GEANT-4 GPU Port:

Test Plan

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

Version 0
March 21, 2016

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
Changes for rev0 presentation	Matt	2016-02-08
Added Test Schedule Table	Stuart	2015-10-30
Reformatted 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. GEANT4 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
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

Title	Author(s)	Description
GEANT-4 GPU Port: SRS	Stuart, Matthew, Rob, Victor	Requirements specification for the project
PoC Plan for GEANT4-GPU	Stuart, Matthew, Rob, Victor	Demonstration details for product's Proof of Concept
UniNotesTestPlan	Umar Khan, Ashwin Samtani, Olakotumbo Agia	Test plan outline/framework

2 Test Types

3 Unit Testing

3.1 Use of Automated Testing

3.1.1 Overview

Our unit testing system is semi-automated. The user runs a program to generate a test results text file, inputting whether or not Geant4 was compiled with CUDA enabled or disabled. Then, they recompile Geant4 in the opposite configuration (i.e. with CUDA enabled if previously disabled, and vice versa) and run the test program again. At this point there will be two test results text files, one for CUDA enabled, and one for CUDA disabled. In addition, two text files containing runtimes of all computationally-intensive functions are produced. After generating the files, a program to analyze the results is run outputting whether each test case passed or failed, and creating an Excel document (.csv) with the running times.

3.1.2 Generating Test Results

`GenerateTestResults` first initializes several G4ParticleHPVector objects from data files included with Geant4 of varying numbers of entries, including the creation of one G4ParticleHPVector with 0 entries. After the vectors have been initialized, the unit-tested methods are tested with a variety of input values. These cover edge cases (i.e. negative index for array, index greater than number of elements etc.) as well as more "normal" cases. The result of each function is then written to the results text file. This can be a single value in the case of "clean" functions that simply return a value, or it could be the state of the G4ParticleHPVector object, that is the array of points stored by that object. For performance reasons, instead of writing out the entire array of points, a hash value is generated from the array and is outputted. The value of the input variable for each function call is also outputted, so the results for specific inputs can be analyzed.

3.1.3 Analyzing Test Results

After the above files are generated, the `AnalyzeTestResults` utility runs through both documents and for each unit test outputted its status. If it failed, then the result from the CPU and from the GPU are both printed out. After the analysis completes, the total number of tests passed is outputted. In addition, `AnalyzeTestResults` will read the files containing runtimes for each function and output them in .csv format to simplify performance analysis.

3.1.4 Note About Random Results

Some of the tests run in `GenerateTestResults` are based off of random numbers, which differ between the CPU and GPU implementations. To counteract this, each of those tests is run multiple times and the result is averaged. When analyzing results for those functions, they are only marked as failed if the difference in the values of the GPU and CPU results are more than a specified tolerance. There are some functions that depend on random numbers that modify the data array. Since a hash is outputted and will differ no matter how small the difference in the values of the array are, before hashing the values are all rounded to a lower precision.

3.2 Definition of Variables Used for Unit Testing

The following are variables that are used for multiple unit tests. Instead of defining them again for each unit test they are defined here only once. Other variables used for specific unit tests will be defined in their respective unit test sections
For all unit tests:

Table 5: General Unit Test Variables

Name	Type	Description
n	G4double	number of entries in the G4ParticleHPVector
r1	G4double	-1.0
r2	G4double	0.0
r3	G4double	0.00051234
r4	G4double	1.5892317
r5	G4double	513.18
vec0	G4ParticleHPVector	0 entries
vec1	G4ParticleHPVector	80 entries
vec2	G4ParticleHPVector	1509 entries
vec3	G4ParticleHPVector	8045 entries
vec4	G4ParticleHPVector	41854 entries
vec5	G4ParticleHPVector	98995 entries
vec6	G4ParticleHPVector	242594 entries

[What unit testing framework are you planning to use? —DS] [Mentioned that we will be making our own unit testing framework —MP]

3.3 void Init(istream & aDataFile, G4int total, G4double ux, G4double uy)

Initializes the data in the current vector with `total` data points from `aDataFile`. Each data point is multiplied by factor `ux` for the x-value and `uy` for the y-value.

Each vector *vec1*, *vec2* ... *vec6* is associated with a data file, which is the input `aDataFile` to the `Init` function. These data files are bundled with Geant4, and include measured data points for a given isotope of an element. `vec0` is not initialized with a data file, as it is meant to be an empty vector to test edge cases. As such, `vec0` is not tested with this function.

