# Using GPU Computing to model Mars subsurface ice detection with polarimetric synthetic aperture radar

Author: Jack Hanlon[1]
Supervisors: Bill Bird[1], Étienne Boulais[2].
*[1]Department of Computer Science, University of Victoria*
*[2]Sun-Earth System Science, Space Utilization, Canadian Space Agency*

**Abstract:** A Monte Carlo method tracing the multiply scattered electric field is used to simulate the propagation of polarized light in Martian subsurface ice. This method needs to calculate a large quantity of photons to achieve an accurate result, making computation time-consuming. To accelerate the runtime of the simulation, a parallel programming approach was applied using GPGPU computing. PyCUDA was used to simplify the implementation, and to show the runtime difference in comparison with a fully CPU-based implementation. The code was tested on a Nvidia MX150 GPU and compared with the result from the Intel i5-8250U CPU. A speedup of approximately 1244x the initial unthreaded EMC algorithm was achieved.

## 1. Introduction

This project concept was derived from ongoing research at the Canadian Space Agency by Dr. Etienne Boulais in which discovering Martian subsurface ice has been deemed valuable in assessing the landing sites for future Human colonization.

Numerous studies [12],[13] have suggested that polarimetric orbital synthetic aperture radar (SAR) platforms would be capable of identifying accessible water resources necessary for the onset of Mars colonization. A preemptive understanding of these orbital SAR systems could be simulated in addition to providing tools to visualize the collected images. Recently, the CSA developed a prototype Monte-Carlo computational model that shows promising results for sufficiently describing electromagnetic wave propagation under Mars' surface. However, using conventional CPU-based parallel computing approaches, the current code performance is still prohibitive in regard of the large parameter space associated to Mars soil and large illuminated area, limiting radar image interpretation and instrument development [11].

Therefore, this project aims to explore the use of GPU-based computing to increase the computational speed of the CSA radar propagation model. Based on recent literature, similar problems have shown improvements of orders of magnitude in computational speed, which could be of great interest to the Canadian Space Agency if transferrable into their current model [11]. As a proof of concept, the Electric Monte Carlo propagation algorithm is modified to use CUDA element-wise kernel programming. Runtimes are then compared to their CPU counterparts. Lastly, an estimation on potential computational speedup, and a recommendation will be formulated that may induce action on specific hardware selection in an operational setting.

## 2. Theoretical Formalism

The lack of analytical solutions to the radiative transfer equation of polarized light in media creates a necessity for the implementation of a numerical solution – specifically using a Monte Carlo simulation [2]. This implementation of Electric Field Monte Carlo (EMC) traces the Jones vector in a homogenous isotropic non-attenuating medium [2]. “Suppose a monochromatic plane wave of light is travelling in the +z direction, with angular frequency $\omega$ and wave vector **k** = (0,0, k), where the wavenumber k = $\omega/c$ . Then the electric and magnetic fields **E** and **H** are orthogonal to **k** at each point; they both lie in the plane “transverse” to the direction of motion. Furthermore, **H** is determined from **E** by 90-degree rotation and a fixed multiplier depending on the wave impedance of the medium. So, the polarization of light can be determined by studying **E**.” [4] The complex amplitude of **E** is written as:

$$\begin{bmatrix} \boldsymbol{E_x(t)} \\ \boldsymbol{E_y(t)} \\ \boldsymbol{o} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E_{0x}e^{i(kz-\omega t+\phi_x)}} \\ \boldsymbol{E_{0y}e^{i(kz-\omega t+\phi_y)}} \\ \boldsymbol{o} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E_{0x}e^{i\phi_x}} \\ \boldsymbol{E_{0y}e^{i\phi_y}} \\ \boldsymbol{o} \end{bmatrix} \boldsymbol{e^{i(kz-\omega t)}}$$

