GEANT-4 GPU Port:

Test Plan

Stuart Douglas – dougls2
Matthew Pagnan – pagnanmm
Rob Gorrie – gorrierw
Victor Reginato – reginavp

Version 0
October 29, 2015

Contents

Revision History

All major edits to this document will be recorded in the table below.

Table 1: Revision History

Description of Changes	Author	Date
Initial draft of document	Stuart, Matthew, Rob, Victor	2015-10-26

List of Tables

1 General Information

1.1 Summary

GEANT4-GPU will be a library for GEANT4 that will make use of the GPU's processing power to reduce the computation times of GEANT4 programs while still outputting the same results as if GEANT4 was run without the GPU code. This testing plan outlines how the development team is expected to test these two aspects with the use of unit testing, system testing, structural testing and code testing.

1.2 Risks

The following table outlines the major risks associated with the testing of the product. A more detailed analysis of each of the risks follows the table.

Table 2: Risks

Risk #	Summary	Severity
1	differing order of random numbers on GPU could lead to difficulty comparing results with simulations run on CPU	Very High
2	isolating GEANT4 methods to test with unit tests may be too difficult	High
3	running time of tests will be too long to run them frequently	High

Risk 1 – Random Numbers:
The GEANT4 project is heavily dependent on random numbers. Random numbers are used to determine attributes about particles (indepedent of all other particles) as they move through the system. By parallelizing the workload, the order in which the particles are evaluated may change, causing it to draw a different random number from the sequence, leading to different results. If it is decided to generate the random numbers using GPU API calls, then they will certainly be different than in the serial, CPU version of the software.

Two solutions to this problem have been discussed. The first is to develop a solution to pass the CPU-generated numbers to the GPU while ensuring they remain in sync to the order they would be received in the serial version. This would require extra development work, but would ensure that testing the product against the existing system can be done in a straight-forward manner. The alternative is to test the software based on degree of similarity between single points of summarizing data (e.g. a mean). Comparing the products in this manner would have us check that the results are within a given margin of error, and if so the product would be deemed correct. This is the solution we are planning on implementing at this point, however if it does not seem feasible during the design and development phases we are open to exploring solution one.

Risk 2 – Isoating Functions:
The codebase of GEANT4 is large and complex, a potential testing risk would be that the functions to be parallelized are not well-enough isolated to easily test inputs and outputs. The objects used in GEANT4's calculations are large and often dependent on other objects. To mitigate this testing risk, development will focus on porting single functions in classes to be tested, and unit testing will test those specific functions.

Risk 3 – Test Runtime
Running a full system test on the modified code is a computationally intensive procedure. Even with a relatively small number of particles the computation could take several hours. With a larger, more realistic number of particles, running the program could take on the order of days. This would prevent frequent system testing, and the team would have to rely on unit tests during development. To address this risk, we will use relatively simple examples for most system tests (i.e. with few particles) and we will run those tests regularly along with unit tests. After large changes, more complex system tests will be run to ensure that the correctness has not been affected when using larger, more realistic examples.

1.3 Constraints

1.4 Definitions and Acronyms

2 Test Types

2.1 Manual and Automatic Testing

Manual testing needs to be conducted by people, where test cases and inspections are manually performed. On the other hand, automated testing relies less on people and performs tests quickly and effectively returning feedback on results not meeting expectations. Manual testing is more flexible; However, it is far more time consuming.

Our system testing will be done automatically – a small script will be written to run the example and check the output file against an output file from the existing product. Unit tests will also be automated, by passing in specific inputs to modified functions and checking that ouput against the correct output from the existing product.

2.2 System Testing

There are several examples that come with GEANT4 as well as some created by McMaster's Engineering Physics department for G4-STORK. Both examples in G4-STORK and GEANT4 will be used for system testing. Due to the compuation time for some of the exmaples, they will be run less frequently than the simpler ones. This will let us run system tests regularly on the simple files, while checking that more realistic examples are also correct after bigger changes.

2.3 Structural Testing (Stress Testing)

Structural tests focus on the program's internal structure by evaluating the structure and the non-functional requirements of the system, particularly on any abnormal or extreme behavior.

We will have several tests where there will be an extremely high number of particles to process. We will test to see how this affects the computation time compared to the non-GPU code. We will also check to ensure that the large number of particles does not crash the system when it doesn't on the non-GPU code.

2.4 Unit Testing

Unit tests will be created throughout development as functions and classes are ported to the GPU. Before porting a specific function of a class, a variety of inputs covering

all edge cases for the function (i.e. input is 0, negative, very large, very small) as well as several "normal" cases will be created. The results of the function for each of the inputs will be recorded, and a test case will be added to test whether the outputs are the same with GPU computation enabled (once it's implemented). Due to the nature of this testing strategy, explicit unit tests are not currently known, as no functions have yet been ported to the GPU. We have identified the following functions that we are likely to port to the GPU, and explain how each unit test will work for them.

2.5 Code Testing

3 Testing Factors

3.1 Factors to be Tested

3.2 Description of Factor

3.2.1 Correctness

Tests run on the GPU-enabled product should provide the same resulting output as tests run on the existing product

Rationale
In order for the product to be of use it must be able to compute the correct outputs.

Methods of testing

- The output files of tests run on the product will be compared with the output files of tests run on the non-GPU existing product

3.2.2 Performance

Tests run with the GPU functionality enabled should take less time to compute compared to the same tests run on the the existing CPU-based product

Rationale
The focus of this project is to reduce the computation time for G4-STORK by parallelizing functions on a GPU.

