Requirement Specification Document
EMC GPU Implementation

University of Victoria, Canadian Space Agency

June 3, 2021

Jack Hanlon, Etienne Boulais

Contents

Revision History

Name	Date	Reason for Changes	Version
emc_cpu.py	March 18, 2021	Build a working EMC algorithm	0.0.1
emc_cpu_multi.py	March 21,2021	An EMC algorithm that uses the Python pool library to improve runtime 2x	0.0.2
emc_gpu.py	April 30, 2021	Initial CUDA accelerated EMC algorithm, could only run 2 photons in parallel. Submitted to the University of Victoria as a final project.	0.0.3
emc_gpu.py	May 27, 2021	Working massively parallelized EMC algorithm completing a 1244x speedup over version 0.0.1	1.0.0

1 Introduction

1.1 Purpose

The Electric field Monte Carlo GPU-based implementation is a model for detecting subsurface ice on Mars, using polarimetric synthetic aperture radar. It uses the PyCUDA wrapper to access CUDA cores of the designated Nvidia GPU, to hardware accelerate the algorithm through massive parallelism. This Requirements Specification Document (RSD) describes the intended purpose, requirements and nature of the software developed, intended to satisfy the desired pseudocode.

1.2 Project Scope

Emc_gpu.py was intended as an experimental algorithm, as a subset of other software in this area created by the Canadian Space Agency. The main feature is to take input scattering data, compute a Monte Carlo update to the Jones vector of a scattered photon in media, and calculate updates to the local co-ordinate system of the photon. This is a subprocess of the overarching principle of

polarimetric synthetic aperture radar. Generating many photons, for the large parameter space of Mars' surface, is necessary and only technologically viable through runtime speedup.

2 Overall Description

2.1 Product Features

The main feature of the emc_gpu.py file is the six PyCUDA SourceModule kernels enclosed in the scattering loop. Each of these six kernels compute components of the EMC algorithm consecutively. There also is logic that downsizes the CSV data from 5248 elements to 1024, to maintain optimality with CUDA architecture and to allow for photons to be separated by blocks.

2.2 Operating Environment

The technology stack that the emc_gpu.py algorithm was implemented in is documented in the textbook: "Hands on GPU programming with Python and CUDA", by Dr. Brian Tuomanen. This stack was built on a Windows 64-bit operating system. The tech stack includes latest versions of:

Anaconda 3
Nvidia CUDA:
Development 11.2
Development 11.3
Documentation 11.2
Documentation 11.3
NSight NVTX 11.2
NSight NVTX 11.3
Runtime 11.2
Runtime 11.3
Samples 11.2
Samples 11.3
Visual Studio Integration 11.2
Visual Studio Integration 11.3

PyCUDA (Latest)
Visual Studio 2019

Output is sent to the terminal upon calling the command:

python emc_gpu.py

Output and corresponding compute profiling is sent to a text file of choice using the command:

python -m cProfile -s cumtime emc_gpu.py > file_name.txt

2.3 Design and implementation constraints

The design is constrained by the CSV data being of finite size, therefore the possible scattering angles is a discrete list and not a continuous function. Secondly, the size of the CSV data had to be downsized to build a more efficient algorithm. This means there is some loss in quality of the accuracy of the photon scattering calculation. Other constraints pertain to the way random numbers are generated in the code. They are sampled from a uniform distribution between 0 and 1. Next, these values are stored in an array equal in length to the length of the array of sums. Upon each scattering, each element of the random number array is multiplied by each respective element of the array of sums. A scattering bit and a modulus calculation are then used to iterate through the elements of the random number array for each scattering. This is an implementation constraint as the random number choices selected have a period and thus is pseudo-random.