The Jones vector is: $\begin{pmatrix} \boldsymbol{E_{0x}e^{i\phi_x}} \\ \boldsymbol{E_{0y}e^{i\phi_y}} \end{pmatrix}$ and is normalized to length 1 for describing the polarization of plane waves [4]. After each scattering, an update is applied to the Jones vector and to the local right-handed orthonormal co-ordinate system of the photon defined by (**m**, **n**, **s**), where **m**, **n**, **s** are unit vectors such that **m** is parallel to the electric field, **n** is perpendicular to the electric field (direction of magnetic field), and **s** points in the direction of photon propagation [2]. The flow of the algorithm follows the format below:

Let **E** be a 2x1 vector representing the Jones vector, and **m**, **n**, **s** be the local co-ordinate system (lcsys) of that photon in the global **x**, **y**, **z** basis [2]. Furthermore, let E_scatter_theta, E_scatter_theta_rot, E_scatter_phi, E_scatter_phi_rot be Nx1 datasets of random complex angles used to compute the EMC calculation for the input electric field **E**. Let theta and phi be real angle datasets of same length as the arrays above, used as random selections for the EMC algorithm.

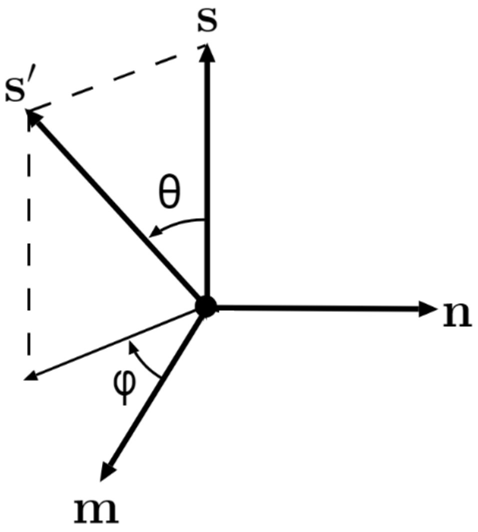

**Figure 1: Photon local co-ordinate system [2]**

Then for each scattering:

1. Calculate the total squared electric field for all EMC values by summing the multiplication of the input electric field angle datasets by their complex conjugates, multiplied by sin(theta) to account for the differential solid angle in spherical co-ordinates.

$$*** \; PRIVATE \; PSEUDOCODE \; ***$$

$$* \; Note\text{:} \; This \; equation \; is \; broadcasted \; in \; a \; numpy \; array$$

2. Compute the index of theta and phi to be used to update the electric field and local co-ordinate system. This is done by summing the total squared electric field vector and then creating another vector containing the cumulative sum of the total squared electric field. A random number R is calculated in the range (0, sum of total electric field). Next, the cumulative sum array must be searched for the smallest value that is larger than R, and that index is used as the index of theta and phi to be used in the updates to the electric field and photon lcsys.

3. Update the 2x1 electric field vector by picking E_scatter_theta, E_scatter_theta_rot, E_scatter_phi, E_scatter_phi_rot values from the datasets using the index calculated in 2 and multiplying them by the current electric field vector. Normalize this by dividing E by the square root of the sum of the square of the components of itself.

4. Next update **m**, **n,** and **s** by applying the spherical co-ordinates rotation matrix [6] based on the new calculated theta and phi values. Compute the dot product of each basis vector with each row in **A** respectively.

$$\mathbf{A} = \begin{bmatrix} cos(\theta)cos(\phi) & cos(\theta)sin(\phi) & -sin(\theta) \\ -sin(\phi) & cos(\phi) & 0 \\ sin(\theta)cos(\phi) & sin(\theta)sin(\phi) & cos(\theta) \end{bmatrix}$$

5. The new **E**, **m**, **n**, **s** is used as inputs for the next photon scattering. (Return to 1.)

This process is then completed for 1,000 scatterings per photon for 100,000 photons as a test input.

