CSE 6230: GPU Extension for PySPH NNPS

Pushkar Godbole
GTid: 903041324

December 8, 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.

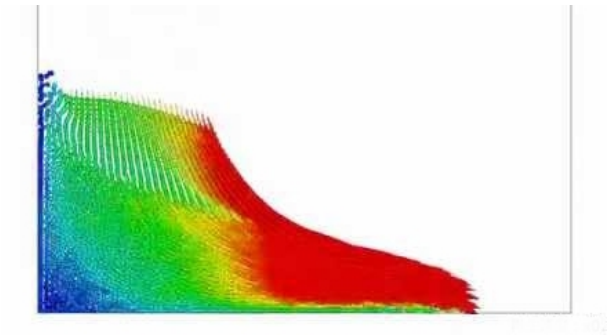

Figure 1: Dam-break SPH simulation[1]

SPH works by dividing the fluid into a set of discrete elements, referred to as particles. These particles have a spatial distance property known as the "smoothing length" (h), 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. Kernel functions commonly used include the Gaussian function and the Cubic-Spline. The latter is exactly zero for particles further away than a fixed distance unlike the Gaussian function, in which the influence follows the Gaussian curve and every particle maintains a diminishing but finite impact with the distance. In context of SPH, a Kernel function is almost always defined with a finite radius of influence. In general, this radius is taken to be twice the smoothing length and a particles' influence beyond the radius of influence 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 release 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.

[1] https://i1.ytimg.com/vi/DkRaKdHPbWI/hqdefault.jpg

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:

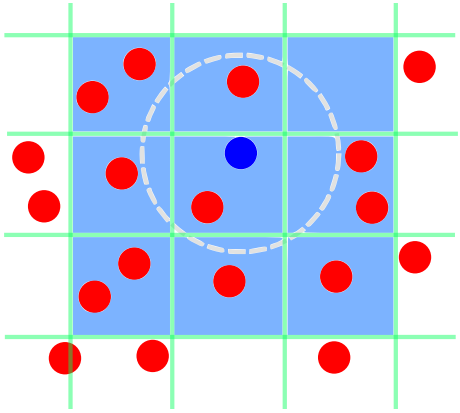

Figure 2: Neighbor identification in T=1 cell layer

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. The NNPS phase on the other hand is relatively delinked from the internal framework of PySPH. 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:** The binning of particles into their corresponding cells is particularly crucial in the context of SPH as it significantly reduces the subsequent neighbor search space from $O(n^2)$ to $O(nk)$ (n: number of particles | k: average number of neighbors per particle). This step is responsible for imparting SPH computations the localized nature, subsequently making them suitable for implementation over GPUs. The first step in initializing binning is to define the cell dimensions. Since the radius of influence of an SPH particle is equal to twice the smoothing length (h), it can be intuitively observed from figure 2 that cells with dimensions of twice the maximum smoothing length over all particles would suffice in identifying the neighbors for each particle. After the cell size is computed, each particle is mapped to its corresponding cell based on its position. 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.

5 CUDA NNPS: Design & Implementation

This section describe the design and working of the algorithms implemented for GPUizing Binning and Neighbor Search and additional optimizations made thereof to completely port NNPS to the GPU. The descriptions of the algorithms are in chronological order of implementation.

5.1 Cuda Neighbor search

Initially, while keeping the Binning unaltered, two different approaches were attempted to execute the neighbor particle search over GPU. The first was a linked-list based approach that although more threaded, resulted in substantially higher runtimes even with respect to the CPU version. It however set the basis for the implementation of the subsequent algorithm that has achieved significant speedup over the CPU version.

5.1.1 Linkedlist 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 contained in each cell
3D Cell-map of the cells' x, y, z coordinates to the Particle indices array position

Kernels:

- `isneighbor(source: [x, y, z, h], destination: [x, y, z, h], cell size, max cell population, cell indices, cell map, is_neighbor)`
  This kernel is called by $n \times MaxCellPopulation \times 27$ threads. Each thread represents a destination particle - potential neighbor pair. This function sets the boolean is_neighbor flag corresponding to that pair if the two particles are neighbors based on their distance.

