Running Geant4 Functions on a GPU

Discussion of Results

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

McMaster University

April 15, 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
   - Final Thoughts

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 methods available for use on GPU
    - Update build system to support compiling and linking with GPU code
    - Rewrite algorithms in parallel fashion to run on GPU
- Ensure correctness of each GPU-available method by matching results to the corresponding CPU method
- Compare performance of GPU-available methods to CPU methods

Design Phase – Possible Implementations

There were initially two different implementation approaches:

- Port much of Geant4 to the GPU such that each particle runs in parallel
  - Unreasonable given schedule/resource limitations
- Port all methods in some modules to the GPU, storing all relevant data in GPU memory
  - Easy to switch between CPU & GPU implementations
  - Supports splitting up work by module and by method, and working incrementally

Purpose

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

Features

Overview of Main 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 same[1], acceleration happens behind the scenes

[1] implementation 1 only

Demonstration

Demonstration – Enabling CUDA Acceleration

Easily Enable/Disable GPU Acceleration

Methods with GPU versions forwarded to GPU-based implementation at compile time

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

G4ParticleHPVector Overview

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[2]
+ Data rarely copied to GPU memory

Cons:

– All methods are run on the GPU, even if not well-suited to parallelism
– Return values must be copied from GPU memory to CPU memory (slow)

[2] CPU cache was implemented later

Demonstration

Demonstration – Running Geant4 on the GPU
Hadr04 With Visualization

Caching Data Vector in CPU Memory

To improve data-copying performance, maintain cache of data in CPU memory as well

- Only updated when necessary
- For methods that are not parallelizable, can run on CPU using cached data

CopyToCpuIfDirty

```
if (isDataDirtyHost) {
   cudaMemcpy(h_theData, d_theData, nEntries);
   isDataDirtyHost = false;
}
```

Impl. 1 – Times

Multiplies each element in data vector by factor

Times_CUDA

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

Impl. 1 – GetXSec

Returns y-value of first point with energy at least e parameter

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

- Methods generally 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 copying
- Many problems in vector class not well-suited to parallelism

Impl. 1: Performance Results – `Times`

Multiplies each point in vector by factor

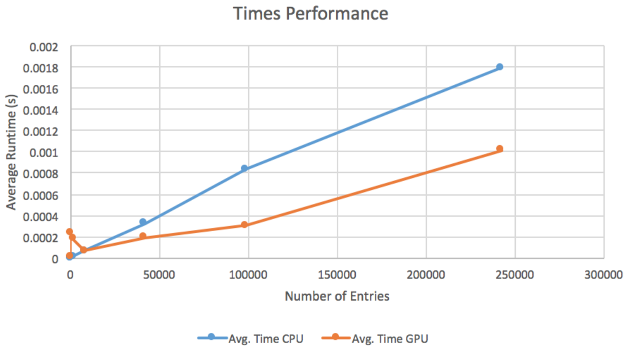

Figure: Runtime vs. Number of Data Points – `Times`

Impl. 1: Performance Results – `GetXSec`

Returns y-value of first point with energy at least ‘e’

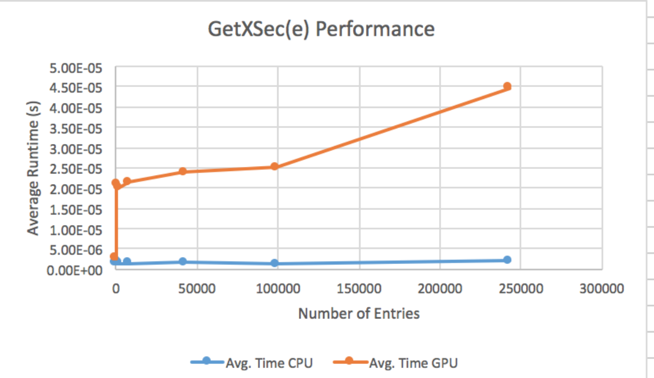

Figure: Runtime vs. Number of Data Points – `GetXSec`

Impl. 1: Performance Results – `SampleLin`

Interpolates between two random, consecutive points and their corresponding integrals

