Experiences Porting a Molecular Dynamics Code to GPUs on a Cray XK7

Don Berry[1] Joseph Schuchart[2] Robert Henschel[1]

[1]Indiana University

[2]Oak Ridge National Laboratory

May 9, 2013

Overview

IUMD

- IUMD is a code for molecular dynamics simulations of dense nuclear matter in astrophysics.
- Developed at Indiana University by Prof. Charles Horowitz, and Don Berry (UITS)
- Different from typical MD codes such as NAMD, AMBER, GROMACS used in chemistry and biology
- Currently running version 6.2.0 in production on Kraken and (old) Big Red
  - Hybrid MPI+OpenMP
  - Main MPI call is to MPI_Allreduce across all processes
  - Scales well to 128-MPI $\times$ 6-OpenMP $=$ 768 cores on Kraken
- This talk is about a new version, 6.3.0 that will scale to higher MPI process counts and use the GPUs on Big Red II (XE6/XK7)

White Dwarf Stars (WD)

- End stage of sun-like stars ($\approx 1 M_{\odot}$)
- Composed of completely ionized atoms, mostly C, O, Ne
- Ions interact via screened Coulomb potential

$$V(r) = \frac{Z_i Z_j e^2}{r} e^{-r/\lambda}$$

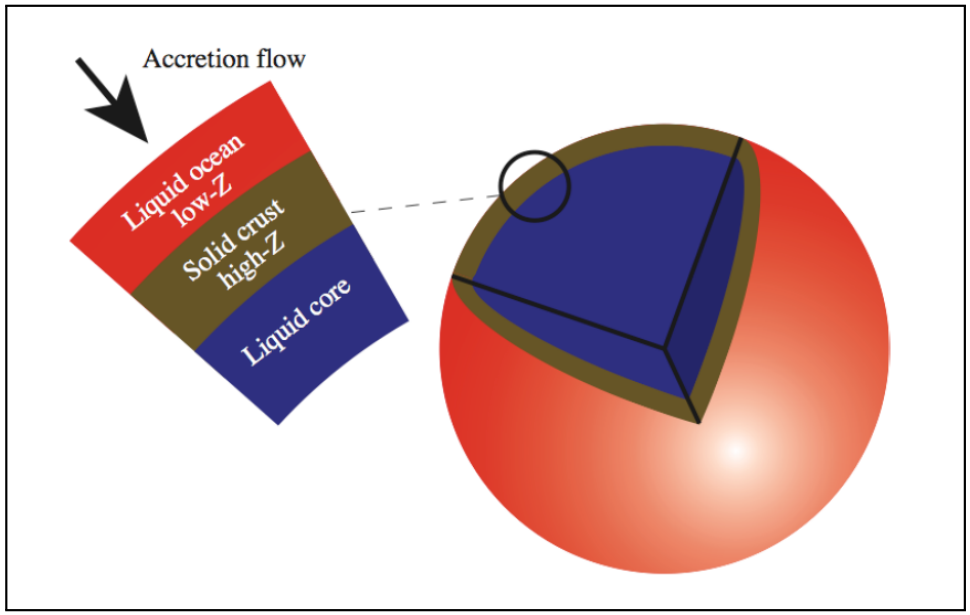
- Corresponding force is

$$\mathbf{f}(r) = \frac{Z_i Z_j e^2}{r^3} \left(1 + \frac{r}{\lambda}\right) e^{-r/\lambda} \mathbf{r}$$

Neutron Stars (NS)

- End stage of $\approx 8M_{\odot}$ stars
  - Most mass is ejected in a Type II Supernova
  - Remaining mass $\approx 1.4M_{\odot}$ is the NS
  - Radius $\approx 10km$
- Interior is neutrons
- Ocean and crust consist of many heavy ion species
- Screened Coulomb interaction still applies to ocean and crust material

- Between crust and interior is "Nuclear Pasta"
  - Odd phase where nucleons form long strings and sheets
  - IUMD can simulated this matter too, but not discussed here



Uses of MD in Physics of Compact Stellar Matter

MD has an important role to play in the study of dense stellar matter in white dwarfs, neutron stars and supernovas.

- WD,NS: Phase diagram (freezing temperature vs. composition)
- WD,NS: Electric and thermal conductivity (related to cooling time)
- NS: Insight into pycnonuclear fusion reactions
- NS: Stress-strain diagram of solid crust
  - Important for estimating maximum size of mountains
  - Mountains on rotating neutron stars radiate gravitational waves that may be detectable
- NS,SN: Nuclear pasta formation
- SN: Neutrino transport

