THINS project: Intermediary report on the SPH code parallelization and validation for free surface flow

Luca Massidda, CRS4

CRS4, Science and Technology Park Polaris - Piscina Manna, 09010 Pula (CA) - ITALY

1 Introduction

The simulation of liquid metal targets may be difficult on standard CFD tools since phenomena such as pressure waves, cavitation, liquid splashing and free surface flows are not easy to be described with such tools.

The smoothed particle hydrodynamics (SPH) is a particle method very flexible and well suited for nonlinearities, it has been widely used to simulate explosions and free surface motion of water with wave breaking (Liu and Liu, 2003). First attempts to use this methodology to study the hydrodynamics of liquid metal targets have been conducted in the ISOLDE experiment at CERN (Noah et al., 2008). The Armando code has been developed to simulate such problems and has been verified on experiments and applied to the simulation of liquid metal targets such as for ESS.

The main disadvantage of particle-based methods is that they require a very large number of particles to obtain realistic results. It is therefore necessary to take advantage of parallel computing for this technique to be effective.

The method appears to be particularly suitable for massively parallel machines, rather than cluster of CPU for instance, so that it is possible to run particle computation in parallel as separate threads. It is therefore natural to adopt the massive parallel computation capabilities of modern GPUs to simulate large systems in extremely low computational times.

Graphics Processing Units (GPUs) appear as a cheap alternative to handle High Performance Computing for numerical modeling. GPUs are designed to manage huge amounts of data and their computing power has increased in the last years much faster than the CPUs. Compute Unified Device Architecture (CUDA) is a parallel programming method and software for parallel computing with some extensions to C/C++ language. Researchers and engineers of different fields are achieving high speedups implementing their codes with the CUDA language. Several works on particle methods have already appeared and also on SPH, obtaining good acceleration and performance, but still with some space for improvement.

2 Overview of the method

The simulation tools for physics may be roughly divide in two types – Eulerian (grid-based) methods, which calculate the properties of the simulation at a set of fixed points in space, and Lagrangian (particle) methods, which calculate the properties of a set of particles as they move through space.

SPH is a Lagrangian particle method, in which the continuum is discretized with a finite set of mutually interacting particles.

The idea at the basis of the method is to approximate any function $f(x)$ of the computational domain as:

$$f(x) = \int_{\Omega} f(x')W(x - x', h)d\Omega \tag{1}$$

The kernel function W is a smooth and differentiable function depending on the distance between the particles $r = |x - x'|$ and on the smoothing length h. It has a compact support and mimics the Dirac function as h approaches zero, so that its integral over the domain is equal to unity. In discrete notation, the previous approximation leads to the

following expression of the value of the function at the particle i position $f_i = \hat{f}(x_i)$:

$$f_i = \sum_j m_j \frac{f_j}{\rho_j} W_{ij} \quad (2)$$

where $W_{ij} = W(x_i - x_j, h)$ and the sum is carried out on all the particles and having mass m_j and density ρ_j.

The spatial derivatives of the function are obtained similarly by deriving the kernel, if $\frac{\partial W_{ij}}{\partial x^\alpha}$ is the gradient of $W(x_i - x_j, h)$ calculated with respect to the position of particle i, the discrete expression reads:

$$\frac{\partial f_i}{\partial x^\alpha} = \sum_j m_j \frac{f_j}{\rho_j} \frac{\partial W_{ij}}{\partial x^\alpha} \quad (3)$$

The methodology may be applied to gases, liquids and solids depending on the physical model considered (Liu and Liu, 2003).

The SPH approximations of the Euler equations are expressed as follows:

$$\dot{\rho}_i = \sum_j m_j (v_i^\alpha - v_j^\alpha) \frac{\partial W_{ij}}{\partial x^\alpha} \quad (4)$$

$$\dot{v}_i = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \frac{\partial W_{ij}}{\partial x^\alpha} + g_i^\alpha \quad (5)$$

$$\dot{u}_i = \frac{1}{2} \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \left(v_i^\alpha - v_j^\alpha \right) \frac{\partial W_{ij}}{\partial x^\alpha} + q_i \quad (6)$$

An Equation Of State is added to close the system, $p_i = f(\rho_i, u_i)$.

The variation in time of density, velocity and energy for the particle i are evaluated as a sum on all the other particles, therefore in general the complexity of the algorithm for a single time step is of the order of n_p^2. The kernel function W has a compact support, and rapidly becomes null as the distance between two particles increases. Therefore a cutoff distance is introduced, proportional to the smoothing length h, the number of neighbor particles to which the sum is extended is limited n_n, the complexity of the calculation of the interactions between particles is reduced to $n_n n_p$.

