CSE 6230: GPU Extension for PySPH NNPS

Pushkar Godbole
GTid: 903041324

December 6, 2014

1 Introduction

Smoothed Particle Hydrodynamics (SPH)[1, 2] is a meshless Lagrangian particle based method for numerical simulation of continuum mechanics problems. The method was primarily developed for simulating astrophysical problems by Gingold, Monaghan and Lucy in 1997 but has attracted significant attention in the last decade in the fluid-solid simulation paradigm, particularly for free-surface flows. SPH works by dividing the fluid into a set of discrete elements, referred to as particles. These particles have a radius of influence known as the *smoothing length*, over which their properties are *smoothed* by a kernel function; which is essentially a function that defines the variation of the influence of a particle on its neighbors with variation in the distance. The particles' influence beyond the smoothing length is ignored.

PySPH[3, 4] is a Python based open source framework for Smoothed Particle Hydrodynamics (SPH) simulations developed under Prof. Prabhu Ramachandran in the Aerospace Engineering department at IIT-Bombay. The current version is able to carry out million particle fluid, solid simulations both in serial and parallel. It is capable of dynamic load-balancing across processors using different load-balancing algorithms, based on the type of problem being solved. The computations for the simulations however are executed completely in CPU. Augmenting the module with a GPU extension would further speed-up the computations, thus expanding its capacity to do larger and finer simulations.

2 Objective

Because the influence of an SPH particle is restricted within the radius of smoothing length, particles in SPH are divided into boxes of dimensions equal to the smoothing length, known as *cells*. While identifying the particles within the influence radius of a particle, a cell layer of thickness (T) equal to one is considered for search as illustrated in the figure below:

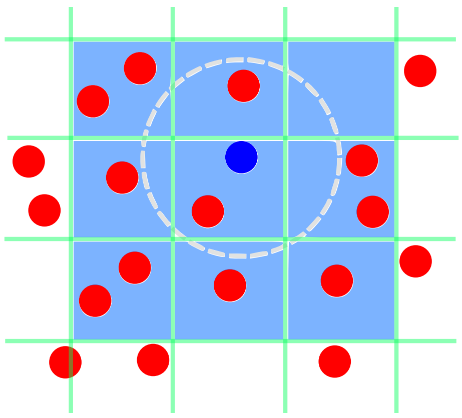

Figure 1: Neighbor identification in T=1 cell layer

Only the particles in the cells colored blue are evaluated for neighbor identification.

Each step of the SPH simulation can be broadly divided into two parts:

- Nearest Neighbor Particle Search (NNPS)
  - All particles are binned into cells based on their current positions
  - These cells are used to identify neighbors for each particle from within the adjacent cell layer
- Property update (PU)
  - Forces between all particles are computed based on the NNPS information
  - Properties of particles (such as positions, density, pressure, etc) are updated based on the computed forces

These steps are cyclically repeated at each iteration until the end of simulation time. Due to the $O(kn)$ nature of both the above phases (n being the number of particles and k the number of neighbors of a particle), they consume approximately equal time-fraction per iteration.

To that end, the localized nature of the SPH computations renders it particularly suitable for implementation over the GPU. Within the current design of PySPH, GPU compatible modifications to the the PU phase would require deeper changes in the architecture of the framework, beyond the scope of this project. Hence the objective of this project is restricted to GPUizing the NNPS phase only. The speed-up achieved by GPUizing, NNPS would contribute to the subsequent development of a GPU compatible version of PySPH.

3 Related Work

With the advent of High Performance Computing, highly parallel million to billion order particle SPH simulations have been made possible recently. Notable amongst them are SPHysics[5], GPUSPH[6] that can carry out massively parallel CPU-GPU based SPH simulations; GADGET[7] which is developed primarily for cosmological SPH/N-Body simulations.

GPU based SPH simulations is a relatively new field and has particularly gained a boost after the introduction of Nvidia's CUDA architecture and programming environment in 2007. The first instance of GPU's capability to simulate SPH was demonstrated in the pioneering work by Harada et al.[8] implemented prior to CUDA using OpenCL, that achieved speedups of about 30 over single core CPUs. The non-CUDA based nature of this implementation imposed several restrictions on their implementation such as flat 3D texture based memory allocation and use of the (RGBA) alpha channels for storage of indices.

With most of these problems addressed for general purpose GPU usage in CUDA, the recent work by Hérault et al.[9] on the CUDA based GPUSPH illustrates speedups of order of 60 for over half a million particles. The neighbor particle search strategy implemented by them is based on the algorithm described in the *Particle Simulation* documentation by S. Green [10] in the CUDA SDK. Green illustrates two approaches for grid generation (that can be used for implementing the cell structure of SPH). The Atomic Operations approach (supported by GPUs with compute capability 1.1 or higher) allows multiple concurrent threads to update the same memory area without use of explicit locks. This facilitates building of neighbor lists on a per-cell rather than per-particle basis, making the implementation very intuitive and portable. However, the memory access in this case is non-uniform and if multiple particles write to the same cell location simultaneously, the writes are serialized, causing further performance degradation. An alternative to this is the Sorting based approach that generates a hash for each particle based on the cell it belongs to. The particles are then sorted and ordered cell-wise by running a Radix sort on their hash-values. Since this method is particle based, the issue of concurrent writes, as in case of the Atomic operations, is evaded. A similar Radix-sort based approach is used in the neighbor location in DualSPHysics (Crespo et al.)[11], which is a GPU extension to SPHysics. Speedups of one to two orders of magnitude over the serial CPU version have been observed in this case. The Python library PyCUDA[12] will be used for implementation and integration of the GPU based NNPS with PySPH.