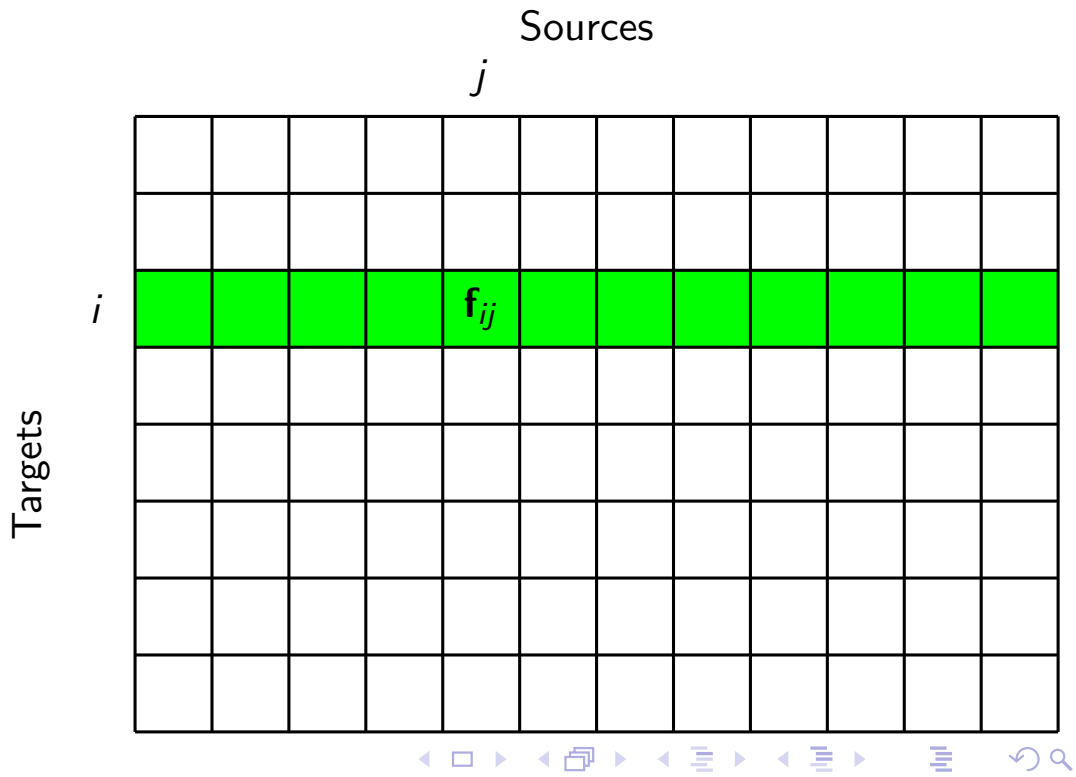
Particle-Particle (PP) Algorithm

- Range of force $\approx$ box edge length
- Methods for short range interactions not useful
- Cannot use FMM for long-range Coulomb
  - It is specific to $1/r$ potential
- Simply sum force of every particle on particle i:

$$\mathbf{F}_i = \sum_{\substack{j=0 \\ j \neq i}}^{\mathbf{N}-1} \mathbf{f}_{ij}$$

Force Matrix

- Convenient to think of *source* particles acting on *target* particles



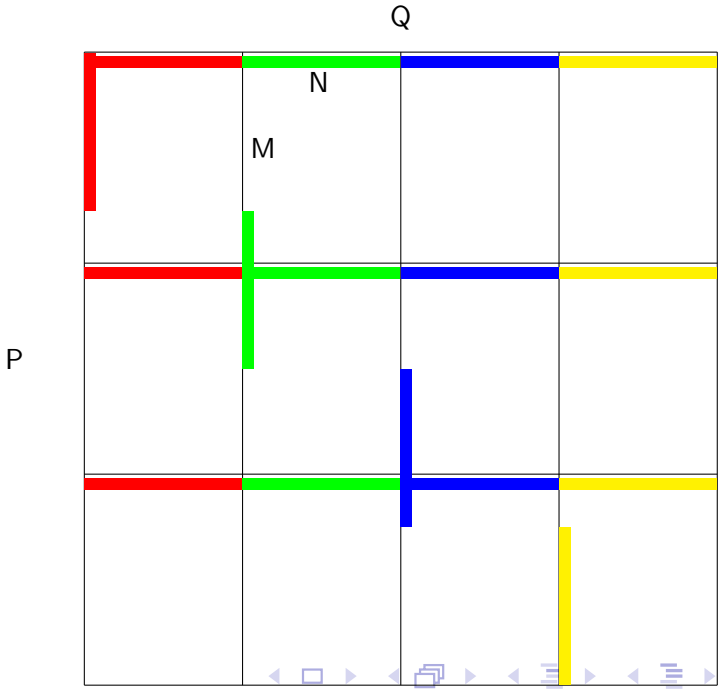
MPI Algorithm

Computation

- Partition force matrix into a $P \times Q$ Cartesian grid of MPI processes
- $M = \mathbf{N}/P$ targets/proc
- $N = \mathbf{N}/Q$ sources/proc
- Process (p, q) applies N sources to M targets
  - $\mathbf{F}_i^q = \sum_{j \in S_q} \mathbf{f}_{ij}, \ i \in T_p$

Communication

- MPI_Allreduce across rows to get total force
  - $\mathbf{F}_i = \sum_{q=0}^{Q-1} \mathbf{F}_i^q$
- MPI_Allreduce along columns to get new source postions



Newton Subroutine

Over 99% of runtime every time step is spent executing the `newton` subroutine. It updates target positions and velocities, and calls the force kernel. Our benchmark results measure average time to complete one call.