- `update_linkedlist(linkedlist1, linkedlist2, is_neighbor)`
  The lengths of these linkedlists are equal to the length of the is_neighbor array. This function is similar to the prefix-sum function that updates linkedlist2 based on linkedlist1. Essentially, if an element of linkedlist1 points to a non-neighbor, the corresponding element in linkedlist2 is updated to point to the element pointed by that non-neighbor and unchanged otherwise. This kernel is also called by $n \times MaxCellPopulation \times 27$ threads (which is also equal to length_is_neighbor), each thread updating an element of linkedlist2 based in the corresponding element of linkedlist1.

- `getneighbors(is_neighbor, linkedlist, neighbors, num_neighbors, max cell population)`
  This function uses the linkedlist (which is assumed to be a chain of addresses of neighbors in is_neighbor) to populate the neighbors list for each particle. This kernel is called by num_destination_particles threads, each building the neighbor list for a particle.

Algorithm:

- The first step is identification of source particles within the influence radius for each particle in the destination array. This is done running the `isneighbor` kernel on the 3D boolean array `is_neighbor`. 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. The objective of this step is to generate a linked list for each destination particle, linking all its neighbors. Thus at every step, the `update_linkedlist` kernel is called by alternatively swapping linkedlist1 and linkedlist2 until one of them stagnates to a chain of neighbor addresses, 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 `getneighbors` is responsible for copying the addresses pointed by the linkedlist 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 anf GPU memory freed.

This method is theoretically bottlenecked by the third kernel call `getneighbors` as it runs only $NumDestinationParticles$ threads each individually running k threads, k being the average number of neighbors per particle. Therefore the depth of this algorithm comes out to be k. It was however observed that, majority of the time in this method was consumed by the repeated calls to `update_linkedlist` kernel until the linkedlist got updated completely. With this inference under consideration, the next algorithm for Neighbor Search was designed to be simple as follows.

5.1.2 Simple Neighbor Search

The algorithm for this function is very similar to the Linkedlist Neighbor Search algorithm with one major difference that, the algorithm calls only one kernel exactly once to generate the neighbor list. The algorithm does away with the `isneighbor` and `update_linkedlist` kernels while `getneighbors` does the complete task of iterating over all potential neighbors of a particle, computing distances and populating the neighbor list for each particle. Thus the algorithm generates neighbor lists for all particles by independently running a thread for every destination particle iterating over all potential neighbors in adjacent cells. The independence of each thread in generating its own neighbor list makes its execution clean and latency free. This simple method, as can be seen in the results, performs significantly well over the single threaded CPU version of Neighbor Search. With this method, the theoretical execution time (depth) of the Neighbor Search routine comes down from $O(kn)$ to just $O(n)$.

5.1.3 Optimizations

- **Device Memory Pool:** Since NNPS needs to be executed at every iteration of the SPH simulation, the function `getneighbors` has to be called frequently. This implies frequently copying the property arrays (x, y, z, h) from the host to device. Hence a device memory pool is initialized at the start of the simulation and the gpu arrays to transfer these properties to the device are requested from this pool.
- **Pagelocked memory:** Similarly, since the neighbor list has to be generated at every iteration and the max number of particles doesn't change frequently, the `neighbors` array to be fed to `getneighbors` is declared as pagelocked memory at the start of the simulation once the maximum cell population is computed. The array is sized to accommodate twice the max cell population to avoid constant re-initialization. If the max cell population increases through the simulation beyond this size, the `neighbors` array is reinitialized. Pagelocking of the `neighbors` array rules out the necessity to explicitly transfer it back to the host for subsequent force computations.

5.2 Cuda Binning

Although particle binning is an $O(n)$ process compared to the Neighbor Search, which is an $O(kn)$ process, the primary reason to port binning to GPU is to completely shift NNPS to GPU, precluding the necessity of any data transfer. Particularly, in the case of neighbor search, it was observed that majority of the time in the routine was consumed in building the cell indices lists from the cells data structure generated by binning. Thus GPUizing binning would not only speedup the binning process but also reduce the requirement of host to device data transfers, thus augmenting to further speedup. The peculiar aspect of particle binning is that, unlike Neighbor Search, it is not thread independent, at least in the context of particle-wise binning as multiple particles attempting to append to a cell is inevitable. This introduced the necessity of Atomic Operations as will be seen in the subsequent description. The particle binning operation has been done in two steps:

5.2.1 Get max cell population

Previously, the max cell population necessary for Neighbor Search and also for binning was generated in the CPU. But this meant $O(n)$ execution time. To reduce the execution time of this operation, a special function to calculate maximum cell population via GPU has been implemented.

