GEANT-4 GPU Port:

Test Plan

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

Version 0
October 30, 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
Reformatting Tables	Matthew	2015-10-30
Added System test cases	Matthew	2015-10-29
Initial draft of document	Stuart, Matthew, Rob, Victor	2015-10-26

List of Tables

1 General Information

1.1 Summary

The product, GEANT4-GPU, will be a modified version of G4-STORK that will leverage GPU computing to increase performance while still outputting the same results as the existing, CPU-based G4-STORK project. This testing plan outlines how the development team is expected to test the correctness and performance of the product through the use of unit testing, system 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 (independent 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 the developers check that the results are within a given margin of error, and if so the product would be deemed correct. This is the solution that is planning on being implemented at this point, however if it does not seem feasible during the design and development phases the developers are open to exploring solution one.

Risk 2 – Isolating 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, relatively simple examples will be used for most system tests (i.e. with few particles) and they will be run 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

Time is one of the largest constraints, the completed project is due next April, which is 6 months from the initial revision of this document. In order to perform rigorous testing on all functions, classes, and sections of code that have been parallelized, a large amount of time is required. G4-STORK is a large library with many different objects and functions, with each section contributing to the overall performance. Each addition of a control branch to the code must be tested to ensure that the original functionality is preserved. Maximizing performance while minimizing changes to the actual code should help to deal with the time constraint.

1.4 Definitions and Acronyms

Table 3: Definitions and Acronyms

Acronym/Definition	Definition
GEANT-4	open-source software toolkit used to simulate the passage of particles through matter
G4-STORK	(Geant-4 STOchastic Reactor Kinetics), fork of GEANT-4 developed by McMaster's Engineering Physics department to simulate McMaster's nuclear reactor
GPU	graphics processing unit, well-suited to parallel computing tasks
CPU	computer (central) processing unit, general computer processor well suited to serial tasks
GP-GPU	concept of running "general purpose" computations on the GPU
CUDA	parallel computing architecture for general purpose programming on GPU, developed by NVIDIA
PoC	proof of concept
SRS	software requirements specification

1.5 References

The following documents are referenced within this test plan.

Table 4: Referenced Documents

Document	Author(s)	Version/Date
GEANT-4 GPU Port: SRS	Stuart, Matthew, Rob, Victor	0/2015-10-12
PoC Plan for GEANT4-GPU	Stuart, Matthew, Rob, Victor	2015-10-20
UniNotesTestPlan	Umar Khan, Ashwin Samtani, Olakotumbo Agia	1/2014-04-16

2 Test Types

2.1 System Testing

There are several examples included with the GEANT4 toolkit 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 computation time for some of the examples, they will be run less frequently than the simpler ones. This will let the developers run system tests regularly on the simple files, while checking that more realistic, complex examples are also correct after bigger changes.

Table 5: System Tests

#	Initial State	Inputs	Outputs	Description
1	Fresh start up	Runs = 50 Events = 12 Particles = 20	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Default example
2	Fresh start up	Runs = 1 Events = 12 Particles = 20	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – 1 run
3	Fresh start up	Runs = 500 Events = 12 Particles = 20	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Many runs
4	Fresh start up	Runs = 50 Events = 2 Particles = 20	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Short simulation

5	Fresh start up	Runs = 50 Events = 60 Particles = 20	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Long simulation
6	Fresh start up	Runs = 50 Events = 12 Particles = 2	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Few Particles
7	Fresh start up	Runs = 50 Events = 12 Particles = 2000	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Many Particles
8	Fresh start up	Runs = 1 Events = 2 Particles = 2	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Minimum stress
9	Fresh start up	Runs = 500 Events = 60 Particles = 2000	Same output as non-GPU G4STORK	SCWRDopplerInput.txt – Stress test

2.2 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 edge cases for the function as well as several "normal" cases will be created. The results of the function for each of the inputs will be recorded from the CPU version, 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, most explicit unit tests are not currently known, as no functions have yet been ported to the GPU. The following functions have been identified that are likely to be ported first to the GPU, and the tests to be created for them have been noted as well.

Table 6: Unit Testing – *G4NeutronHPVector.hh::Times(G4double factor)*

#	Initial State	Inputs	Outputs	Description
10	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
11	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

12	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
13	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
14	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
15	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
16	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
17	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 7: Unit Testing – *G4NeutronHPVector.cc:: G4NeutronHPVector& operator + (G4NeutronHPVector *left, G4NeutronHPVector *right)*

#	**Initial State**	**Inputs**	**Outputs**	**Description**
18	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
19	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

20	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
21	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
22	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

The unit tests as described in tables 6 and 7 are not complete – as the project progresses the number of tests will increase. Since functions are being ported one-at-a-time to the GPU, having full (or near-full) coverage is relatively straightforward. To accomplish this, every ported function must have a series of associated unit tests that cover a variety of edge and normal cases. Combined with system tests to test how all the parts work together, test coverage will be exhaustive.

2.3 Code Testing

Testing the code will be done manually due to the variation in the snippets of code that will be added. Tests will based on the addition of control branches to current sections of code. When new conditionals are added, tests will be done to ensure that there are no differences in the expected output. Since the code that needs to be added is currently unknown, the tests described in the following table are general examples of how likely additions to existing code will be tested. To minimize the risk of adverse side effects after adding code, variables before and after a conditional or new statement are recorded and any changes are noted. If the changes were not intended the code added is likely at risk of causing errors.