- $\mathbf{r}_i = \mathbf{r}_i + \mathbf{v}_i \Delta t + \frac{1}{2}\mathbf{a}_i \Delta t^2$
- call MPI_Allreduce to distribute $\mathbf{r}_T$ as new $\mathbf{r}_S$
- $\mathbf{v}_i = \mathbf{v}_i + \frac{1}{2}\mathbf{a}_i \Delta t$
- call force kernel to get new $\mathbf{a}_i,\ i = 0 \ldots M-1$
- $\mathbf{v}_i = \mathbf{v}_i + \frac{1}{2}\mathbf{a}_i \Delta t$

NVIDIA K20 GPU

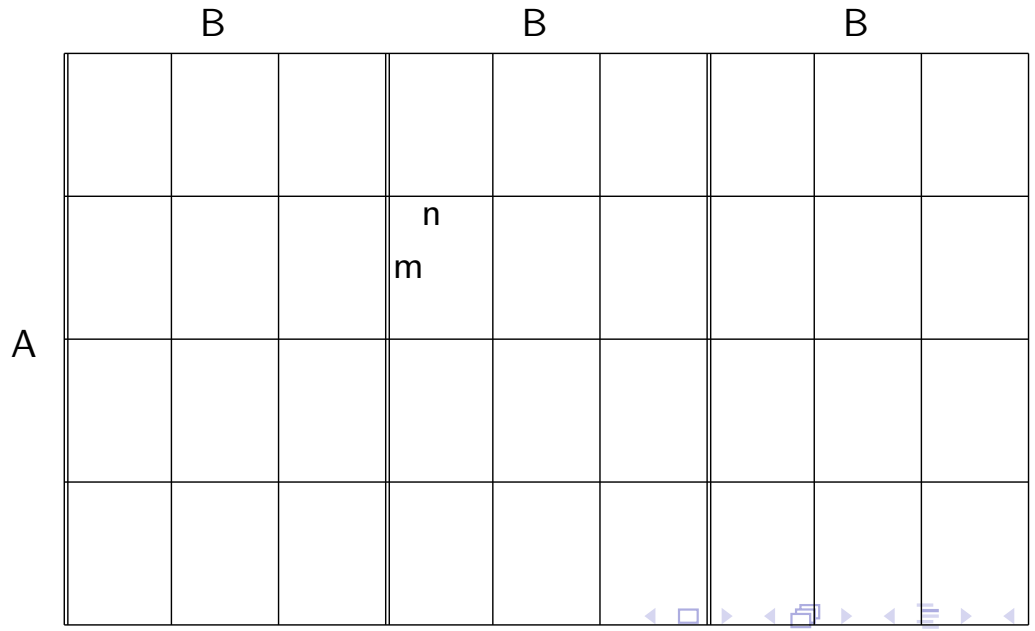
- 13 Streaming Multiprocessors (SMX)
  - 192 single precision cores
  - 64 double precision cores
  - 32 special function units
- 16KB/48KB, 32KB/32KB or 48KB/16KB Shared/L1
- max 1024 threads per thread block
- max 16 thread blocks per SMX
- 5 GB GDDR3 device memory

OpenACC

- **In main program, copyin charges and positions:**
  !$acc data copyin(ztii,zsii,xs,xt) create(at)
- **In subroutine newton:**
  !$acc update device(xt) async
  !$acc update device(xs) async
- **In force kernel:**
  !$acc wait
  !$acc kernels present_or_copyin(xt,xs,at)
  do i=0,ntgt-1
    !Calculate force on i from all sources.
    !Divide by mass to get acceleration at(i).
  end do
  !$acc end kernels
  !$acc update host(at)

Partition $M \times N$ Submatrix

- Each $M \times N$ submatrix is assigned to a GPU
  - $A \times B$ grid of submatrices
  - $A \times B$ grid of thread blocks
- A, B determined by:
  - C = number of source chunks
  - (m, l) = thread block dims.
- Each submatrix will have:
  - m targets
  - $n = lm/4$ sources
- Thus matrix is partitioned into,
  - A = M/m block rows
  - B = N/(nC) block columns