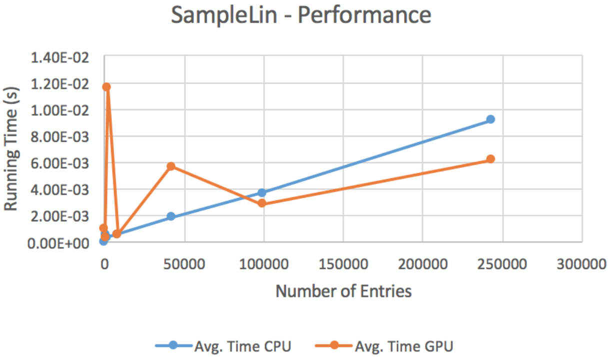

Figure: Runtime vs. Number of Data Points – `SampleLin`

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, 500 events

System Test #2:

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

Table: Performance - Water, 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 methods 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

Cons:

- Will have to maintain two copies of the data vector
- Will need to copy the vector to the GPU whenever method called

Impl. 2: `GetXSecList`

`GetXSecList` Overview

- 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);
e = queryList[threadID];

for (int 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 similar CPU-based method

- Performance on GPU linear to number of elements in data array
- Performance on CPU not affected by number of elements after point, due to saved hashes

Impl. 2: Performance Results – `GetXSecList`

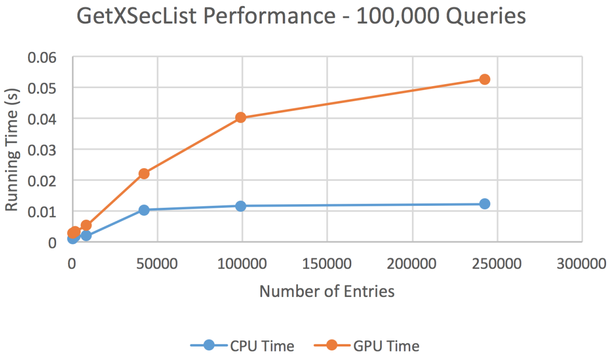

Figure: Runtime vs. Number of Data Points – `GetXSecList`, 100,000 Queries

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 (blocking divergence)

Accuracy

- All modified methods except `SampleLin` and `Sample` yield results that precisely match original implementations
    - Some methods fell extremely close in accuracy to the original, and were considered to ‘pass’
    - For `Sample` and `SampleLin`, the average of 1000 tests was compared, with a relative error tolerance of 0.01
- The system test results differ with more than 500 events
    - `Sample` and `SampleLin` only called if more than 500 events

Accuracy Discussion

- The deviations in `SampleLin` and `Sample` can be attributed to their use of random numbers
  - CUDA random number generator will have different results than `rand()`
  - Both methods take random point and interpolate it and its neighbour, so values differ significantly based on random number
- The negligible deviations in other ported methods are attributed to small differences in CPU and GPU arithmetic round-off errors (`log`, `exp`, etc.)

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 with GPU acceleration enabled, once with it disabled

GenerateTestResults Details

- Outputs simple results directly to results file
- For arrays, calculates hash for array 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
```

`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
2. Generate `.csv` file from timing data
   - One row per unique method call comparing CPU and GPU times
   - Can use Excel to analyze performance results

Demonstration

Demonstration – Analyzing Test Results

Summary of Results

- Impl. 1 was about 1.3X slower on average in system tests
- Impl. 2 was about 4X slower in unit tests
- Most commonly-used methods not well-suited to parallelism
- `Sample` and `SampleLin` have accuracy issues due to random number generation

For further work parallelizing Geant4:

- Use Geant4-GPU project as framework for parallelizing other modules
- Look for modules storing large amounts of structured data
- Methods with nested loops are prime candidates for parallelization
- Probabilistic methods and getter/setter methods won't have considerable benefits
- Methods with extensive conditional branching may cause difficulties in parallelizing

Conclusion

- All project collaborators have gained a lot of experience
- Lack of speedup disappointing, but lets us know `G4ParticleHPVector` not perfect candidate
- Parallelization of existing software is hard