There are three main steps to the performing the simulation:

1. Find the neighbors for each particle
2. Calculate particle interactions
3. Integrate particle quantities in time

The integration step is the simplest step. It integrates the particle attributes (position and velocity) to move the particles through space. We use a three step Total Variation Diminishing Runge-Kutta scheme since it gives good smoothing properties and allows an higher value of the allowable time step than the common Leap Frog scheme. The increase of density velocity and energy is calculated from particle interactions, the particle positions are integrated on the basis of velocity value.

3 Neighbor search

A computationally expensive part in SPH simulation is the neighborhood search, performed on each particle for every time step. Searching for neighbors consists in finding the set of particles whose distance is lower than a given value for each particle of the computational domain. A brute-force force algorithm would result in computing all the n_p^2 possible distances, and would be very slow for any significant simulation.

The performance may be increased with a spatial subdivision techniques which divide the simulation space so that it is easier to find the neighbors of a given particle.

A uniform grid subdivides the simulation space into a grid of uniformly sized cells. For simplicity, we use a grid where the cell size is equal to $2h$ and is constant for the whole model. Each particle can potentially overlap several grid cells, but since the kernel function value is negligible at distances bigger than $2h$, the search for neighbor particles is limited to the grid cell where the particle is located and to the 8 in 2D and 26 in 3D cells surrounding it (3 x 3 x 3 = 27 in total).

Fixed grids have commonly been used to allocate particles to buckets for fast spatial range queries. The drawback of grid structures, however, is that they may use too much memory since the grid cell "buckets" containing the particles are allocated with a predefined capacity. In a typical simulation, many of these buckets are empty.

In our implementation the grid structure is generated from scratch at each time step but the size

of the buckets is not determined "a-priori", it is variable depending on the particle position, what is fixed is the maximum number of neighbors that any particle may have. The structured grid is described by means of an hash table.

The structure of the GPU requires a careful selection of the algorithm, to be optimized for this architecture, the choice of this neighbor search algorithm has been made on this basis, since it requires the minimum memory transfer operation between the host and the device and does not cause any memory conflict for parallel processes.

The algorithm consists in calculating for each particle the corresponding grid cell or bucket containing it, the array with this information has the size of the particle number and contains the index of the structured grid, that represents the hash key. The number of possible hash keys is equal to the number of grid cells of the structured grid that divides the computational space.

This array is sorted with respect to the hash keys, this way the indexes of particle that belong to the same grid bucket are located in adjacent position of the array.

Then two arrays are generated, with the size of the hash keys, pointing at the beginning and at the end of the group of particle with the same hash value.

The neighbor list can be finally generated, the possible candidates belong the finite set of grid cells around the particle (9 in 2D and 27 in 3D), the distances for all this particles are calculated and the particles whose distance is lower then the cutoff value are stored in the neighbor list until the maximum number of neighbors is reached.

In a standard configuration the smoothing length is usually close to the particle diameter $h = 1.8r$, therefore the numerical density of particle in a grid cell in 3D is

$$n_{cell} = \frac{(2h)^3}{\frac{4}{3}\pi r^3} \approx 11$$

and the number of neighbors may be estimated as:

$$n_n = \frac{\frac{4}{3}\pi(2h)^3}{\frac{4}{3}\pi r^3} \approx 47$$

The number of grid cells in normal problems is similar to the number of particles, therefore the data structure required for particle interaction computation in this implementation requires approximately $50n_p$ integers. At each time step it is necessary to calculate the distance for $27n_{cell}n_p \approx 300n_p$ particle pairs and the particle interactions to calculate the time variation of density, momentum and energy for $n_n n_p \approx 47n_p$ particle pairs.

It appears to us as a good balance between memory occupancy and computational requirements.

The algorithm consists of several kernels. The first kernel calculates a hash value for each particle based on its cell id. At this stage we are using a linear cell id as the hash, but it may be beneficial to use other functions such the Z-order curve to improve the coherence of memory accesses. The kernel stores the results as an array containing the hash value for each particle.

We then sort the particles based on their hash values. The sorting is performed using the fast radix sort provided by the CUDPP library, available with the CUDA distribution, that is at the moment the fastest sorting algorithm and library on GPUs. The kernel creates a list of particle indexes sorted on the basis of the hash value.

A third kernel is then used to sort the particle variables with the new index array, in this way particles that belong o the same cell and therefore have the same hash value are in close position in the arrays that store the particle data. Sorting improves the memory access coherency and the overall speed of the computation. Furthermore in a future optimization of the code, this can be used to further accelerate computation taking advantage of the block shared memory.