4 Nearest Neighbor Particle Search (NNPS)

As described previously, the NNPS routine is responsible for identifying the neighbors for each particle in the simulation domain. NNPS achieves this task in two steps:

- **Binning:** Each particle is mapped to a cubical bin of particles based on its position and the maximum cell-size. The maximum cell-size is the dimension of the bin and is essentially the maximum over the influence radius of all particles in the simulation. These mapped particles are then added to the appropriate bins and the cell lists populated. The non-Cuda version of NNPS achieves this in $O(n)$ time (n being the number of particles), using a dictionary-like data structure.
- **Neighbor search:** The binning information from the above step is used in generating the neighbor lists corresponding to each particle. Each particle searches for neighbors in its adjacent cells (9 in case of 2D and 27 in case of 3D simulation domains). Thus, for an average of k particles per cell and n particles in the simulation domain,the neighbor particle search over all particles in the non-Cuda version takes $O(kn)$ time.

Due to their localized nature, both the above routines exhibit good potential in terms of portability to GPU. The following sub-section will describe the working of the algorithms implemented and optimizations made to migrate NNPS to the GPU. The descriptions are in chronological order of implementation.

4.1 Cuda Neighbor search

Two different approaches were attempted to execute the neighbor particle search over GPU. The first is a linked-list based approach that although more threaded, was substantially slower than the second approach.

4.1.1 Linkedlist based Neighbor Search

Brief: Takes as input a destination array (of particle indices) for which neighbors are to be computed particle-wise and a source array (of particle indices) that serves as the source for the neighbor particle search space.
Returns as output an array of particle indices of neighbors from the source for each destination particle.

Data-transfer (CPU → GPU):

Source arrays: *x, y, z, h* (smoothing length: required for influence radius calculation)
Destination arrays: *x, y, z, h*
2D array of source Particle indices corresponding to each cell
3D Cell-map of the cells' x, y, z coordinates to the Particle indices array position

Kernels:
Algorithm:

- The first step is identification of source particles within the influence radius for each particle in the destination array. This is done running a kernel on a 3D boolean array. Each slice of the array corresponds to a destination particle. Rows in this slice correspond to each neighboring cell (9 for 2D simulation and 27 for 3D) Every element in each row is a boolean value representing interaction between that destination particle and that source particle (0 $\Rightarrow distance > influence\ radius$, 1 $\Rightarrow distance \leqslant influence\ radius$). Therefore,

$$NumThreads \approx NumDestinationParticles \times MaxParticlesPerCell \times 9[2D]/27[3D]$$

- With all neighbors for all destination particles identified, the next step is to extract them out into a contiguous block. This is done by an approach similar to the parallel prefix-sum scheme. The objective of this step is to generate a linked list for each destination particle, linking all its neighbors. Thus at every recursive step each element points to its next element if it is 1 (is a neighbor) or the element pointed by the next element otherwise. The linked-list for a particle would be complete when every element containing a 1 points to the next element containing a 1. Therefore, this check has to be performed using a global flag at every iteration of the loop. The kernel can exit when all linked-lists for all destination particles are complete.

- Finally, the third kernel is responsible for copying the linked-lists into a contiguous block. For this, $NumDestinationParticles$ threads are run, one corresponding to each linked list and the neighbor index data aggregated.
- In the end, all neighbor index data is copied back to the CPU and compared to the non-cuda results for run-time and accuracy.

References

[1] J J Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703, 2005.

[2] Daniel J. Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *J. Comput. Phys.*, 231(3):759–794, February 2012.

[3] Prabhu Ramachandran and Kunal Puri, Indian Institute of Technology Bombay. Pysph, 2009.

[4] Kunal Puri, Prabhu Ramachandran, Pankaj Pandey, Chandrashekhar Kaushik, and Pushkar Godbole. Pysph: A python framework for sph. *8th International SPHERIC Workshop, Trodheim, Norway*, 2013.

[5] University of Manchester (U.K.). Sphysics, 2007.

[6] Alexis Hérault, Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia. Gpusph, 2008.

[7] Volker Springel, Max Planck Institute for Astrophysics. Gadget, 2005.

[8] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on gpus. page 63ñ70, 2007.

[9] Alexis Hérault, Giuseppe Bilotta, and Robert A. Dalrymple. Sph on gpu with cuda. *Journal of Hydraulic Research*, 48(sup1):74–79, 2010.

[10] Simon Green. Particle simulation using cuda. *NVIDIA Whitepaper, December 2010*, 2010.

[11] Alejandro C. Crespo, Jose M. Dominguez, Anxo Barreiro, Moncho Gómez-Gesteira, and Benedict D. Rogers. Gpus, a new tool of acceleration in cfd: Efficiency and reliability on smoothed particle hydrodynamics methods. *PLoS ONE*, 6(6):e20685, 06 2011.

[12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to {GPU} run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.