3 GPUARRAYS

3.1 Array Function

NAME	FUNCTION
E_totsq_gpu	Holds the calculation for the total squared electric field
float_E_totsq_gpu	Converted dtype to float for gpuarray E_totsq_gpu
cum_sum_E_totsq_gpu	Duplicate of float_E_totsq_gpu except designed for running_sum Kernel
photon_m_gpu	Photon local co-ordinate system m basis vector for all photons, ready for updates.

photon_n_gpu	Photon local co-ordinate system n basis vector for all photons, ready for updates.
photon_s_gpu	Photon local co-ordinate system s basis vector for all photons, ready for updates.
mprime_lc_gpu	m basis rotation matrix update to be applied to photon_m lcsys
nprime_lc_gpu	n basis rotation matrix update to be applied to photon_n lcsys
sprime_lc_gpu	s basis rotation matrix update to be applied to photon_s lcsys
tmp_m_gpu	Update to m component of photon lcsys for each photon
tmp_n_gpu	Update to n component of photon lcsys for each photon
tmp_s_gpu	Update to s component of photon lcsys for each photon
E_gpu	Contains the Jones vector for each photon, ready for updates by EMC
E_scatter_phi_gpu	Contains the contents of E_scatter_phi array from CSV file, tiled to the number of photons
E_scatter_phi_rot_gpu	Contains the contents of E_scatter_phi_rot array from CSV file, tiled to the number of photons
E_scatter_theta_rot_gpu	Contains the contents of E_scatter_theta_rot array from CSV file, tiled to the number of photons
E_scatter_theta_gpu	Contains the contents of E_scatter_theta array from CSV file, tiled to the number of photons
phi_gpu	Contains the contents of phi array from CSV file, tiled to the number of photons
theta_gpu	Contains the contents of theta array from CSV file, tiled to the number of photons

R_gpu	Array of random numbers generated using a uniform distribution
out_gpu	Contains the multiplication of the R_gpu gpuarray and the float_E_totsq_gpu gpuarray
out_gpu_reduced	Removes unnecessary data points from the out_gpu array by taking only every 1024^{th} element
run_sum_gpu	Contains a running sum of the cum_sum_E_totsq_gpu gpuarray for each consecutive thread for each block as to calculate the INDEX for next scattering.
INDEX_gpu	Array of indices for each photon for next update to E_gpu and photon_m,n,s_gpu
EE1_gpu	Component of Jones vector update for second element of each photons' Jones vector
EE2_gpu	Component of Jones vector update for first element of each photon's Jones vector
nE_gpu	Jones vector element updates
Norm_gpu	Normalization value for each photon lcsys update
float_theta_gpu	Passes theta_gpu copy as a float32 gpuarray instead of complex64
float_phi_gpu	Passes a phi_gpu copy as a float32 gpuarray instead of complex64

3.2 Array Information

Name	Shape (Nx1)	Dtype
E_totsq_gpu	1024 * num_photons	complex64
float_E_totsq_gpu	1024 * num_photons	float32

cum_sum_E_totsq_gpu	1024 * num_photons	float32
photon_m_gpu	3 * num_photons	float32
photon_n_gpu	3 * num_photons	float32
photon_s_gpu	3 * num_photons	float32
mprime_lc_gpu	3 * num_photons	float32
nprime_lc_gpu	3 * num_photons	float32
sprime_lc_gpu	3 * num_photons	float32
tmp_m_gpu	3 * num_photons	float32
tmp_n_gpu	3 * num_photons	float32
tmp_s_gpu	3 * num_photons	float32
E_gpu	2 * num_photons	complex64
E_scatter_phi_gpu	1024 * num_photons	complex64
E_scatter_phi_rot_gpu	1024 * num_photons	complex64
E_scatter_theta_rot_gpu	1024 * num_photons	complex64
E_scatter_theta_gpu	1024 * num_photons	complex64
phi_gpu	1024 * num_photons	complex64
theta_gpu	1024 * num_photons	complex64
R_gpu	1024* num_photons	float32
out_gpu	1024 * num_photons	float32
out_gpu_reduced	num_photons	float32
run_sum_gpu	num_photons	float32
INDEX_gpu	num_photons	float32
EE1_gpu	2 * num_photons	complex64
EE2_gpu	2 * num_photons	complex64

nE_gpu	2 * num_photons	complex64
Norm_gpu	2 * num_photons	complex64
float_theta_gpu	1024 * num_photons	float32
float_phi_gpu	1024 * num_photons	float32

