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

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

What is GP-GPU

- 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

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
- 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
- No new functions to learn [1]

[1] implementation 1 only

Implementation

- Header forwards function calls to GPU or CPU Implementation
- This decision is made at compile time

Accelerating Module on GPU

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

Implementation – `Times`

Implementation – `GetXSec`

Implementation – SampleLin

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

Performance Results – Times

- Multiplies each point in vector by factor

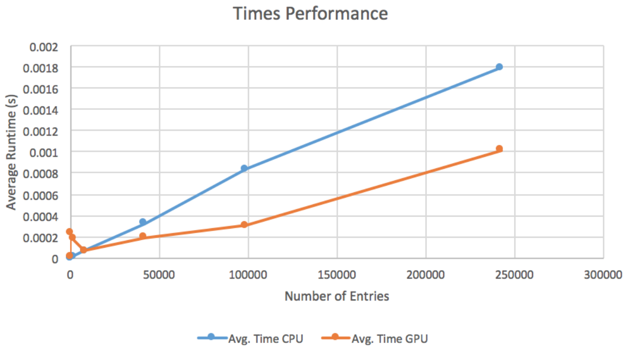

Figure: Runtime vs. Number of Data Points – Times

Performance Results – `GetXSec`

Performance Results – `SampleLin`

Performance Results – System Tests

Performance Discussion

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

Implementation – `GetXSecList`

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

Implementation – `GetXSecList`

Performance Results Summary

Performance Results – `GetXSecList`

Performance Results – System Tests

Performance Discussion

Accuracy

- All modified functions except SampleLin and Sample yield results that precisely match
- 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 negligable 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 files from CPU and GPU runs
- Run `GenerateTestResults` once for GPU acceleration enabled, once with it disabled

`GenerateTestResults` Details

- Outputs simple results directly
- For vectors, calculates hash for vector and output it
-

Summary of Results

Recommendations