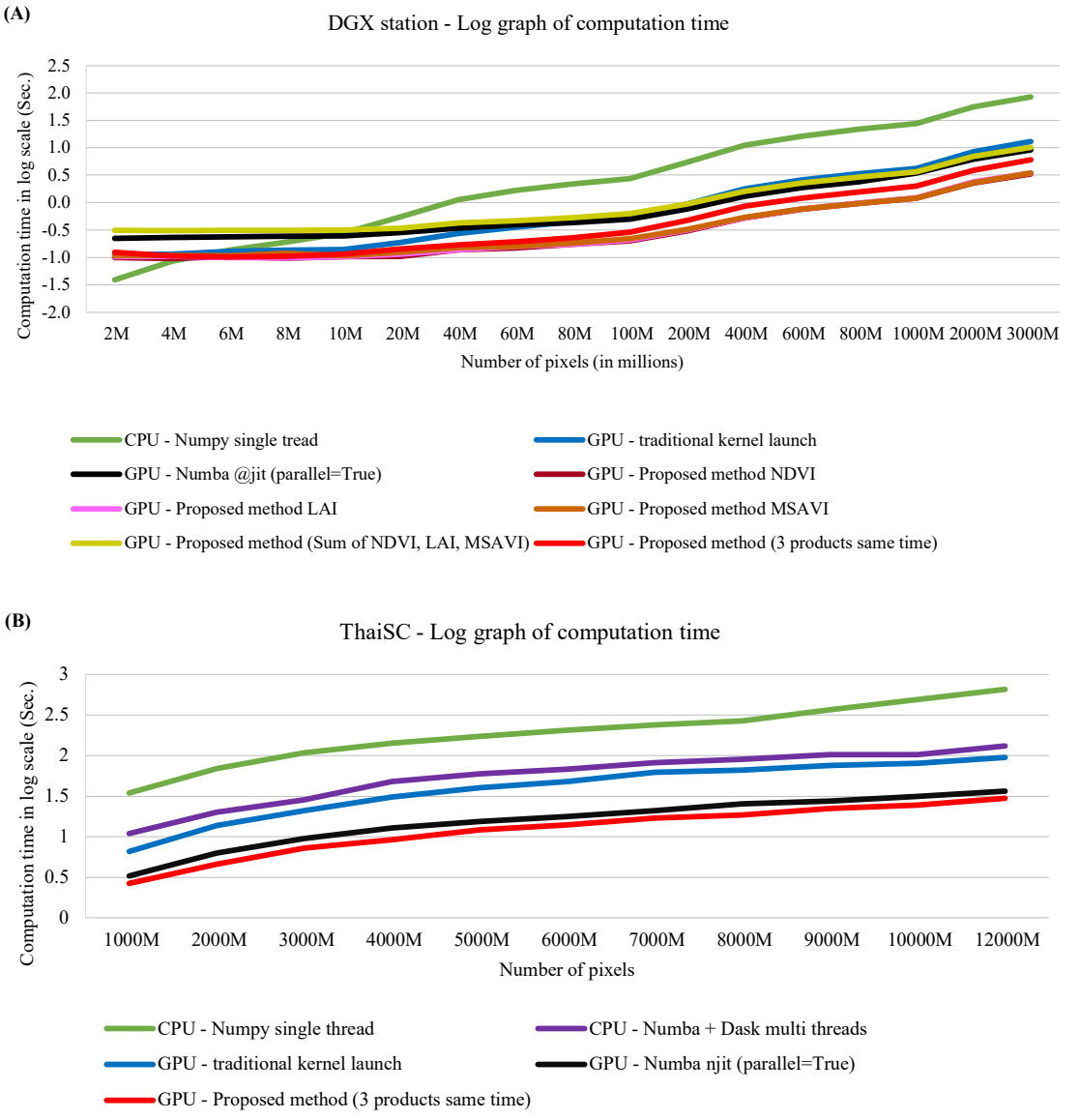

Figure 2 DGX station and ThaiSC HPC – LOG graph of the comparison between the size of pixels and computation time. (A) – DGX station, (B) – ThaiSC HPC.