Table 6: Unit Tests - `Init`

Test #	Inputs			
	`aDataFile`	`G4int`	`ux`	`uy`
1	Current data file	n	1	1

3.4 G4ParticleHPVector & operator = (const G4ParticleHPVector & right)

Create a new, temporary G4ParticleHPVector object and assign the current vector to it. Outputs the data and the integral from the new vector.

3.4.1 Test Inputs

Table 7: Unit Tests - `=` (overloaded assignment operator)

Test #	Inputs
	`right`
2	Current vector

3.5 const G4ParticleHPDataPoint GetPoint(G4int i)

Returns the G4ParticleHPDataPoint at index `i` in the current vector. The `x` and `y` values of the point are outputted.

3.5.1 Test Inputs

Table 8: Unit Tests - `GetPoint`

Test #	Inputs
	`i`
3	-1
4	0
5	n/2
6	n-1
7	n

3.6 G4double GetX(G4int i)

Returns the energy at index `i` in the current vector. The `x` value of the point are outputted.

3.6.1 Test Inputs

Table 9: Unit Tests - `GetX`

Test #	**Inputs** `i`
8	-1
9	0
10	n/2
11	n-1
12	n

3.7 G4double GetEnergy(G4int i)

Returns the energy at index `i` in the current vector. The `x` value of the point are outputted.

3.7.1 Test Inputs

Table 10: Unit Tests - `GetEnergy`

Test #	**Inputs** `i`
13	-1
14	0
15	n/2
16	n-1
17	n

3.8 G4double GetY(G4int i)

Returns the xSec at index `i` in the current vector. The `y` value of the point are outputted.

3.8.1 Test Inputs

Table 11: Unit Tests - `GetY`

Test #	Inputs `i`
18	-1
19	0
20	n/2
21	n-1
22	n

3.9 G4double GetXsec(G4int i)

Returns the xSec at index `i` in the current vector. The `y` value of the point are outputted.

3.9.1 Test Inputs

Table 12: Unit Tests - `GetXsec`

Test #	Inputs `i`
23	-1
24	0
25	n/2
26	n-1
27	n

3.10 void SetData(G4int i, G4double x, G4double y)

Sets the energy and xSec at index `i` in the current vector.

3.10.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 13: Unit Tests - `SetData`

Test #	Inputs		
	`i`	`x`	`y`
28	-1	r1, r2, r3, r4, r5	r1, r2, r3, r4, r5
29	0	r1, r2, r3, r4, r5	r1, r2, r3, r4, r5
30	n/2	r1, r2, r3, r4, r5	r1, r2, r3, r4, r5
31	n-1	r1, r2, r3, r4, r5	r1, r2, r3, r4, r5
32	n	r1, r2, r3, r4, r5	r1, r2, r3, r4, r5

3.11 void SetPoint(G4int i, const G4ParticleHPDataPoint & it)

Sets a point at a given index in a given vector.

3.11.1 Test Inputs

- "rPoint" is a G4ParticleHPDataPoint with random values
- "nPoint" is a G4ParticleHPDataPoint with negative values
- "zPoint" is a G4ParticleHPDataPoint with zero values

Commas denote multiple sub-test inputs. If one of the sub-tests fail then the whole test fails.

Table 14: Unit Tests

Test #	Inputs	
	`i`	`it`
33	-1	rPoint, nPoint, zPoint
34	0	rPoint, nPoint, zPoint
35	n/2	rPoint, nPoint, zPoint
36	n-1	rPoint, nPoint, zPoint
37	n	rPoint, nPoint, zPoint

3.12 void SetX(G4int i, G4double e)

Sets the energy at index `i` in the current vector.

3.12.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 15: Unit Tests - `SetX`

Test #	Inputs	
	`i`	`e`
38	-1	r1, r2, r3, r4, r5
39	0	r1, r2, r3, r4, r5
40	n/2	r1, r2, r3, r4, r5
41	n-1	r1, r2, r3, r4, r5
42	n	r1, r2, r3, r4, r5

3.13 void SetEnergy(G4int i, G4double e)

Sets the energy at index `i` in the current vector.