Methods of testing

- Compare the computation times of tests run on the product with GPU enabled with the computation times of test run on the existing product to see if the GPU-based computation times are smaller.
- Stress test the product by having it compute programs that would take an infeasible amount of time to compute on the serial, CPU-bound existing product.

4 Test Items

4.1 Requirements Testing

Testing requirements refers to tests which verify that documented requirements are met. The requirements documentation includes a variety of non-functional and functional requirements to be tested. The unit and system tests that will be run frequently will be aimed at testing the functional requirements related to the correctness of the system, as well as the non-functional requirements related to performance. At the end of development of Revision 0, other requirements will be tested. The results from those evaluations will lead to decisions made for the final product.

4.2 User Manual Testing

4.3 Error Handling Testing

Error handling indicates the extent to which users can recover from or know the basis of an error occurring in the system. Error handling would imply trying to find ways of committing errors, such as passing in invalid input or function arguments. For examples, if users try to call one of the new parallelized functions and provide it with the wrong types of data, they must be clearly shown the reason for their failure and, if possible, be presented with a solution. Error handling would be performed both manually and in an automated setting, once a set of potential errors are discussed and figured out amongst the members of the testing team.

5 Automated Testing Plans

System tests and unit testing will both be automated, as they will be run frequently throughout the development process.

System Testing
A small script will be created to run the chosen example file and compare the output file with the correct output file from the existing product. If the results are the same within a margin of error elicited from consultation with the stakeholders, then the test is passed. Although the test itself will be automated, we will manually run the test. This will let us try out different example files at different times, and to choose more complex ones when larger changes are made. The output files contain run-time data which we will use to evaluate performance.

Unit Testing

A testing class will be added to the project which will contain all unit tests. These will be created as we port functions to the GPU by deciding on several inputs and recording

the results as well as the running time from the existing CPU-bound product. The testing class will then call the ported function with the same inputs and verify that the outputs are the same. The testing function will record performance while it runs to ensure that the changes are positive from a performance standpoint. If the results for all functions in the testing class match the expected results, the unit tests have passed. We will manually run the unit tests throughout development as the developer sees fit. As well as those frequent tests, all unit tests must pass before any code is merged into the repository. Changes that cause a performance regression will be allowed, under the assumption that they will be improved upon until their performance is better than the existing non-GPU functions.

6 Proof of Concept Test

7 Schedule

7.1 Testing Schedule

7.2 Deliverables

Table 3: Unit Testing – *G4NeutronHPVector.hh::Times(G4double factor)*

#	**Initial State**	**Inputs**	**Outputs**	**Description**
1	object represents vector (1,2,...,512)	factor: 0	no output, vector object now represents vector (0,0,...,0) of length 512	multiplying non-zero vector object by 0 should change the object's state to a zero-vector of the same length
2	object represents vector (1,2,...,512)	factor: -1	no output, vector object now represents vector (-1,-2,...,-512)	multiplying non-zero vector object by -1 should change the sign of all elements
3	object represents vector (1,2,...,512)	factor: 1	no output, vector object still represents vector (1,2,...,512)	multiplying non-zero vector object by 1 should not change the object's state
4	object represents vector (1,2,...,512)	factor: 4	no output, vector object still represents vector (4,8,...,2048)	multiplying non-zero vector object by 4 should multiply each element by 4
5	object represents vector (0,0,...,0)	factor: 0	no output, vector object still represents vector (0,0,...,0)	multiplying zero vector object by 0 should not change the object's state
6	object represents vector (0,0,...,0)	factor: -1	no output, vector object still represents vector (0,0,...,0)	multiplying zero vector object by -1 should not change the object's state
7	object represents vector (0,0,...,0)	factor: 1	no output, vector object still represents vector (0,0,...,0)	multiplying zero vector object by 1 should not change the object's state
8	object represents vector (0,0,...,0)	factor: 4	no output, vector object still represents vector (0,0,...,0)	multiplying zero vector object by 4 should not change the object's state

Table 4: Unit Testing – *G4NeutronHPVector.cc:: G4NeutronHPVector& operator + (G4NeutronHPVector *left, G4NeutronHPVector *right)*

#	**Initial State**	**Inputs**	**Outputs**	**Description**
9	NA (vectors passed as arguments)	left: (1,2,...,512), right: (2,4,6,...,1048)	vector object representing (3,6,...,1560)	adding two non-zero vectors should output the correct vector
10	NA (vectors passed as arguments)	left: (2,4,6,...,1048), right: (1,2,...,512)	vector object representing (3,6,...,1560)	adding two non-zero vectors should output the correct vector
11	NA (vectors passed as arguments)	left: (1,2,...,512), right: (0,0,0,...,0) of length 512	vector object representing (1,2,...,512)	adding a zero-vector to a non-zero vector should output the non-zero vector
12	NA (vectors passed as arguments)	left: (0,0,...,0) of length 512, right: (1,2,...,512)	vector object representing (1,2,...,512)	adding a zero-vector to a non-zero vector should output the non-zero vector
13	NA (vectors passed as arguments)	left: (-1,-2,...,-512), right: (1,2,...,512)	vector object representing (0,0,...,0) of length 512	adding a vector to its inverse should output the zero vector