## 3. CUDA programming

Compute Unified Device Architecture is a parallel computing platform and API model created by Nvidia, with the purpose of allowing software developers to use CUDA-enabled GPUs for general purpose processing [9]. It is beneficial for computational problems that would significantly benefit from the parallelization of basic mathematical operations. The performance and functionality of CUDA is dependent on the hardware and architecture of the GPU. Limitations to floating point accuracy and runtime also are affected by the device architecture. For example, the Nvidia MX150 GPU has noticeable performance decrease when handling double precision floating point numbers, that is not realized in a more expensive model. Further limitations occur in the architecture of the GPU itself; the processors have predefined quantities of threads that they can handle.

This is important because understanding and effectively utilizing the structure and dimensions of the GPU processors will increase performance. The structure is defined by three key concepts: grids, blocks, and threads.

A Thread is a single sequential flow of control within a program [10]. This could be a mathematical operation such as an addition of an element from two arrays. Threads are organized intentionally by CUDA into blocks of dimensions x, y, and z for easy mapping into the conceptual model of many real-world problems. For example, the operation of matrix multiplication of a 2x2 matrix with a 2x1 vector can be broken down using threads into four separate threads that run in parallel, using the x and y dimensions to specify the matrix elements. Blocks have restrictions on their size in all dimensions based on the hardware of the GPU. This can be viewed by running deviceQuery.py and observing the MAX_BLOCK_DIM and MAX_GRID_DIM parameters. Once Threads are organized into blocks, blocks are then organized into a grid using the grid dimension parameters. The flow of a process then is executed via a kernel as a grid of blocks of threads [8].

The Nvidia MX150 GPU has block x, y, z dimensions of 1024, 1024 and 64, respectively. This means that any array calculation that contains more than 1024 elements will not fit in a single block in a GPU. The MAX_THREADS_PER_BLOCK variable will specify the maximum number of threads that can be run in a block on the CUDA-enabled GPU. In this case it is 1024 and so the y and z dimensions cannot be utilized within the block.

Implementation of the EMC algorithm must be adjusted accordingly by downsizing the input complex angle arrays to be of length 1024.

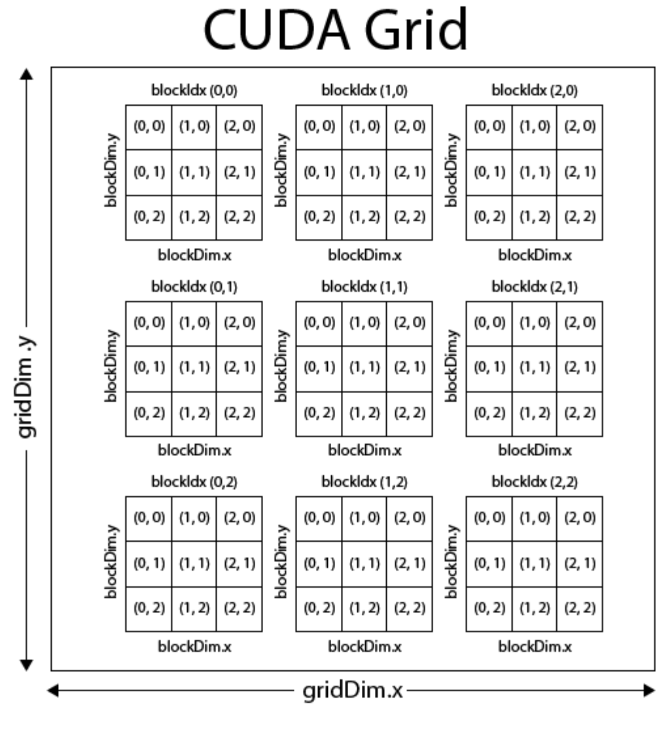

**Figure 2: GPU layout from CUDA perspective [7]**

General purpose GPU computing with CUDA is mainly written in C, which is a prohibitively difficult programming language for proof-of-concept implementations. Therefore, a Python wrapper to CUDA (PyCUDA) was created to add GPU processing functionality to NumPy arrays. NumPy arrays are converted into gpuarray's and the function "to_gpu" can be called to complete the memory allocation and copying process between the host (CPU) and device (GPU). There are built-in kernels that can be used with gpuarray’s, or customized kernels can be written in inline C code. Some examples of these built-in kernels are reduction kernels such as InclusiveScanKernel or element-wise kernels such as ElementwiseKernel. Furthermore, the Source Module function can be used to write inline C functions that work elementwise as well and run asynchronously with respect to the host code.

