

Running Geant4 Functions on a GPU

Discussion of Results

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

McMaster University

April 14, 2016

Overview

1 Introduction

- Brief Project Overview
- Explanation of Terms
- Scope
- Purpose

2 Features

- Easily Enable/Disable GPU Acceleration
- Impl. 1: Existing Module in GPU Memory
- Impl. 2: Add New GPU-Accelerated Methods to Interface
- Accuracy / Testing

3 Conclusion

- Summary of Results
- Recommendations

Brief Project Overview

Take an existing particle simulation toolkit - Geant4 - and have some functions run on a GPU device to improve performance.

What is Geant4?

- Geant4 is a toolkit that is meant to simulate the passage of particles through matter.
- It has been developed over the years through collaborative effort of many different institutions and individuals.
- Geant4's diverse particle simulation library has a wide variety of applications including
 - High energy physics simulations
 - Space and radiation simulations
 - Medical physics simulations

Demonstration

*Demonstration – Running Geant4 on the CPU
Hadr04 With Visualization*

What is GP-GPU Computing?

- General-purpose graphic-processing-unit computing is a re-purposing of graphics hardware
- Allows GPUs to perform computations that would typically be computed on the CPU
- If a particular problem is well suited to parallelization, GP-GPU computing can greatly increase performance

Scope

- Make current CPU functions available for use on GPU
 - Add appropriate prefixes to function definitions
 - Make use of multiple parallel threads to execute each function
- Ensure correctness of each GPU available function by matching results to the corresponding CPU function
- Compare performance of GPU available functions to CPU functions

Possible Implementations

There were initially five possible implementations to reach a solution:

- Port Geant4 to the GPU such that each particle runs in parallel
- Port all the functions of some class(es) to the GPU, with those functions privatized to the GPU
- Port some functions of some class(es) to the GPU, memory stored on host, passing mem to device as necessary
- Port some functions of some class(es) to the GPU, memory stored and updated on host and device
- Port some functions of some class(es) to the GPU, data divided between host and device, passing mem as necessary

Solution Choice

- Implementation 1 was believed to be unreasonable given schedule/resource limitations
- Implementation 5 was found to be most suitable
 - Easy to switch between CPU & GPU versions
 - Less memory usage than other implementations
 - Least redundancy in computation

Purpose

- Determine if target functions are suitable to parallelization
- Increase performance of functions when run on GPU
- Decrease time required to run simulations involving ported functions

Features

- GPU acceleration available on an “opt-in” basis
- Easy to enable/disable GPU acceleration
- If GPU acceleration is enabled, some methods will run on GPU
- Same results whether acceleration enabled or disabled

Easily Enable/Disable GPU Acceleration

- Existing projects can use GPU acceleration without having to change any code
- Flag during build phase enables/disables GPU acceleration
- Interface remains the same¹, acceleration happens “invisibly”

¹implementation 1 only

Demonstration

Demonstration – Enabling CUDA Acceleration

Easily Enable/Disable GPU Acceleration

Method calls to G4ParticleHPVector forwarded to GPU-based implementation

- This decision is made at compile time based on cmake flag

Example of Forwarding Method Calls

```
inline G4double GetY(G4double x)
{
    #if GEANT4_ENABLE_CUDA
        return cudaVector->GetXsec(x);
    #else
        return GetXsec(x);
    #endif
}
```

Accelerating Module on GPU

Existing module `G4ParticleHPVector` ported to GPU using CUDA

Definition: CUDA

CUDA is a GP-GPU programming model developed by NVIDIA, for use with NVIDIA graphics cards

Why G4ParticleHPVector?

- Represents empirically-found probabilities of collisions for different particles based on their energy
- Identified as starting point by relevant stakeholders
 - Used heavily in simulations run by stakeholders
- Seems well-suited to parallelization
 - Based on large vector of 2D points
 - Performs calculations over this vector
 - Sorted by x-value (particle energy)

Two Implementations

- 1 Forward all calls to existing G4ParticleHPVector interface to a GPU-based implementation of the module
 - Store data vector in GPU memory
 - Copy results back to the CPU to return to the caller
- 2 Add new methods to G4ParticleHPVector interface that are well-suited to GPU computing
 - Copy data vector to GPU memory on method call
 - Existing G4ParticleHPVector methods unchanged, continue to run on CPU

Impl. 1: Existing Module in GPU Memory

Calls to `G4ParticleHPVector` forwarded to new GPU-based class

Pros:

- + Do not have to maintain a copy of the vector on the CPU
- + Do not have to maintain a hashed vector
- + Reduces how much is being copied to the GPU

Cons:

- All methods are run on the GPU

Demonstration

Demonstration – Running Geant4 on the GPU Hadr04 With Visualization

Impl. 1 – Times

Times_CUDA

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;  
if (tid < nEntries)  
    theData[tid].xSec = theData[tid].xSec * factor;
```

Impl. 1 – GetXSec

GetXSec_CUDA

```
int start = (blockDim.x * blockIdx.x + threadIdx.x);  
for (int i = start; i < nEntries; i += numThreads)  
    if (theData[i].energy >= e) {  
        resultIndex = Min(resultIndex, i);  
        return;  
    }
```

Impl. 1: Performance Results Summary

- Most methods slower on GPU until ~10,000 entries in data vector
- Most *commonly-used* methods significantly slower on GPU, even with large data vector
 - Lots of data accesses
- Many problems in vector class not well-suited to parallelism

Impl. 1: Performance Results – Times

- Multiplies each point in vector by factor

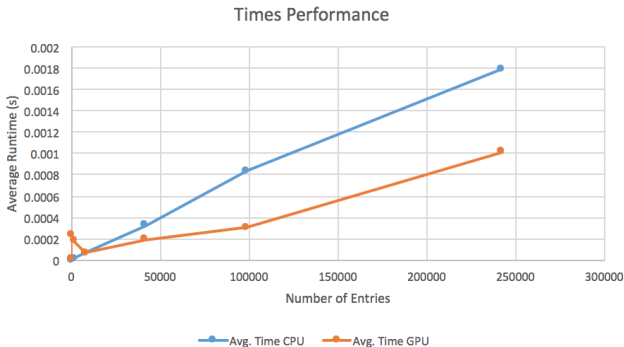


Figure: Runtime vs. Number of Data Points – Times

Impl. 1: Performance Results – GetXSec

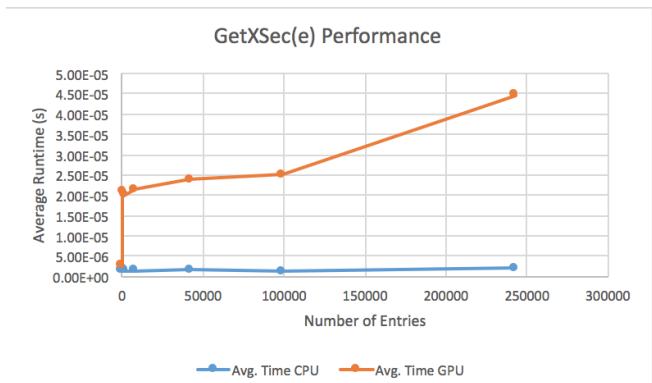


Figure: Runtime vs. Number of Data Points – GetXSec

Impl. 1: Performance Results – SampleLin

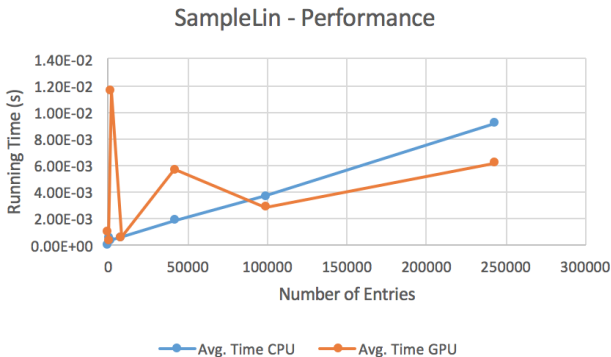


Figure: Runtime vs. Number of Data Points – SampleLin

Impl. 1: Performance Results – System Tests

System Test #1:

CPU Time	GPU Time	Speedup of GPU
54.55s	72.08s	-1.32×

Table: Performance - Water, 2000 events

System Test #2:

CPU Time	GPU Time	Speedup of GPU
54.55s	72.08s	-1.32×

Table: Performance - Uranium, 2000 events

Impl. 1: Performance Results – System Tests (Cont.)

System Test #3:

CPU Time	GPU Time	Speedup of GPU
54.55s	72.08s	-1.32×

Table: Performance - Water, 2000 events

System Test #4:

CPU Time	GPU Time	Speedup of GPU
54.55s	72.08s	-1.32×

Table: Performance - Uranium, 2000 events

Impl. 1: Performance Discussion

- Simple “getters” and “setters” now require copy from GPU to CPU memory
- Current code calling `G4ParticleHPVector` more data-oriented than computation-oriented
- Low `GetXSec` performance due to lack of `Hash` object on GPU to accelerate finding min index
- Although some functions faster, rarely used in practice

Impl. 2: Add New GPU-Accelerated Methods to Interface

Add new methods to G4ParticleHPVector interface that are well-suited to parallelism

Pros:

- + Only methods that run faster on the GPU are implemented
- + Not forced to run methods that run slowly on GPU

Cons:

- Will have to maintain two copies of the vector
- More copying the vector to and from the GPU

Impl. 2: GetXSecList

- Fill an array of energies for which we want the cross section values for
- Send the array to the GPU to work on
- Each thread works on its own query(s)

Implementation – GetXSecList

GetXSecList