3.3 Array Use Case

Name	Kernel	Blocks/Grid	Threads/Block
E_totsq_gpu	monte_carlo	num_photons	1024
float_E_totsq_gpu	sum_array, mult_random	Ceiling(num_photons/2), num_photons	1024,1024
cum_sum_E_totsq_gpu	running_sum	num_photons	1
photon_m_gpu	photon_lcsys_update	num_photons	3
photon_n_gpu	photon_lcsys_update	num_photons	3
photon_s_gpu	photon_lcsys_update	num_photons	3
mprime_lc_gpu	photon_lcsys_update	num_photons	3
nprime_lc_gpu	photon_lcsys_update	num_photons	3
sprime_lc_gpu	photon_lcsys_update	num_photons	3
tmp_m_gpu	photon_lcsys_update	num_photons	3
tmp_n_gpu	photon_lcsys_update	num_photons	3
tmp_s_gpu	photon_lcsys_update	num_photons	3
E_gpu	monte_carlo, E_next_update	num_photons, num_photons	1024, 2
E_scatter_phi_gpu	monte_carlo	num_photons	1024
E_scatter_phi_rot_gpu	monte_carlo	num_photons	1024
E_scatter_theta_rot_gpu	monte_carlo	num_photons	1024

E_scatter_theta_gpu	monte_carlo	num_photons	1024
phi_gpu	monte_carlo	num_photons	1024
theta_gpu	monte_carlo	num_photons	1024
R_gpu	mult_random	num_photons	1024
out_gpu	mult_random	num_photons	1024
out_gpu_reduced	mult_random, running_sum	num_photons, num_photons	1024, 1
run_sum_gpu	running_sum	num_photons	1
INDEX_gpu	running_sum, E_next_update, photon_lcsys_update	num_photons, num_photons, num_photons	1,2,3
EE1_gpu	E_next_update	num_photons	2
EE2_gpu	E_next_update	num_photons	2
nE_gpu	E_next_update	num_photons	2
Norm_gpu	E_next_update	num_photons	2
float_theta_gpu	photon_lcsys_update	num_photons	3
float_phi_gpu	photon_lcsys_update	num_photons	3

4 Kernels

4.1 Kernel Logic

monte_carlo:

Input Arrays:

E_scatter_phi_gpu, E_scatter_phi_rot_gpu, E_scatter_theta_rot_gpu, E_scatter_theta_gpu, theta_gpu, E_gpu, E_totsq_gpu, float_E_totsq_gpu, cum_sum_E_totsq_gpu.

Modified Arrays:

E_totsq_gpu, float_E_totsq_gpu, cum_sum_E_totsq_gpu

Logic:

Firstly, the E_totsq_gpu is set to zero to make sure that the data from the previous scattering does not interfere with the current calculation. Then, the CSV data is taken in and the total squared electric field value for each photon is computed in independent threads for each element. Next, the E_totsq_gpu is copied into the float_E_totsq_gpu gpuarray, which has the dtype float32 instead of complex64. After that, values returned in scientific notation are set to zero to save computational time, and numbers are rounded to 5 decimal places. Lastly, the float_E_totsq_gpu array is copied into cum_sum_E_totsq_gpu for use in separate kernels.

sum_array:

Input Arrays:

float_E_totsq_gpu, and integer: n = num_photons*PHOTON_LENGTH (PHOTON_LENGTH = 1024)

Modified Arrays:

float_E_totsq_gpu

Logic:

Firstly, tid is initialized to the threadIdx.x in the x direction and an offset is initialized to 2*blockIDx.x * blockDim.x in the x direction. Next, an efficient divide and conquer algorithm for summation is used to sum the photons so that every 1024^{th} element contains the sum of each half block (all calculated E_totsq values for each photon). The nature of the algorithm leaves many elements in the gpuarray as temporary values; therefore, it is important here to downsize the gpuarray to just the sums for each photon. That task is completed in the mult_random kernel.