Gpuarray’s come with the benefit of simplicity of implementation as they are designed to mimic NumPy arrays for GPU processing, however drawbacks occur in runtime. The process of memory allocation and memory copying is completed for you with gpuarray’s, analogously to any programming language that does garbage collection itself. There is also an important caveat here on datatype for gpuarray’s as they inherit the datatype from the NumPy array you parse to them. The latest version of NumPy does not support single precision complex numbers meaning low-end GPUs have to suffer the runtime performance loss of double precision when using floating point numbers.

Copying memory to and from the host and device has the most expensive overhead for any GPU operation. This means after transferring the data to the GPU, kernels must run sequentially without doing any host memory accesses for the best runtime, often at the cost of reducing the optimality of the individual kernels themselves. Every kernel call is allocating a set of grids blocks and threads, therefore by implementing a singular kernel that completes the full algorithm, you would have optimal runtime as there would be no overhead for reallocation.

Long arrays (with a maximum length of MAX_GRID_DIM_X) can be sent to gpuarray’s by splitting up the computation into blocks of equal size. In this way, multiple photons can be simulated by initializing the maximum length array and breaking it down by dedicating a block per photon.

The Nvidia RTX 8000 GPU has 4,608 parallel processing cores as opposed to the 384 cores in the Nvidia MX150 GPU. Thus, with the same algorithm, many more photons could be ran in parallel than the MX150 GPU. It would therefore be beneficial to move the algorithm to better performing hardware, such as this top of the range GPU. Furthermore, the main bottleneck to runtime is the amount of GPU memory so an upgrade from 4096MB to 48GB would be very beneficial.

## 4. Procedure

The initial setup of the PyCUDA environment was completed by following the setup advice of “Hands-on GPU Programming with Python and CUDA”. This involved the installation and setup of the newest versions of Anaconda, CUDA, PyCUDA, and Visual Studio Community. A batch file was written (launch-python-cuda-environment.bat) that initializes the environment in a command prompt. To test that the environment was working correctly deviceQuery.py was ran to display the CUDA-enabled GPU in the workstation.

To display comprehension of matrices and complex vectors necessary to implement the EMC algorithm the pseudocode below was converted into Python in the file cpu_matrix_test.py.

```
*** PRIVATE PSEUDOCODE ***
```

**Figure 3: Initial pseudocode for testing complex matrix multiplication in Python**

To gain an understanding of PyCUDA and how it works as a wrapper to CUDA, and to prepare for creating the Electric Field Monte Carlo algorithm, gpu_matrix_test.py was implemented. This Python file uses PyCUDA and gpuarray's to complete the functionality of the pseudocode above. Matrix multiplication of random complex matrices with random complex vectors is completed using a custom CUDA kernel to increase runtime, by running each calculation elementwise on separate threads.

The Electric Field Monte Carlo algorithm was then implemented in Python in the file emc_cpu.py. This file simulates a single photon as it passes through some media, using the Monte Carlo approximation of radiative transfer, as discussed in the Theoretical Background section. Runtimes were tested and averaged for this file using various quantities of scatterings to produce a graph of change in runtime. Once this file was completed and verified to be producing the correct output, it was then modified using the Python multiprocessing library into a multi-processed version saved as emc_cpu_multi.py. This version harnessed the 8 Logical Processors (LP) in the CPU, to allow for parallel asynchronous computation of 8 photons at a time. Due to runtimes varying amongst photon calculations, the CPU is almost fully utilized for the duration of the simulation, because upon completion of a photon, the LP is tasked to start a new photon. Utilization only decreases on the last batch of photons because there are less than 8 Logical Processors in use.