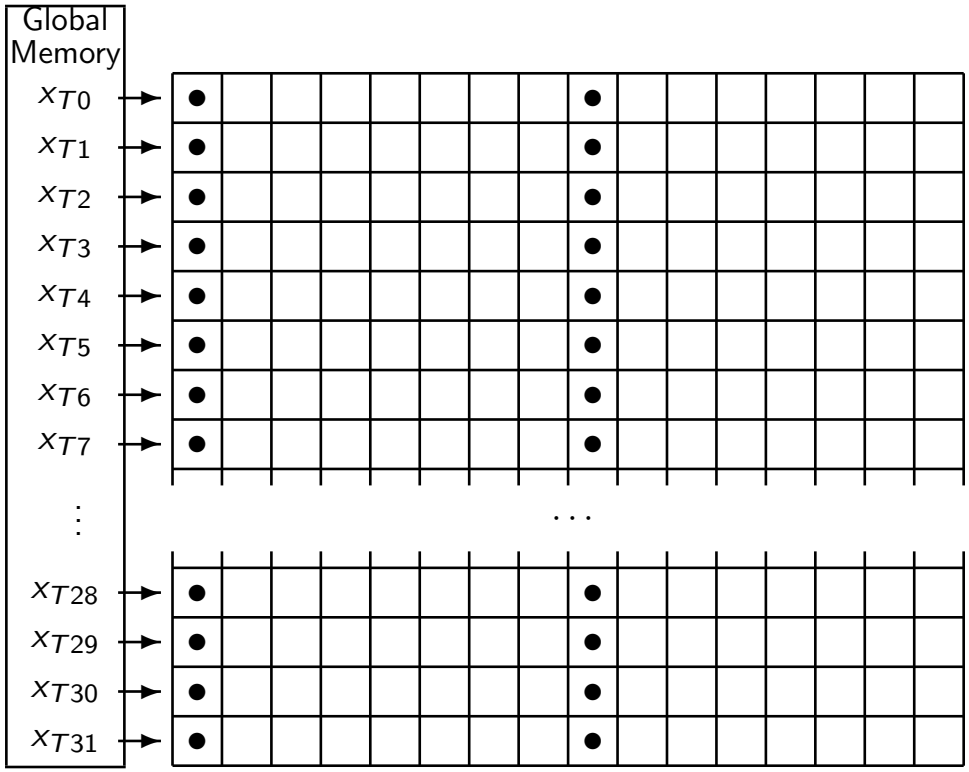


PGI CUDA Fortran Kernel: How threads read target data

Example

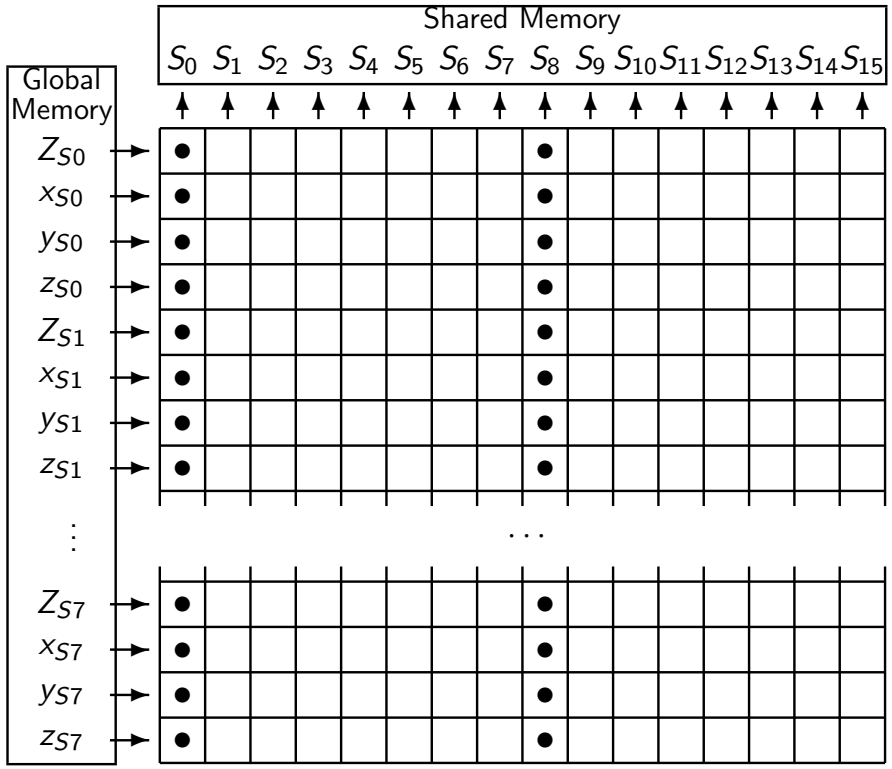
- Thread block dimensions: $(m, l) = (32, 2)$
- Force submatrix dimensions: $(m, n) = (32, 16)$
- Each thread column is one warp.
- Data order matches thread ID order
  - Reads are coalesced.
- Threads keep (x_T, y_T, z_T) in registers

Circles represent threads.



PGI CUDA Fortran Kernal: How threads read source data

- Source data stored in transpose order to target data in global memory.
- Four data items per source: (Z_S, x_S, y_S, z_S)
- Each warp reads 8 sources.
- Data order again matches thread ID order
  - Reads are coalesced.
- Threads write $S = (Z_S, x_S, y_S, z_S)$ to shared memory.



Calculation of Partial Forces by Thread Block (a, b)

- Thread (i, k) in thread block (a, b) applies its sources to its target

$$\mathbf{F}_i^k = \sum_{c=0}^{C-1} \sum_{j=km/4}^{(k+1)m/4-1} \mathbf{f}_{am+i,(cB+b)n+j}$$

- Threads use shared memory to combine their results.

$$\mathbf{F}_s(i) = \sum_{k=0}^{I-1} \mathbf{F}_i^k$$

- Thread columns share work of writing block's results to private area in global memory

$$\mathbf{F}'_g(am + i, b) = \mathbf{F}_s(i)$$

Sum Forces Over All Blocks in Row *a*

- Last block to write its results to $\mathbf{F}'_g$ sums over all blocks in row *a*. Thread columns share reduction work.