3.13.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 16: Unit Tests - `SetEnergy`

Test #	Inputs	
	`i`	`e`
43	-1	r1, r2, r3, r4, r5
44	0	r1, r2, r3, r4, r5
45	n/2	r1, r2, r3, r4, r5
46	n-1	r1, r2, r3, r4, r5
47	n	r1, r2, r3, r4, r5

3.14 void SetY(G4int i, G4double e)

Sets the xSec at index `i` in the current vector.

3.14.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 17: Unit Tests - `SetY`

Test #	Inputs	
	`i`	`e`
48	-1	r1, r2, r3, r4, r5
49	0	r1, r2, r3, r4, r5
50	n/2	r1, r2, r3, r4, r5
51	n-1	r1, r2, r3, r4, r5
52	n	r1, r2, r3, r4, r5

3.15 void SetXsec(G4int i, G4double e)

Sets the xSec at index `i` in the current vector.

3.15.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 18: Unit Tests - `SetXsec`

Test #	Inputs	
	`i`	`e`
53	-1	r1, r2, r3, r4, r5
54	0	r1, r2, r3, r4, r5
55	n/2	r1, r2, r3, r4, r5
56	n-1	r1, r2, r3, r4, r5
57	n	r1, r2, r3, r4, r5

3.16 G4double Sample()

Performs samples of the vector according to interpolation its interpolation scheme.

3.16.1 Test Inputs

Table 19: Unit Tests - `Sample`

Test #	Inputs
	`N/A`
58	N/A

3.17 G4double SampleLin()

Performs samples of the vector with a linear interpolation scheme.

3.17.1 Test Inputs

Table 20: Unit Tests - `SampleLin`

Test #	**Inputs** `N/A`
59	N/A

3.18 void Times(G4double factor)

Multiplies every element in the vector by `factor`.

3.18.1 Test Inputs

Table 21: Unit Tests - `Times`

Test #	**Inputs** `factor`
60	r1
61	r2
62	r3
63	r4
64	r5

3.19 void ThinOut(G4double precision)

Removes any element from the vector whose neighbor is closer than `precision`.

3.19.1 Test Inputs

Table 22: Unit Tests - `ThinOut`

Test #	Inputs `factor`
65	r1
66	r2
67	r3
68	r4
69	r5

3.20 G4double GetXsec(G4double e)

Returns the first xSec from the current vector whose energy is greater than `e`.

3.20.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 23: Unit Tests - `GetXsec`

Test #	Inputs `e`
70	r1, r2, r3, r4, r5
71	r1, r2, r3, r4, r5
72	r1, r2, r3, r4, r5
73	r1, r2, r3, r4, r5
74	r1, r2, r3, r4, r5

3.21 G4double GetXsec(G4double e, G4int min)

Returns the first xSec from the current vector whose energy is greater than `e`.

3.21.1 Test Inputs

Commas denote multiple sub test inputs. If one of the sub tests fail then the whole test fails.

Table 24: Unit Tests - `GetXsec`

Test #	Inputs	
	`e`	`min`
75	r1, r2, r3, r4, r5	-1
76	r1, r2, r3, r4, r5	0
77	r1, r2, r3, r4, r5	n/2
78	r1, r2, r3, r4, r5	n-1
79	r1, r2, r3, r4, r5	n

3.22 G4double Get15percentBorder()

Returns the integral from each data point to the last data point and returns the first one within 15% of the last data point.

3.22.1 Test Inputs

Table 25: Unit Tests - `Get15percentBorder`

Test #	Inputs
	`N/A`
80	N/A

3.23 G4double Get50percentBorder()

Returns the integral from each data point to the last data point and returns the first one within 50% of the last data point.

3.23.1 Test Inputs

Table 26: Unit Tests - `Get50percentBorder`

Test #	Inputs
	`N/A`
81	N/A

4 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. Examples in 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 27: System Tests

#	Initial State	Inputs	Outputs	Description
82	Fresh start up	Events = 2000 Material = Water	Same output as non-GPU GEANT4	HADR04 no changes
83	Fresh start up	Events = 2000 Material = Uranium	Same output as non-GPU GEANT4	HADR04 – basic example
84	Fresh start up	Events = 600 Material = Water	Same output as non-GPU GEANT4	HADR04 – Shorter test
85	Fresh start up	Events = 600 Material = Uranium	Same output as non-GPU GEANT4	HADR04 – Shorter test
86	Fresh start up	Events = 20000 Material = Uranium	Same output as non-GPU GEANT4	HADR04 – Long simulation stress Test