mult_random:

Input Arrays:

float_E_totsq_gpu, R_gpu, integer n = num_photons*PHOTON_LENGTH, integer scatter_bit, out_gpu, out_gpu_reduced

Modified Arrays:

out_gpu, out_gpu_reduced

Logic:

A thread variable i is initialized to blockIdx.x * blockDim.x + threadIdx.x in the x direction. Then, the out_gpu and out_gpu_reduced gpuarrays are set to zero to remove data from previous

scattering. The out_gpu array is then filled elementwise with the multiplication of the float_E_totsq elements and the R_gpu elements, such that on each scattering the R_gpu array elements are permuted once. Lastly, the relevant data points (every 1024^{th} element) are copied from out_gpu into out_gpu_reduced.

running_sum:

Input Arrays:

cum_sum_E_totsq_gpu, INDEX_gpu, out_gpu_reduced, integer: num_photons, integer: PHOTON_LENGTH, run_sum_gpu

Modified Arrays:

run_sum_gpu, INDEX_gpu

Logic:

A thread variable i is initialized to blockIdx.x * blockDim.x + threadIdx.x in the x direction. Next, the previous run_sum_gpu gpuarray and INDEX_gpu gpuarray are emptied for new scattering. An algorithm here takes each block in parallel and sums through the elements until the run_sum value for that photon is larger than the R_gpu value. It then sets that element of the INDEX_gpu file equal to the index that contains the run_sum value one element larger than the R_gpu element.

E_next_update:

Input Arrays:

INDEX_gpu, E_scatter_phi_gpu, E_scatter_phi_rot_gpu, E_scatter_theta_rot_gpu, E_scatter_theta_gpu, EE1_gpu,EE2_gpu, nE_gpu, Norm_gpu, E_gpu

Modified Arrays:

EE1_gpu, EE2_gpu, nE_gpu, Norm_gpu, E_gpu

Logic:

A thread variable i is initialized to blockIdx.x * blockDim.x + threadIdx.x in the x direction. The first line of E_next_update from emc_cpu.py is completed in parallel for all photons. To prevent unnecessary threads from running and overlapping photons the logic is applied to a modulus of the length of the Jones vector (2). Next, the second line of E_next_update from emc_cpu.py is completed in parallel for all photons in the exact same manner. The E_gpu gpuarray is then filled with the update. Lastly, the normalization for each photon is calculated in parallel using the same modulus logic as previously, and then the E_gpu gpuarray is updated in parallel by dividing by the Norm for each photon.

photon_lcsys_update:

Input Arrays:

INDEX_gpu, float_theta_gpu, float_phi_gpu, photon_m_gpu, photon_n_gpu, photon_s_gpu, mprime_lc_gpu, nprime_lc_gpu, sprime_lc_gpu, tmp_m_gpu, tmp_n_gpu, tmp_s_gpu,

Modified Arrays:

mprime_lc_gpu, nprime_lc_gpu,sprime_lc_gpu, tmp_m_gpu,tmp_n_gpu, tmp_s_gpu, photon_m_gpu,photon_n_gpu,photon_s_gpu

Logic:

A thread variable i is initialized to blockIdx.x * blockDim.x + threadIdx.x in the x direction. Only every third thread is used in the logic of the computation of the rotation matrix using the INDEX_gpu gpuarray. This is to prevent photon information overlapping between blocks. Mprime_lc_gpu, nprime_lc_gpu, and sprime_lc_gpu are filled with the respective components of the rotation matrix in parallel for each photon. Any values returned in scientific notation – whether negative or positive – are then set to zero to increase computational efficiency. Then a logical computation of the matrix transpose and multiplication of the rotation matrix elements with the photon local co-ordinate system is computed in parallel for each photon. These temporary gpuarray's are then copied into the three basis vectors of the photon local co-ordinate system for each photon in parallel.