Brief: Takes as input an array of particle indices to be binned. Computes the number of particles belonging to each cell and returns the maximum over all of them.

Data-transfer (CPU → GPU):

Particle arrays: *x, y, z*

Kernels:

- `gencellpop(x, y, z, cell_size, cell population)`
  This kernel is called by $NumParticles$ threads. Each thread computes the parent cell of a particle and increments the population of that cell by 1. This increment however can occur simultaneously for multiple particles belonging to the same cell. Hence the atomicAdd operation is used to avoid race conditions.

Algorithm:

- After the particle properties (x, y, z) are transferred to GPU, the number of cells along each dimension are calculated using the domain information. These dimensions are used to define the cell population array initialized to zero. These are then fed to the `gencellpop` kernel to fill the cell population array with the populations of cells. Then PyCUDA's gpuarray.max method is used to compute the max across all cells of the gpuarray and the value returned.

5.2.2 Bin particles

With the max cell population known, we can next move on to declaring the $NumParticles \times MaxCellPopulation$ cells array to be populated by the binning function. Again in this case, multiple particles concurrently attempt to append to the same cell giving rise to race conditions. The algorithm below explains how they have been tackled

Brief: Takes as input an array of particle indices to be binned. Uses the max cell population to declare the cells array that is atomically populated by incoming particles.

Data-transfer (CPU → GPU):

Particle arrays: *x, y, z*

Kernels:

- `pop_cells(x, y, z, cell_size, cell population, cells)`
  This kernel also is called by $NumParticles$ threads. Each thread computes the parent cell of a particle and appends to it in the cell array. However, again since multiple particles may attempt to write to the same cell array, the cell population, initialized to zero is atomically incremented (atomicAdd) and the new particle is inserted into a position corresponding to the updated cell population. Since the cell population is incremented atomically, we can be sure that no two particles would attempt to write at the same memory location.

Algorithm:

- Similar to the previous case, after the particle properties (x, y, z) are transferred to GPU, the number of cells along each dimension are calculated using the domain information. These dimensions are used to define the cell population array initialized to zero. The max cell population is used to define the cells array which is initialized to -1s to indicate absence of a particle index in that position. These are then fed to the `pop_cells` kernel to populate the cells arrays.

5.2.3 Neighbor search compatibility

With the GPU based particle binning, NNPS has now completely migrated to GPU by this point. The kernel function for the neighbor search had to be modified to utilize this change. Particularly,

it was observed that in the CPU based Binning ang GPU based Neighbor Search combination, major chunk of the Neighbor Search was consumed by the transfer of binning data from the host to device. With the binning function ported to GPU, the neighbor search function now need no longer read the binning data from the host as it already exists on the device, thus cutting down on the major portion of the routine.

Having said that, counter-intuitively enough, although this GPU based binning and neighbor search combination gives accurate results, the performance of the new neighbor search kernel has degraded significantly, as will be discussed in the next sections.

6 Results

This section will illustrate the performance of CUDA NNPS under various modalities of GPUization as described above. In particular, five combinations of GPUizations and optimizations have been evaluated as follows:

- CPU based Binning and Neighbor Search (Base case)
- CPU based Binning and GPU based Neighbor Search
- CPU based Binning and GPU based Neighbor Search with GPU data allocated from a Device Memory Pool (DMP):
  - Particularly suitable for iterative applications like SPH that repeatedly allocate and request same data with updating values
- CPU based Binning and GPU based Neighbor Search with DMP, with the neighbor list stored in pagelocked memory

  This eliminates two requirements:
  - The need to redeclare an empty the neighborlist at every iteration, although the maximum particles in a cell (responsible for the size of the neighbor list) infrequently changes throughout the simulation
  - The need to copy back the neighbor lists from the device to the host at the end of every iteration
- GPU based Binning and Neighbor Search (Objective case)

The following plot illustrates the overall performance of the GPU based NNPS for the above five modalities:

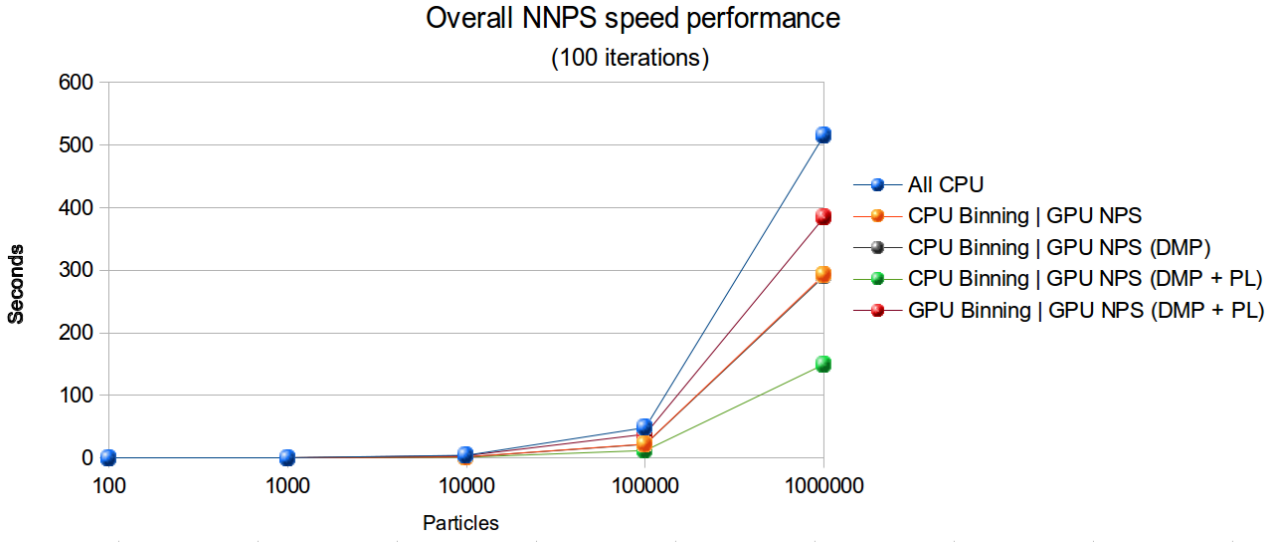

Figure 3: Overall GPU performance comparison

The following figure compares the performance of the CPU vs GPU Binning routines:

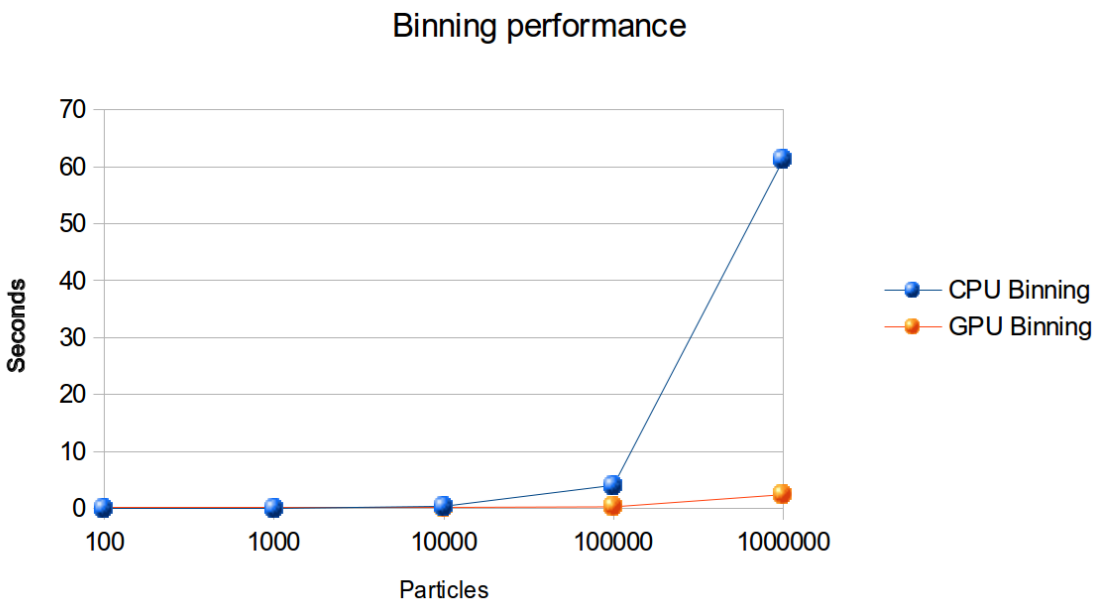

Figure 4: CPU vs GPU Binning comparison

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. *8^{th} 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.