Lastly, the Electric Monte Carlo algorithm was completed using General Purpose GPU computing with the PyCUDA library. This implementation is more challenging than a CPU-based approach as it involves a mixture of understanding of the hardware, the CUDA API and parallel programming through the PyCUDA wrapper on that API. Custom PyCUDA kernels were implemented to replicate the logic of emc_cpu.py, by appending the information of each consecutive photon on to a large array, and then calling the GPU kernels to split the calculations up by block on the GPU and execute the logic in parallel. The tasks in the emc_gpu.py logic were separated into 6 independent kernels that completed the following: a kernel for computing the total squared electric field; a kernel for computing the sum of the previous kernel; a kernel that multiplies the sum for each photon by a randomly generated value; a kernel that computes the index for each photon for next scattering; and then a kernel for each of the scattered electric field update and the photon local co-ordinate system update.

## 5. Results and Discussion

Initially, the algorithm discussed in section 2 was converted into a CPU-based Python implementation utilizing speedup from NumPy functions such as broadcasting, argmax, sums and cumulative sums. There was a significant difference in runtime once the original code was modified from using for loops on lists. The main bottleneck being the fact that EMC requires Nx1 computations (which in this case is 5248x1) to pick the new index (The 5248 computations due to all data arrays from scattering.csv being of length 5248).  This is done 1000 times for each of 100,000 photons, so it is evident that using a for loop here slowed down computation. The NumPy library is fast because it allows for vectorization, broadcasting and indexing which all increase performance. As shown in the code snippets below the monte_carlo function was modified and had a significant runtime improvement.

**Before (slow):**

```
        *** PRIVATE CODE ***
```

**After (fast):**

```
*** PRIVATE CODE ***
```

Initial calculations using lists showed a runtime of 49.6s per photon, and once NumPy was used in this calculation the runtime became 1.28s per photon. A significant improvement that dropped the overall runtime for 100,000 photons from 57.4 days to 35.6 hours. For EMC to be a valid method for the simulation of subsurface ice detection, 35 hours is still a prohibitive time to compute the necessary data (which would be multiple scales larger than the test size). All time measurements completed in this paper were done using the Python time library.

Parallel computing such as multi-threading and multi-processing were then considered as effective methods to reduce this runtime. Multi-processing was implemented on CPU, using the 8 logical processors available on the Intel i5-8250U CPU. The multiprocessing pool library was used in this implementation. This enabled the CPU to asynchronously calculate 8 photons in parallel when the processers are fully saturated. In this case, runtime varied based on the number of photons tested as computational resources are split between the number of running processes. Thus, when the CPU is not saturated the runtime of photons was faster. The multi-processed implementation was ran on the test size of 100,000 photons and showed a speedup of 2.1x over the non-parallelized CPU implementation. The test took 16.68 hours to complete, with an average of 0.60s per photon. This was a significant improvement over the previous implementation but a factor of 2 improvement is still prohibitive to the technology being functional.

| | **CPU-based EMC runtimes using Lists** | **CPU-based EMC runtimes using NumPy** | **Multi-processed CPU-based EMC runtimes** | **PyCUDA GPU-based EMC runtime** | **PyCUDA with Nvidia RTX 3080** |
|---|---|---|---|---|---|
| **Average time to generate a photon** | 49.6 s / photon | 1.28 s / photon | 0.60 s/photon | 0.00103 s/photon | 0.0000692 s/photon |
| **Time taken to generate 100,000 photons** | 57.41 days | 35.56 hours | 16.68 hours | 1.72 minutes | 6.92 seconds |