2.4 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. The following table

Table 8: Code Testing

#	Type of code added	Inputs	Outputs	Description
23	IF statement control branch added	Variables in the class, or function before the if	Variables in the class, or function after the if	Ensure that the statement(s) in the if did not change the variables in an unexpected way. Ensure the execution of the code when the condition is false is the same when the condition is true
24	IF ELSE statement control branch added	Variables in the class, or function before the if else	Variables in the class, or function after the if else	Ensure that the statement(s) in the if else did not change the variables in an unexpected way. Ensure the execution of the code when the condition is false is the same when the condition is true
25	Statement added	Variables in the class, or function before the statement	Variables in the class, or function after the statement	Ensure all variables not modified by the statement are the same. Ensure the execution of the code with and without the statement is the same

outlines Requirements Tests to be performed at the end of Revision 0 (not including those covered by unit and system tests. Requirement number refers to that of the System Requirements Specification.

2.5 Performance Profiling

Performance results (i.e. how much improvement there is) will be measured through the system testing, however performance profiling during development will use proprietary

Table 9: Requirements Testing – Tests for Revision 0

Test #	Req. #	Inputs	Outputs	Description
26	2	NA	NA	enabling GPU computations should be straightforward – an extra flag for the CMake tool will be used to enable or disable it
27	4	NA	NA	trying to enable GPU computations on a computer with incompatible hardware should result in an appropriate error message
28	12	NA	NA	software should be at least as stable as existing product – running the new product with input files known to work on the existing product should not cause unexpected behaviour (i.e. crashing)
29	16	NA	NA	clear installation instructions shall be included with the product – a user comfortable with G4-STORK should be able to follow the instructions to install the product without any issues

CUDA profiling tools, as well as more general profiling software.

2.5.1 General Profiling

Initially during development, the "Time Profiling" tool of the "Instruments" software package will be used to identify the most time consuming operations of the G4-STORK codebase. Identifying these operations first will allow the developers to easily pinpoint the areas that would best benefit from being ported to the GPU. As more functions are ported, the developers will continue to use "Time Profiling" to identify bottlenecks.

2.5.2 CUDA Profiling

NVIDIA has developed a solution for profiling CUDA code, called the "NVIDIA Visual Profiler". This tool will be used extensively to identify the areas of the CUDA code that can be improved, and to better understand how the software is using the hardware. This profiler is available as a plugin for the Eclipse IDE, and very thoroughly analyzes CUDA code for improvements.

3 Testing Factors

3.1 Factors to be Tested

The purpoes of this section is to shed light on the features and properties of the system that we wish to stress and test. The chart below provides a brief description of every property we aim to measure and a method as means for measuring.

Table 10: Test Factors

Factors	**Description**	**Test Method**
Correctness	Objective success or failure of a function	Unit Testing
Performance	Quantitative and Qualitative measure of temporal performance and stability in a particular workload	Unit Testing, black-box, stress
Reliability	Consistent performance of functions in routine circumstances	Unit Testing, requirements

3.2 Description of Factor

3.2.1 Correctness

Tests run on the GPU-enabled product should provide the same 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
- See risk 1 for more information on how they will be compared

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.

3.2.3 Reliability

Consistent performance of functions in routine circumstances.

Rationale
Research software requires functions to perform reliably in order to produce results that can be duplicated.

Methods of testing

- Running the same test multiple times should produce the same result every time.

4 Manual and Automated 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 output against the correct output from the existing product.

4.1 Automated 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, it will be manually run by the developers. This will let the developers 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 will be used to evaluate performance.

4.2 Automated Unit Testing

A testing class will be added to the project which will contain all unit tests. These will be created as functions are ported 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. The unit tests will be manually run 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.

5 Proof of Concept Test

As documented in the Proof of Concept Plan, the demonstration will include running G4-STORK examples with and without GPU computation enabled, as well as a very minor CUDA example. Since it is not planned to have any ported functions running on the GPU, unit tests will not be used at this point. A full system test will be used to verify that adding the small CUDA example does not impact the results. It is expected that the CUDA example won't affect the results, as it will not be passing any data back to G4-STORK to be used. The empty unit testing file (along with a trivial unit test) will be included in the project, and may be demonstrated if requested.

6 Schedule

To determine where changes need to be made (first and to the code) to attain best speedup, the method described in general profiling will be used. Code tests will be performed as necessary, for example when the impact of a conditional or statement is unknown on the overall result. This would happen when code needs to be re-written or re-ordered to achieve a speedup. Unit tests will be used when functions and classes are ported to the GPU. Requirements tests will be done at the end of revision 0 to ensure that all requirements (functional and non-functional) are met. Systems tests will be run less frequently than unit tests due to the complexity of the tests, and the time it takes to run them.

6.1 Deliverables

The deliverables specific to testing are as follows:

Table 11: Timeline of Deriverables

Deliverable	**Description**	**Date**
Test Plan Revision 0	set guidelines and objectives for testing, subject to change	October 30
Test Report Revision 0	report on progress, history, and results regarding testing and test methods	March 21
Test Plan Final	rigorously outline test cases and considerations in preperation for the projects final release	April 1
Test Report Final	report on the final results of tests performed and provide relevant analytics	April 1