```
stepSize = sqrt(nEntries);  
i = 0;  
e = queryList[threadID];  
  
for (i = 0; i < nEntries; i += stepSize)  
    if (d_theData[i].energy >= e)  
        break;
```

Implementation – GetXSecList -- cont

GetXSecList – cont

```
i = i - (stepSize - 1);  
  
for (; i < nEntries; i++)  
    if (d_theData[i].energy >= e)  
        break;  
  
d_queryList[threadID] = i;
```


Impl. 2: Performance Results Summary

Performance of implementation 2 also proved slower than original Geant4 implementations of ParticleHPVector

- Buffered implementation begins to taper off, but at a much slower rate than the original

Impl. 2: Performance Results – GetXSecList

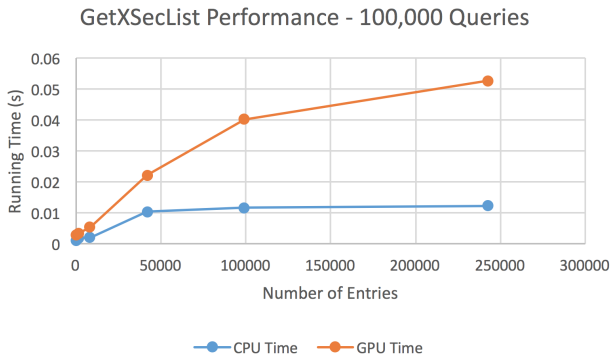


Figure: Runtime vs. Number of Data Points – GetXSecList, 100,000 Queries

Impl. 2: Performance Discussion

- CPU implementation makes use of Hash to quickly find minimum index
- Finding first element satisfying predicate not well-suited to parallelism
- If one thread finds element, must wait for all other threads (blocked ifs)

Accuracy

- All modified functions except SampleLin and Sample yield results that precisely match original implementations
- Some functions fell extremely close in accuracy to the original, and were considered to 'pass'
- The average of 1000 SampleLin tests deviated from the average of 1000 tests of the original with an error of 0.01
- The system tests differ if the number of nentries is greater than 500; if not however the results of the system test conform.

Accuracy Discussion

- The deviations in SampleLin and Sample can be attributed to the functions use of random numbers
- The negligible deviations in other ported functions are likely attributed to differences in CPU and GPU arithmetic, leading to different round-off errors

Testing

- Comparing test results and performance with GPU acceleration enabled and disabled
- Testing framework based on two phases, one program for each phase
 - 1 `GenerateTestResults`: Run unit tests and save results to file
 - 2 `AnalyzeTestResults`: Compare results from CPU and GPU
- Run `GenerateTestResults` once for GPU acceleration enabled, once with it disabled

GenerateTestResults Details

- Includes testing version number in results file for analysis stage
- Outputs simple results directly to results file
- For vectors, calculates hash for vector and output it
- Outputs timing data to separate file

Example: Snippet of Generated Test Results

```
#void G4ParticleHPVector_CUDA::GetXsecBuffer(  
    G4double * queryList, G4int length)_6  
@numQueries=10  
hash: 16548307878283220284  
@numQueries=50  
hash: 3204132713354913775
```

Demonstration

Demonstration – Generating Test Results

AnalyzeTestResults Details

Two main functions:

- 1 Compare results for each test case, printing status to stdout
 - If test failed, output differing values
 - Summarize test results at the end with number passed
- 2 Generate .csv file from timing data
 - One row per unique method call, columns show CPU time, GPU time, method name and parameters
 - Can use Excel to analyze performance results

Demonstration

Demonstration – Analyzing Test Results

Summary of Results

- Both Implementations are on average slower than the CPU
- Most methods slower on GPU until 10,000 entries in data vector
- Most commonly-used methods significantly slower on GPU, even with large data vector
 - Lots of data accesses
- SampleLin has accuracy issues due to random number generation

Recommendations

For further work with regards to ParticleHPVector:

- Abstract further up the Geant4 system, parallelizing components that make reference to NeutronHPVector
- This will decrease the frequency of data transfer between the host and device
- Up-to-date work can be found on our github, along with instructions for installing and testing

For further work with regards to parallelizing Geant4:

- Try parallelizing other commonly use components in similar style
 - Look for classes manipulating list-style data structures
 - Classes with functions that have nested loops or are heavy in computing are prime candidates
 - Probabilistic functions and getter/setter functions won't have considerable benefits
 - Functions with conditional branching may cause bottlenecks in parallelization