**Figure 4: Runtimes for 100,000 photons scattered 1000 times each**

Therefore, CUDA was used to highly-parallelize the photon calculations by transferring the algorithm to PyCUDA kernels for larger quantities of photons at once. The implementation is limited by the hardware of the system and so an implementation on a more powerful GPU was also discussed. Firstly, the implementation method has many factors to consider when attempting to optimize runtime performance, such as memory allocation and transfer, single vs double precision and datatype conversions, bandwidth, synchronization, caching, and many other considerations. To effectively improve runtime of the algorithm, the bottlenecks can be ran in elementwise kernels, reduction kernels and inclusive scan kernels depending on situation. However, sequential kernel calls have a negative effect on runtime as memory copying is expensive. Custom kernels are more challenging to implement, rather than using built in PyCUDA gpuarray functions, but do show significant performance increases.

This implementation of GPU based EMC is not optimal but shows a proof of concept in the application of effective GPU based speedup. Optimality should hypothetically occur when the 6 kernel launches are reduced into 1 singular kernel containing the logic of the entire EMC algorithm including the scattering loop. Each kernel call has an overhead due to PyCUDA, which grows as a function of the memory allocation completed in the call. Therefore, the less calls the more efficient the algorithm. This was not possible with the current build, because despite reducing the number of kernels from 12 to 6, the remaining kernels call different amount of blocks/grid and threads/block and so the computations would be incorrect if merged.

The current CUDA-based implementation completes 1000 scatterings for up to 22432 photons due to the device memory limitation. The Nvidia RTX 8000 GPU would sufficiently handle this bottleneck and allows for significantly more photons to be run in parallel at once. The array length of 1024 is selected as block dimensions with multiples of 32 are maximally optimized according to the CUDA toolkit documentation.

The arrays are then allocated memory and copied to the GPU via the gpuarray “to_gpu” function. Element-wise calculation of each element in the arrays are done with a thread each and are completed asynchronously with respect to the CPU code. The __syncthreads() command is used to prevent threads from overlapping between operations within a kernel, allowing the calculation of each photon to not be corrupted by asynchronous memory writes. Built-in gpuarray functions such as gpuarray.sum(), for summing the components of E_totsq, and scan.InclusiveScanKernel(), for cumulative summing E_totsq, were originally tested to complete the remainder of the algorithm. PyCUDA functions had significant performance decreases relative to the inline C, so the PyCUDA functions were discontinued, and custom C implementations of each kernel were created. Typical runtime of the initial total squared electric field value (E_totsq) was approximately 0.001 seconds, versus the gpuarray.sum() function takes approximately 3.7 seconds. Bottlenecks such as these show that using the gpuarray functions are not effective for runtime considerations and custom implemented inline C kernels would perform significantly better. Furthermore, NumPy does not support single precision complex numbers, and low-end GPUs are prone to low performance in double precision, meaning runtime is also slowed due to these constraints. Until a newer version of NumPy is released supporting complex32, it would be beneficial to purchase a premium GPU for the purpose of computing complex numbers. A more detailed explanation of how the CUDA architecture is utilized for each kernel is shown in supporting documents [14].

The current qualitative perspective on GPU computing for the Electric field Monte Carlo algorithm, is that it is a promising method to using parallel computing to reduce the runtime of the program and make orbital synthetic aperture radar subsurface ice detection a realizable scientific method [11]. However, the simulation written using PyCUDA and NumPy displays significant red flags in the usage of wrappers and libraries to low-level languages. An even more efficient implementation could be completed, using the C CUDA interface to remove these issues in an operational setting.

Speedup of 1244x the initial unthreaded EMC algorithm was achieved showing that massive parallelism using GPU's is a reliable choice for making orbital polar SAR a useful technology. This speedup also decreased the number of seconds needed to compute a thousand scatterings of a single photon from 0.60s/ph to 0.001s/ph, from the previously most optimal implementation.

This technology is promising for the mapping of Martian subsurface ice and with refinement will be operationally ready for such a task. The combined initiative of NASA and three international partners (CSA, ASI, JAXA), intend on a tentative launch date as early as 2026 for this concept [15].