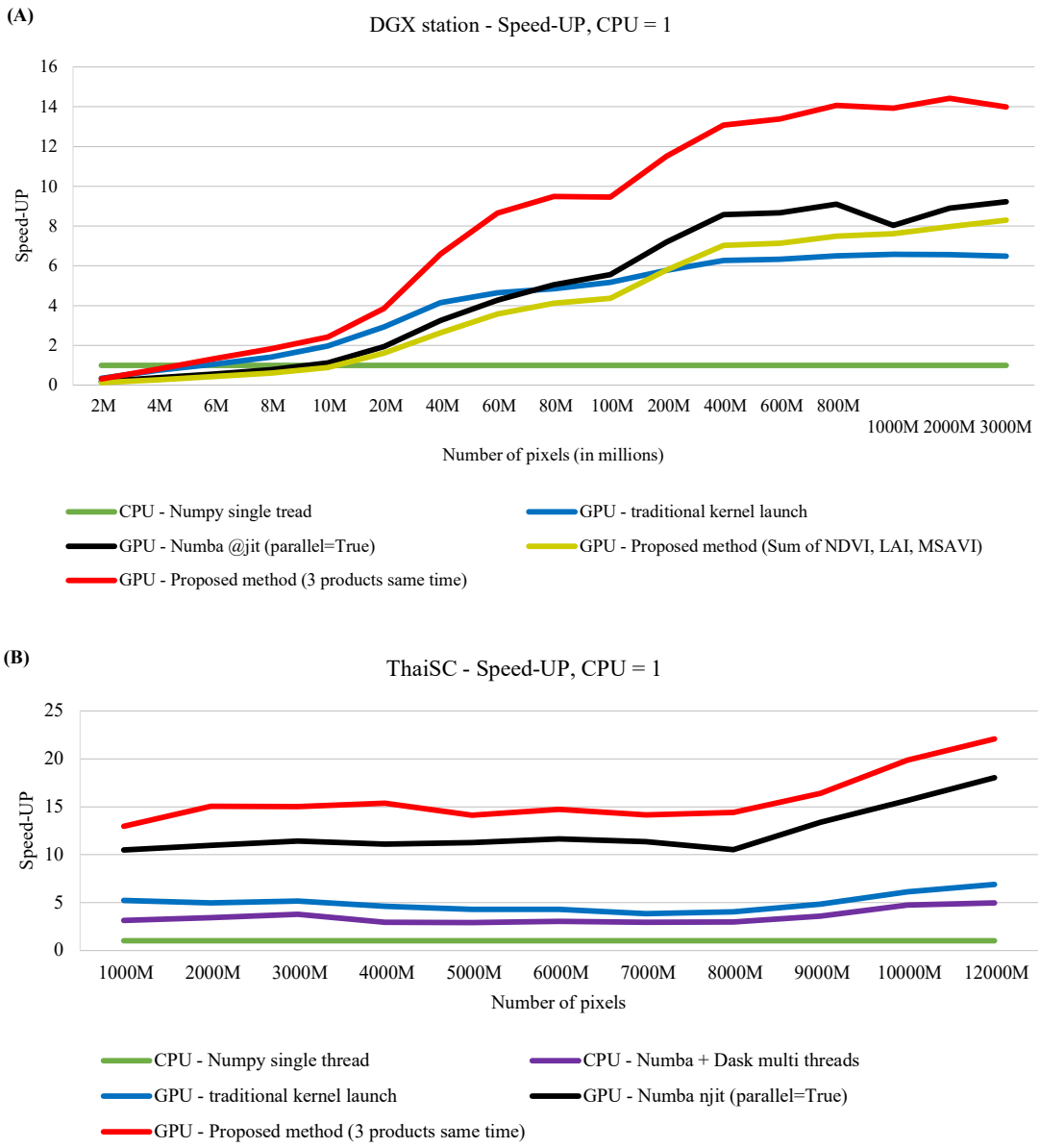

Figure 3 DGX station and ThaiSC – Speed-UP Graph. (A) – DGX station, (B) – ThaiSC HPC.