$$\mathbf{F}_i^k = \sum_{b=0}^{(B-1)/l} \mathbf{F}'_g(am+i, bl+k)$$

- Threads again use shared memory to combine results

$$\mathbf{F}_s(i) = \sum_{k=0}^{l-1} \mathbf{F}_i^k$$

- Thread columns share work of writing final results to global memory

$$\mathbf{F}_g(am+i, b) = \mathbf{F}_s(i)$$

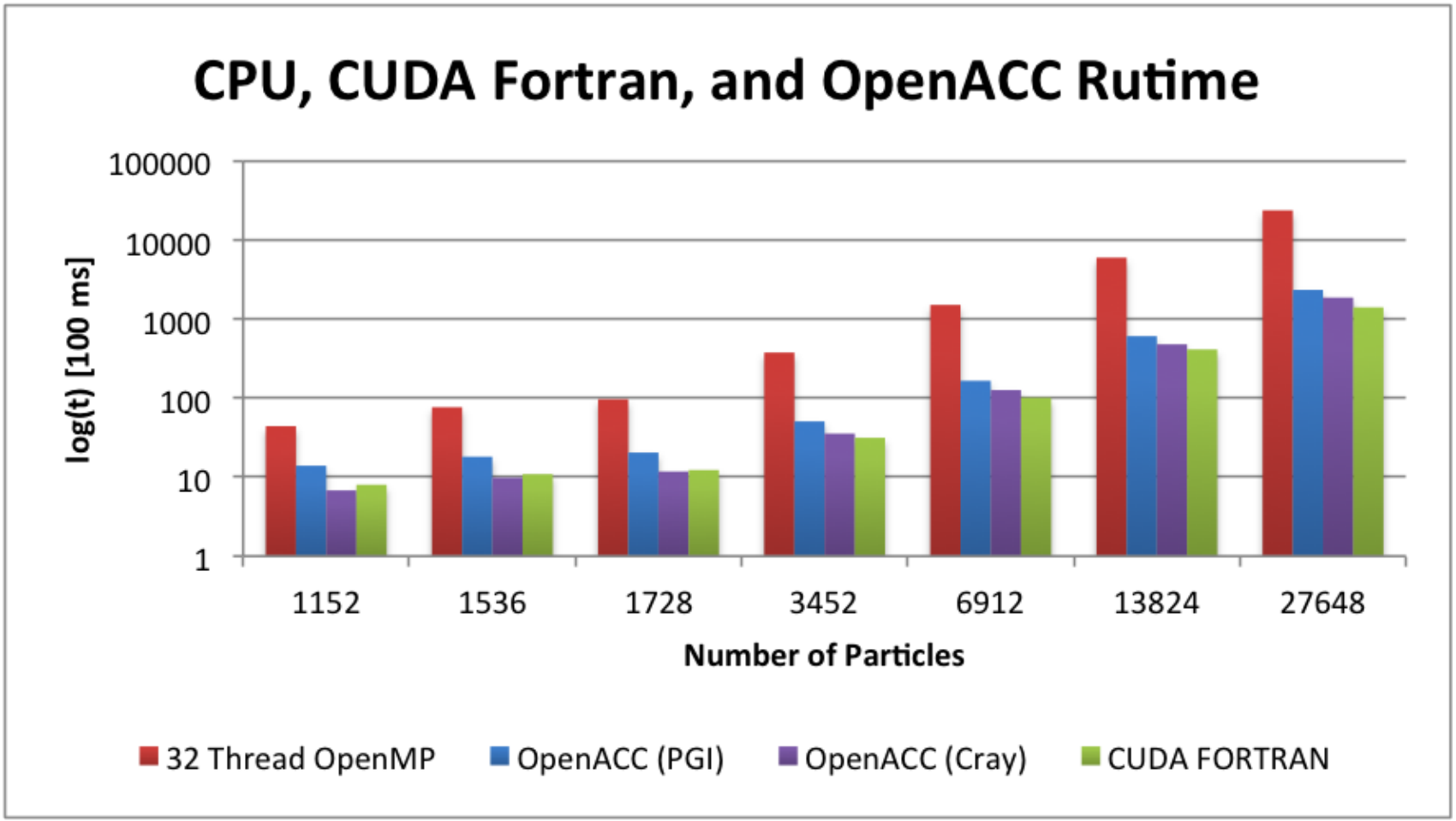
- Host reads $\mathbf{F}_g$ and finishes newton subroutine