## 6. Conclusion

To conclude, a Monte Carlo method to trace the multiply scattered electric field used to simulate photon propagation in Martian subsurface ice was completed successfully with three methods. Firstly, the algorithm was implemented as a CPU-based approach and showed a prohibitive runtime for large parameter space. It was then re-written using the Pool multiprocessing Python library, increasing the runtime performance by over double. However, the runtimes were still prohibitive so a GPGPU approach was completed. This approach showed the significant difficulty involved in computational parallelism using GPUs and showed that the usage of libraries and higher abstractions becomes less optimal when the main performance bottleneck is no longer computation, but instead memory operations. GPU-based computing yields a substantially faster EMC algorithm with an order of magnitude of 1244x or more. In conclusion, the project objective was completed successfully, and more GPU refinement will lead to even better results.

## 7. Bibliography

[1] Y. Wang, P. Li, C. Jiang, J. Wang, and Q. Luo, "GPU accelerated electric field Monte Carlo simulation of light propagation in turbid media using a finite-size beam model," *Optics Express*, vol. 20, no. 15, p. 16618, 2012.

[2] M. Xu, "Electric field Monte Carlo simulation of polarized light propagation in turbid media," *Optics Express*, vol. 12, no. 26, p. 6530, 2004.

[3] T. Young-Schultz, S. Brown, L. Lilge, and V. Betz, "FullMonteCUDA: a fast, flexible, and accurate GPU-accelerated Monte Carlo simulator for light propagation in turbid media," *Biomedical Optics Express*, vol. 10, no. 9, p. 4711, 2019.

[4] “Jones calculus,” *Wikipedia*, 24-Mar-2021. [Online]. Available: https://en.wikipedia.org/wiki/Jones_calculus. [Accessed: 29-Apr-2021].

[5] B. Tuomanen, *Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA*. Birmingham, Mumbai: PACKT Publishing Limited, 2018.

[6] “Spherical coordinate system,” *Wikipedia*, 21-Apr-2021. [Online]. Available: https://en.wikipedia.org/wiki/Spherical_coordinate_system. [Accessed: 29-Apr-2021].

[7] “CUDA Parallel Thread Management,” *Microway*, 30-Aug-2013. [Online]. Available: https://www.microway.com/hpc-tech-tips/cuda-parallel-thread-management/. [Accessed: 29-Apr-2021].

[8] “CUDA Refresher: The CUDA Programming Model,” *NVIDIA Developer Blog*, 25-Aug-2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/#:~:text=A%20group%20of%20threads%20is,%2C%20or%20CUDA%20dynamic%20parallelism). [Accessed: 29-Apr-2021].

[9] “CUDA,” *Wikipedia*, 20-Apr-2021. [Online]. Available: https://en.wikipedia.org/wiki/CUDA. [Accessed: 29-Apr-2021].

[10] “Thread (computing),” *Wikipedia*, 26-Apr-2021. [Online]. Available: https://en.wikipedia.org/wiki/Thread_%28computing%29. [Accessed: 29-Apr-2021].

[11] *jacklhanlon@yahoo.co.uk*. "Project_description_Jack_Hanlon.docx"

[12] B. A. Campbell, “Mars orbital synthetic aperture radar: Obtaining geologic information from radar polarimetry,” *Journal of Geophysical Research*, vol. 109, no. E7, 2004.

[13] S. M. Clifford, “Introduction to the special section: Geophysical detection of subsurface water on Mars,” *Journal of Geophysical Research*, vol. 108, no. E4, 2003.

[14] “EMC GPU IMPLEMENTATION/docs/Requirement Specification Document”

[15] T. Talbert, “NASA, International Partners Assess Mission to Map Ice on Mars,” *NASA*, 03-Feb-2021. [Online]. Available: https://www.nasa.gov/feature/nasa-international-partners-assess-mission-to-map-ice-on-mars-guide-science-priorities. [Accessed: 02-Jun-2021].