4.2 Kernel CUDA Layout

Name	Blocks/Grid	Threads/Block
monte_carlo	num_photons	1024
sum_array	Ceiling(num_photons/2)	1024
mult_random	num_photons	1024
running_sum	num_photons	1
E_next_update	num_photons	2
photon_lcsys_update	num_photons	3

5 Diagrams

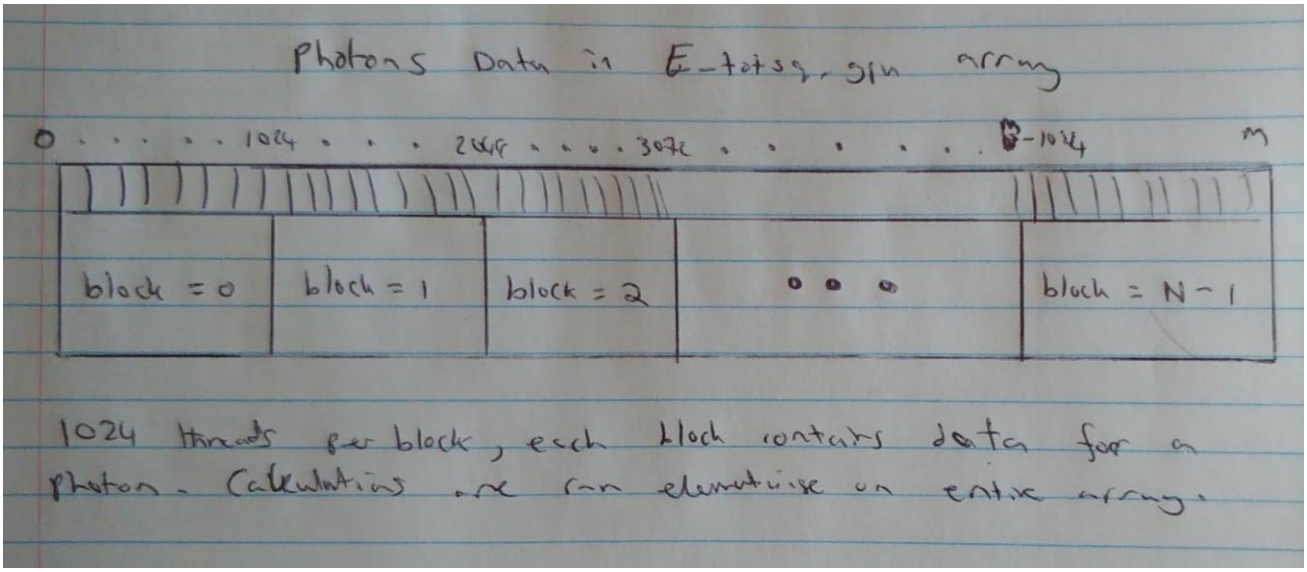

Figure 1: Layout of E_totsq_gpu in CUDA

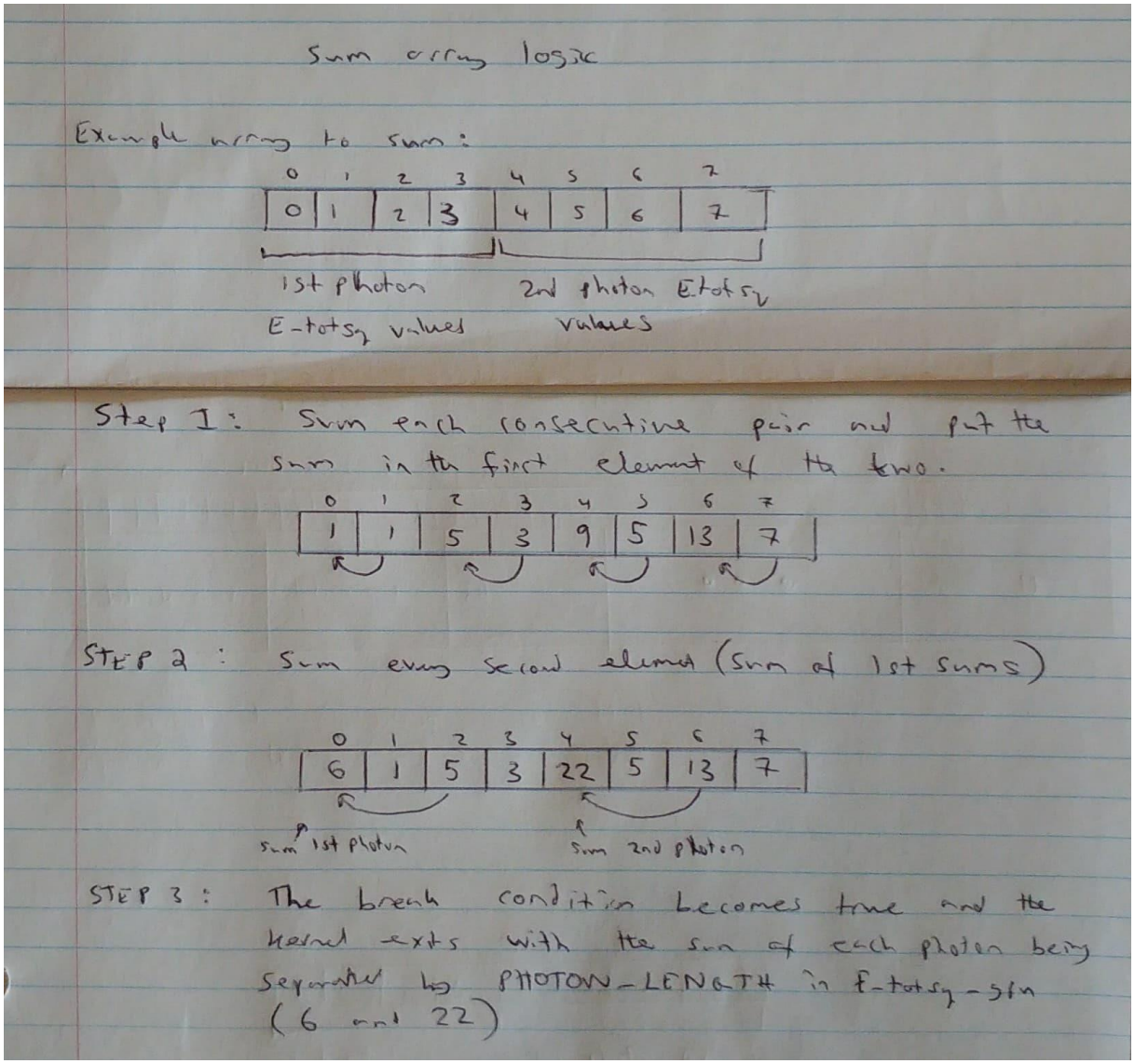

Figure 2: How the sum_array kernel Divide and Conquer algorithm works

6 Modifications

6.1 Requirements for use on Nvidia Quadro RTX 8000 GPU

The Nvidia Quadro RTX 8000 GPU has 4608 CUDA cores (relative to the 384 CUDA cores on the Nvidia mx150), meaning the algorithm may be modified to support the full 5248 elements of the CSV data arrays. This can be done by making a photon into 6 blocks rather than 1, although efficiency losses will occur here due to large components of the last block being empty. The value 1024 is hard coded in some of the kernels in emc_gpu.py so that value would need to be modified if this change were to be made. Most kernels would need modification to fit this change as well. The current emc_gpu.py could be ported directly to a better GPU, and the bottleneck of memory will be reduced, meaning more photons could be generated at once. The current code running on

the mx150 GPU can compute at the rate of 971.53 photons/second, but with a direct increase in number of photons in the array, there will be no change in runtime and only an increase in number of photons generated.