Benchmark Runs

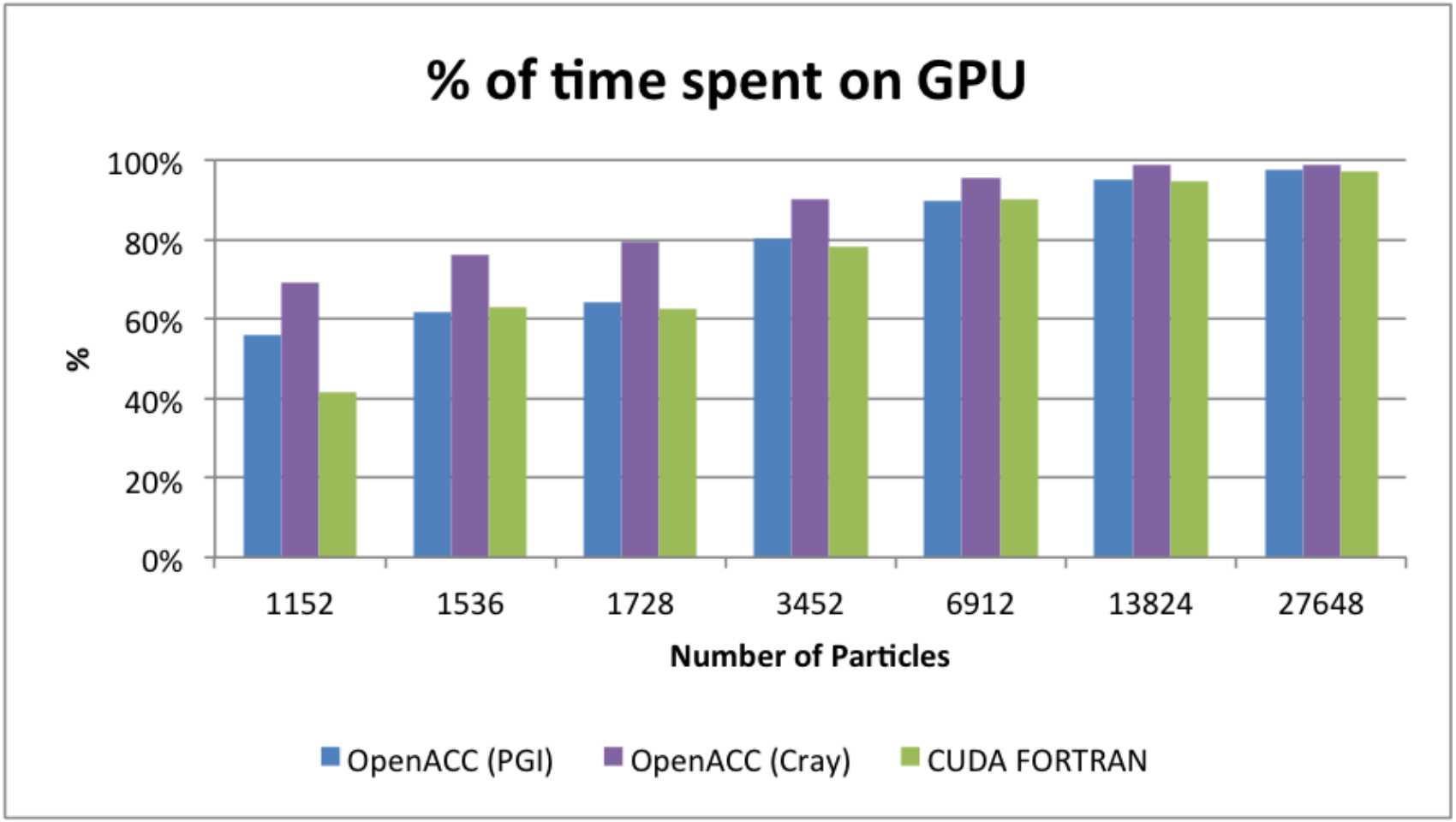
- Only single-node runs using **N** = 27648 particles. No MPI.
- OpenACC, CUDA Fortran vs. 32-thread OpenMP
  - Effect of $P \times P$ MPI process grid simulated by partitioning **N** particles into P sets.