When comparing the kernel function excluding data transfer time between the CPU and GPU, the time needed for computing with GPU is more than 250 times faster than those with CPU. Practically, we must consider the time required for sending data to the GPU for processing. Depending on the quantity of the data, this only accelerates processes ten to twenty times faster than a CPU. However, the major drawback of the GPU process is shown in Table 2. Most of the time, data transfer between host to devices and devices to host is required. Using GPUs for data processing has this drawback because data transport may take longer time than direct computing. It cannot reduce the total amount of data transferred from host to devices and devices to host. The time needed for planning before submitting the data for processing also influences calculation time. The proposed method can take longer time to plan if the data is not large enough. Planning may take longer time than sending the data for processing.

Hence, calculating only one satellite index was ineffective. If the data has already been transferred to GPU, the other satellite index should be calculated simultaneously to reduce the data transfer cost. Moreover, the first execution of the CUDA process might suffer from CUDA overhead, as demonstrated in Figure 4. Thus, executing multiple CUDA kernels over the same dataset will be more efficient.

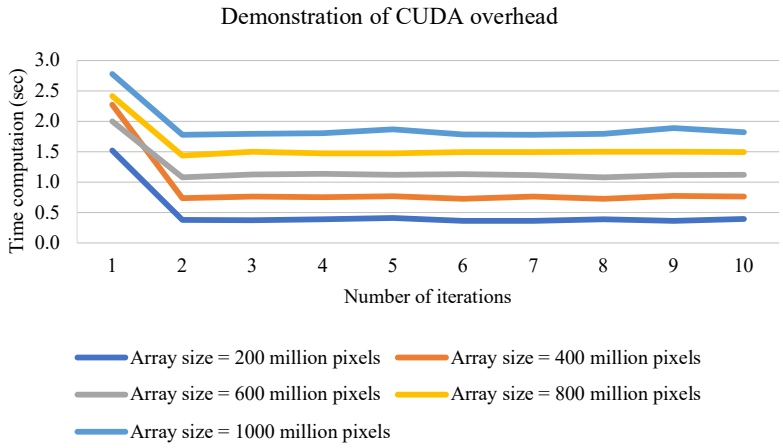

Figure 4 Demonstration of CUDA overhead.

5. Conclusion

Numba has provided the automatic configuration for GPU parallel computing. It performs very well without requiring any background knowledge of CUDA programming. However, Numba GPU computing might not support some external functions and may cause execution failure. The proposed method has the benefit of combining the traditional CUDA kernel function launch and the asynchronous GPU computing. In this way, it can reduce running time compared to other techniques. Even though normalized satellite index equations were not intensively computed and did not take any advantage of CUDA architectures like shared memory, the results demonstrated that if the data is large enough, it is still worth transferring to GPU and processing it parallelly. The complexity of mathematical computations and the not-too-small data size are suitable for computing with GPU programming. If the data size was too small, it might not be worth using this method due to overhead from both CUDA and concurrent. However, when both conditions are met, the kernel computation on GPU is up to 250 times faster than CPU process ~~or~~, especially ten times faster if include data transfer time. Hence, complicated algorithms and enormous data were suitably processed with GPU. In future work, we are planning to explore MPI. MPI will be applied to handle multiple nodes in the cluster.

6. References

[1] Copernicus Sentinel-2 (processed by ESA). MSI Level-1C TOA Reflectance Product, Collection 0 [Internet]. European Space Agency; 2021 [cited 2022 May 15]. Available from: https://doi.org/10.5270/S2_-d8we2fl.

[2] Townshend JRG, Goff TE, Tucker CJ. Multitemporal dimensionality of images of normalized difference vegetation index at continental scales. IEEE Trans Geosci Remote Sens. 1985;GE-23(6):888-895.