[Not using G4-STORK for system tests - therefore had to change system tests —MP]

4.1 Code Testing

The developers will follow strict code testing guidelines to help reduce coding errors and improve collaboration. This will be done by adhering to the *Feature Branch Workflow* and enforcing code reviews for every commit. Developers will create separate branches for each new feature, commit to that branch and then push it to the repository. Using GitHub's pull request feature, all branches will go through a code review process. The author will open a pull request for their feature, and a different developer (to be chosen based on availability ["availability" —DS][fixed spelling mistake —MP] and knowledge of the feature) will be assigned as reviewer. The reviewer will manually read through the changes on GitHub and post comments as necessary. After comments have been addressed, the reviewer will merge the feature branch into the master. Following such a strict model will ensure that all committed code is double-checked by a different

developer, which can catch many bugs and errors. This practice will not apply to changes to documentation.

4.2 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 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.

Table 28: Requirements Testing – Tests for Revision 0

Test #	**Req. #**	**Inputs**	**Outputs**	**Description**
87	2	NA	NA	enabling GPU computations should be straightforward – an extra flag for the CMake tool will be used to enable or disable it
88	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)
89	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

[removed hardware detection, no longer a requirement —MP]

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

4.3.1 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.

4.3.2 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.

5 Testing Factors

5.1 Factors to be Tested

The purpose ["purpose" —DS][fixed spelling mistake —MP] 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 29: 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, System stress testing [How are you stress-testing the system? —DS] [defined what type of stress testing will be used —MP]
Reliability	Consistent performance of functions in routine circumstances	Unit Testing, requirements

5.2 Description of Factor

5.2.1 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

5.2.2 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.

5.2.3 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.

6 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.

6.1 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.

6.2 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.

7 Proof of Concept Test

As documented in the Proof of Concept Plan, the demonstration will include running GEANT4 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.

8 Schedule

The testing schedule can be summarized as follows: manual code reviews for every commit, unit tests added for every ported function and run throughout development and passed before every commit, system tests run throughout development and passed before every commit, performance profiling used throughout development, requirements testing done for Revision 0. The following testing timeline will be followed. The *date* column refers to the date by which the event shall be completed, if it is not a range.

Table 30: Testing Schedule

Date	**Test Type**	**Event**	**Testers**
2015-11-04	System Test	Input files will be chosen for system tests, and their outputs on the existing product recorded	Everyone
2015-11-04	System Test	Script will be created to run system test and compare results. Running the script on the existing product will pass & Victor and Rob	
2015-11-04 - 2016-04-01	System Test	System tests will be run throughout development, and must be passed before every commit	Matt & Stuart
2015-11-04 - 2016-04-01	Code Testing	All changes will be reviewed by a different developer before merging	Matt
2015-11-06	Performance Profiling	Profiling tools will be used to identify the most computationally involved portions of the codebase	Rob & Victor
2015-11-10	Unit Test	Unit tests framework file will be created and added to the project. Methods will be added to run tests and report their results.	Victor

2016-11-10	Unit Test	Two trivial unit tests (one that will fail, one that will pass) will be added to unit tests file. Running the unit tests should produce expected outputs. Rob & Victor	
2015-11-10 - 2016-04-01	Unit Test	Unit tests will be added throughout development for every function that is ported to the GPU.	Matt
2015-11-10 - 2016-04-01	Unit Test	Unit tests will be run throughout development, and must be passed before every commit	Matt
2016-02-10	Requirements Testing	All requirements tests will be evaluated at Revision 0, results will guide product development for Revision 1.	Everyone

[Who will be doing what? —DS][Added people to work on specific testing tasks —MP]

8.1 Deliverables

The deliverables specific to testing are as follows:

Table 31: Timeline of Deriverables

Date	**Deliverable**	**Description**
2015-10-30	Test Plan Revision 0	set guidelines and objectives for testing, subject to change
2016-03-21	Test Report Revision 0	report on progress, history, and results regarding testing and test methods
2016-04-01	Test Plan Final	rigorously outline test cases and considerations in preparation ["preparation" —DS][fixed —MP] for the projects final release
2016-04-01	Test Report Final	report on the final results of tests performed and provide relevant analytics

[Will you be doing any tests that result in exceptions? How will those be performed? You are missing a lot of edge/boundary test cases (or you do not make it clear that they are being tested). —DS] [Added a few exception and edge case tests in unit testing and system testing —MP]