P	27648/P	P^2
1	27648	1
2	13824	4
4	6912	16
8	3456	64
16	1728	256
18	1536	324
24	1152	576

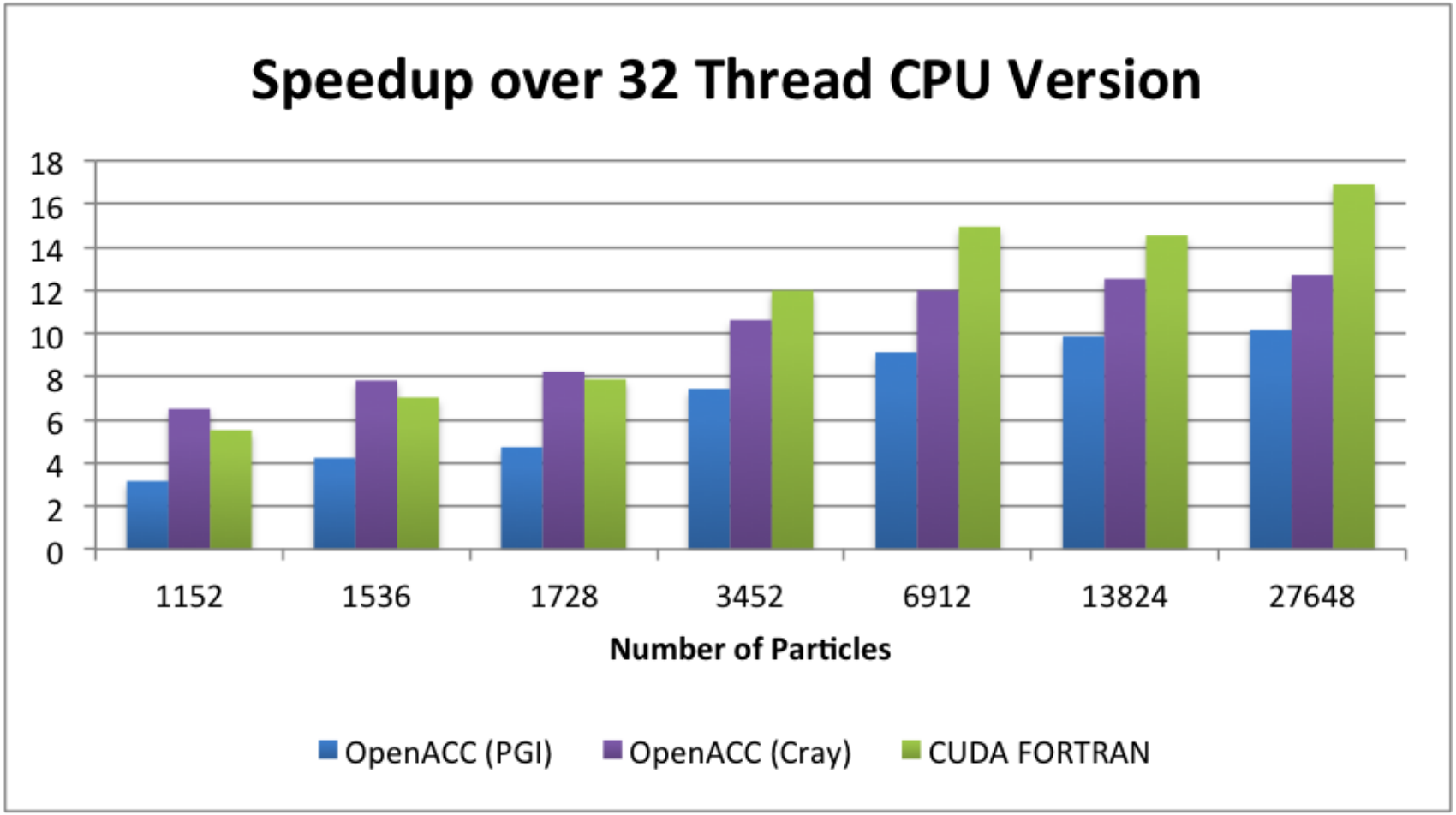
Run Times for Newton Subroutine



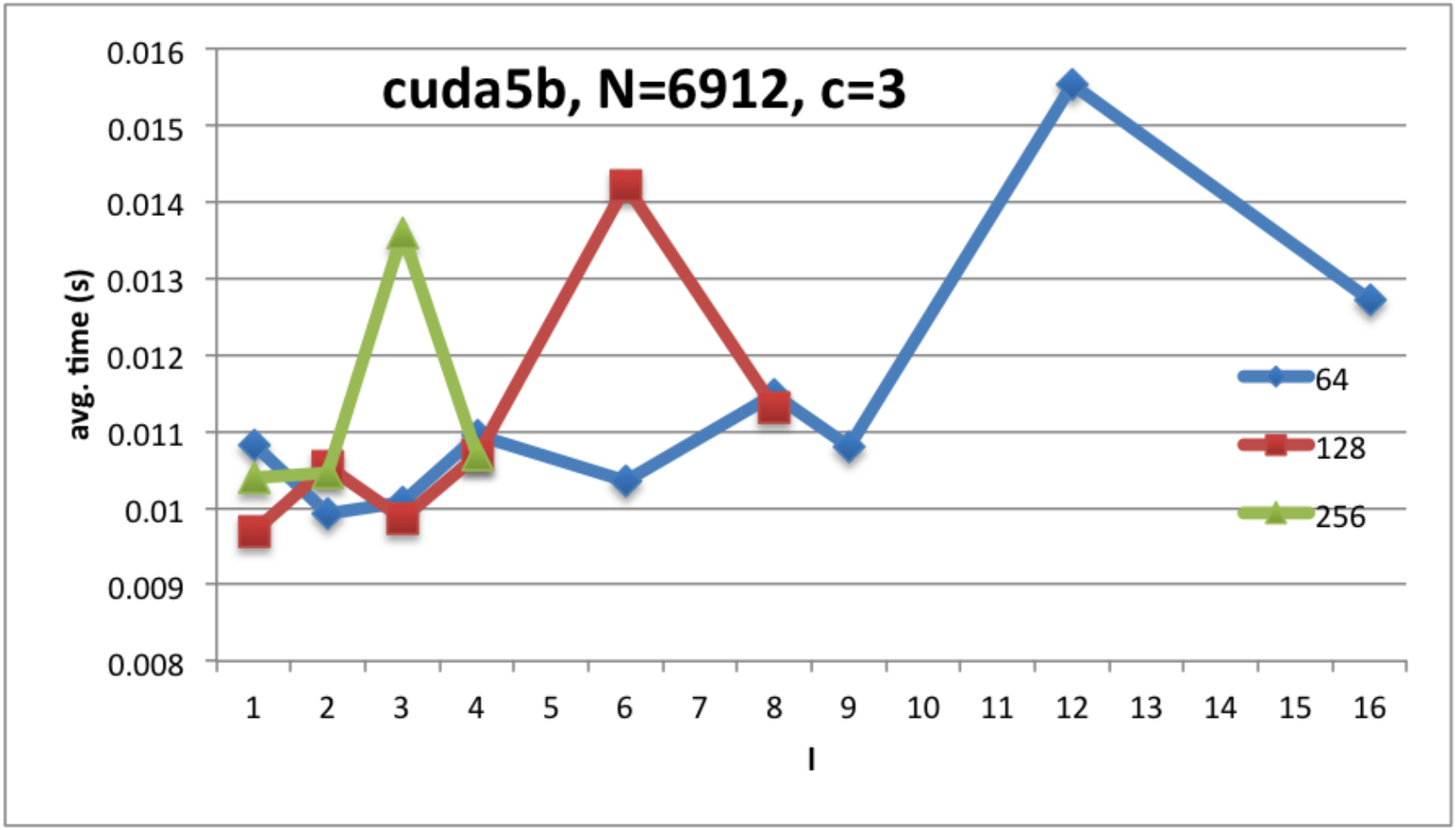
Overhead for Newton Subroutine



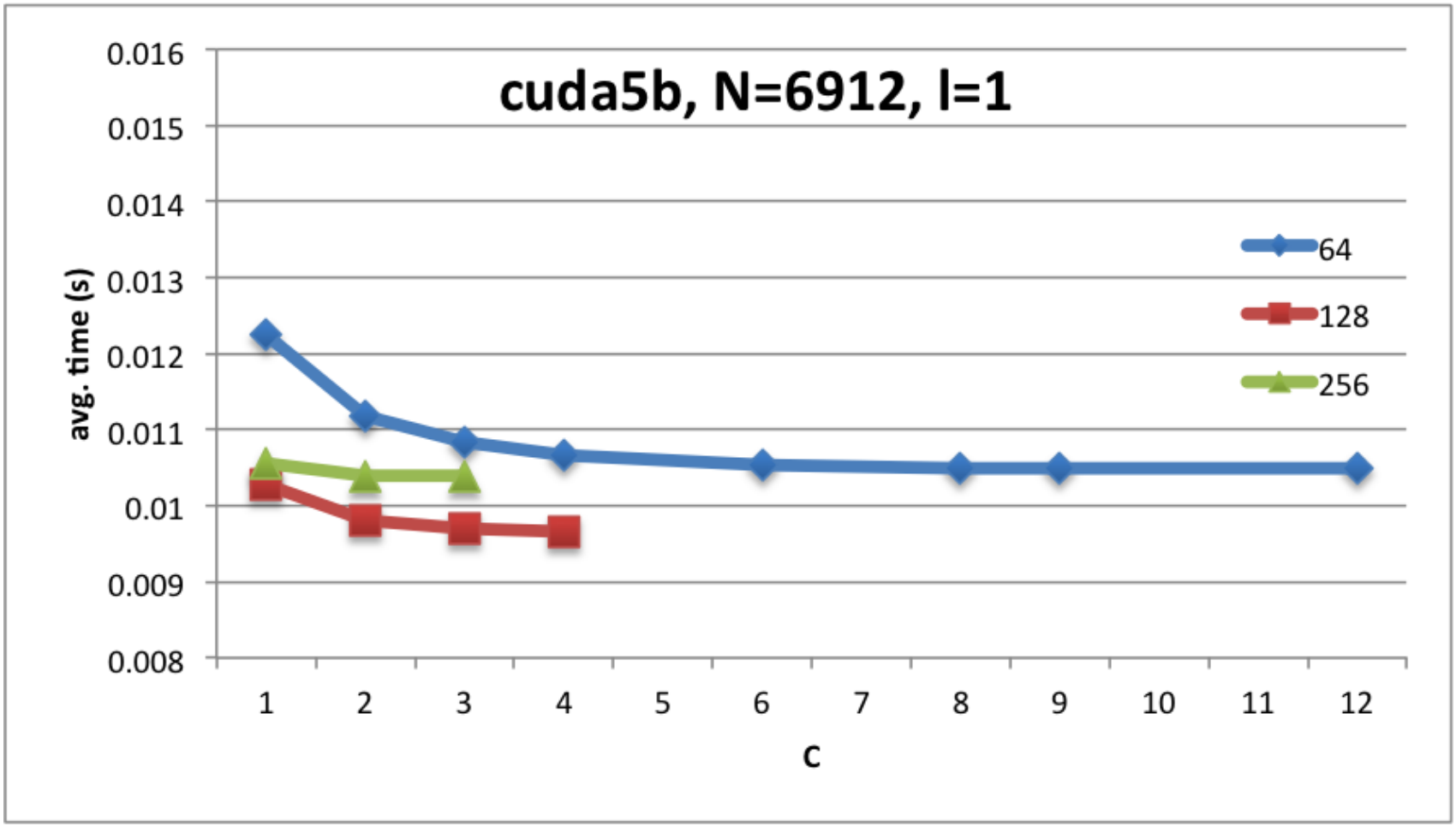
Speedup of Newton Subroutine



Performance of CUDA kernel vs. I for fixed C, m



Performance of CUDA kernel vs. C for fixed (m, l)



Conclusion

- OpenACC provides a quick, easy way to enter GPU computing. Provides good speedup and experience with the hardware.
- For production codes used for your multi-million SU allocation take the time to learn GPU architecture, and the CUDA programming environment!