[3] Gao BC. NDWI—A normalized difference water index for remote sensing of vegetation liquid water from space. Remote Sens Environ. 1996;58(3):257-266.

[4] Liang S, Wang J. Chapter 10 - Leaf area index. In: Liang S, Wang J, editors. Advanced Remote Sensing. 2nd ed. London: Academic Press; 2020. p. 405-445.

[5] Huete AR. A soil-adjusted vegetation index (SAVI). Remote Sens Environ. 1988;25(3):295-309.

[6] Juan W, Jianchao S. A new type of NDVI algorithm based on GPU dividing block technology. International Conference on Computational and Information Sciences; 2013 Jun 21-23; Shiyang, China. USA: IEEE; 2013. p. 709-712.

[7] NVIDIA. CUDA toolkit, release: 10.2.89 [Internet]. NVIDIA Corporation; 2020 [cited 2022 May 15]. Available from: https://developer.nvidia.com/cuda-toolkit.

[8] Guerrouj FZ, Latif R, Saddik A. Evaluation of NDVI and NDWI parameters in CPU-GPU heterogeneous platforms based CUDA. 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech); 2020 Nov 24-26; Marrakesh, Morocco. USA: IEEE; 2020. p. 1-6.

[9] Zuo X, Qi T, Qiao B, Deng Z, Ge Q. Fast parallel extraction method of normalized vegetation index. 15th International Conference on Computer Science & Education (ICCSE); 2020 Aug 18-22; Delft, Netherlands. USA: IEEE; 2020. p. 433-437.

[10] Harris CR, Millman KJ, Van der walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature. 2020;585:357-362.
[11] Rocklin M. Dask: Parallel computation with blocked algorithms and task scheduling. Proceedings of the 14th Python in Science Conference; 2015 Jul 6-12; Austin, United States. p. 126-132.
[12] Crist J. Dask & Numba: Simple libraries for optimizing scientific python code. IEEE International Conference on Big Data (Big Data); 2016 Dec 5-8; Washington, USA. USA: IEEE; 2016. p. 2342-2343.
[13] Lam SK, Pitrou A, Seibert S. Numba: A LLVM-based python JIT compiler. Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC; 2015 Nov 15; Austin, United States. New York: ACM; 2015. p. 1-6.
[14] de Groot C. The Concurrent. Futures Library. New York: Apress; 2020.
[15] Bhangale UM, Durbha SS. Cloud detection in satellite imagery using graphics processing units. IEEE International Geoscience and Remote Sensing Symposium - IGARSS; 2013 Jul 21-26; Melbourne, Australia. USA: IEEE; 2013. p. 270-273.
[16] Zhao B, Liu M, Wu J, Liu X, Liu M, Wu L. Parallel computing for obtaining regional scale rice growth conditions based on WOFOST and satellite images. IEEE Access. 2020;8:223675-223685.
[17] Watkinson N, Tai P, Nicolau A, Veidenbaum A. NumbaSummarizer: A python library for simplified vectorization reports. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2020 May 18-22; New Orleans, USA. USA: IEEE; 2020. p. 1-7.
[18] Oden L. Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing. 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2020 Mar 11-13; Västerås, Sweden. USA: IEEE; 2020. p. 216-223.
[19] ThaiSC – NSTDA Supercomputer Center [Internet]. 2022 [cited 2022 May 15]. Available from: https://thaisc.io/en/mainpage/.
[20] Clarke L, Glendinning I, Hempel R. The MPI message passing interface standard. In: Decker KM, Rehmann RM, editors. Programming Environments for Massively Parallel Distributed Systems. Basel: Birkhäuser; 1994. p. 213-218.
[21] Sentinel Online. Sentinel-2 MSI User Guide [Internet]. 2022 [cited 2022 May 15]. Available from: https://sentinel.esa.int/web/sentinel/user-guides/sentinel-2-msi.