A fourth kernel finds the start and the end of any given cell in the sorted list, this is necessary because in our data structure the cell bucket does not have a fixed capacity, and we need to know which portion of the particle data arrays belong to any given cell. The kernel uses a thread per particle and compares the cell index of the current particle with the cell index of the previous particle in the sorted list. If the index is different, this indicates the start of a new cell, the end of each cell is found in a similar way.

Finally a fifth kernel calculates the neighbor list for each particle on the device; a thread per particle is generated and calculates the distance of the given particle from all the particle that belong to the 27 cells around (~ 300), all the particles whose distance is lower than the cutoff value (~ 47) are

kept in the list and used for further computations.

The CPU version of the code has been rewritten on the same algorithm basis for simplicity of maintenance of the code. The kernels launches on the GPU are substituted by cycles on the particles, the CPU calculates one particle at the time with the same steps previously described. The parallel radix sort of CUDPP library is substituted by the Quick Sort of the standard C library.

4 Particle interactions

The time derivative of density, velocity and energy for each particle are calculated through three sub steps, in a TVD Runge Kutta scheme of the third order. At each sub step the Lagrangian form of the Runge Kutta equations has to be solved, using the particle approximation as previously shown. Therefore for each particle, a summation over the neighbors is required to calculate the time derivative.

This is split in two phases to obtain a preliminary optimization of the code on GPUs. A first kernel solves for the mass and momentum equations, and a second kernel solves for the energy equation. This is due to the fact that the number of registers that can be used in the computation is limited, and if too many data for particle are treated at the same time, it may result in a reduction of the occupancy of the computing hardware due to the lack of registers and local memory for each thread.

The first computational kernel uses a thread per particle, cycles on the neighbors calculating the distance, the kernel function derivative, and sums the contribution on the density and velocity time derivative.

The second computational kernel operates in the same way but calculates the value of the time derivative of the particle energy, using also the velocity divergence to calculate by the first kernel.

The operations performed in this phase are quite simple when compared to the sorting algorithm, but the number of particles involved is very high, so this phase results in being the most computationally expensive for each time step.

The CPU version of the code has the same structure, kernels are substituted by functions, and the thread subdivision is replaced by a cycle on the particles.

5 Particle update

In the particle update phase, the position, velocity, density and energy of each particle is updated from the particle interaction calculation results. Moreover the Equation Of State is evaluated, giving as a result the pressure of each particle for its density and energy.

$$p_i = EOS(\rho_i, u_i) \tag{7}$$

On the GPU, this operation is realized with a kernel, launching one thread per particle. In the CPU version, the multiple thread execution is replaced by a cycle on all the particles.

6 Test cases

For the sake of simplicity and debugging, development and preliminary tests have been made on a 2D version of the code, the generality of the method and of the performance obtained is not affected significantly, and a further improvement in the acceleration is expected for 3D.

Two examples are presented in the following: a classical dam break problem and a pressure wave propagation due to sudden energy increase.

The first example puts in evidence the performance of the code in dealing with strong nonlinearities and free surfaces; the second are oriented towards the analysis of the short time scale transients induced by a sudden energy increase as in the case of beam dumping in a liquid metal target. In both cases, the performance of the code on the GPU and on a CPU are compared for several problem sizes.

The machine used to evaluate the performance is equipped with an Intel Xeon at 2.67GHz and the NVIDIA Tesla C1060 GPU card.

6.1 Dam break

The dam break flow with the consequent wall impact is widely used to benchmark various numerical techniques that tend to simulate interfacial flows and impact problems. A tank whose length is $L = 5.37m$ is partially filled with a column of water ($l = 2m$ and $h = 1m$) located in the left side of the model. Water is considered inviscid, but a

numerical viscosity is added in the modeling to regularize the results as well as some dumping on the mass conservation equation. Density is assumed as $\rho = 1000 kg/m^3$, gravity is $g = 9.81 m/s$ and the equation of state selected is a simple polynomial equation.

$$p_i = \frac{\rho c^2}{7}\left(\left(\frac{\rho_i}{\rho}\right)^7 - 1\right) \quad (8)$$

Energy effect is not considered and the speed of sound in the material is taken equal to $c = 50 m/s$. The compressibility of the water is therefore higher than reality, this allows to greatly reduce the simulation time in this kind of problems, since the time step may be increased. To obtain reasonable results the sound velocity must be sufficiently higher than the maximum fluid velocity found in the simulation.

Figure 1 shows the evolution of the model at several time steps. Results with CPU and GPU are identical.

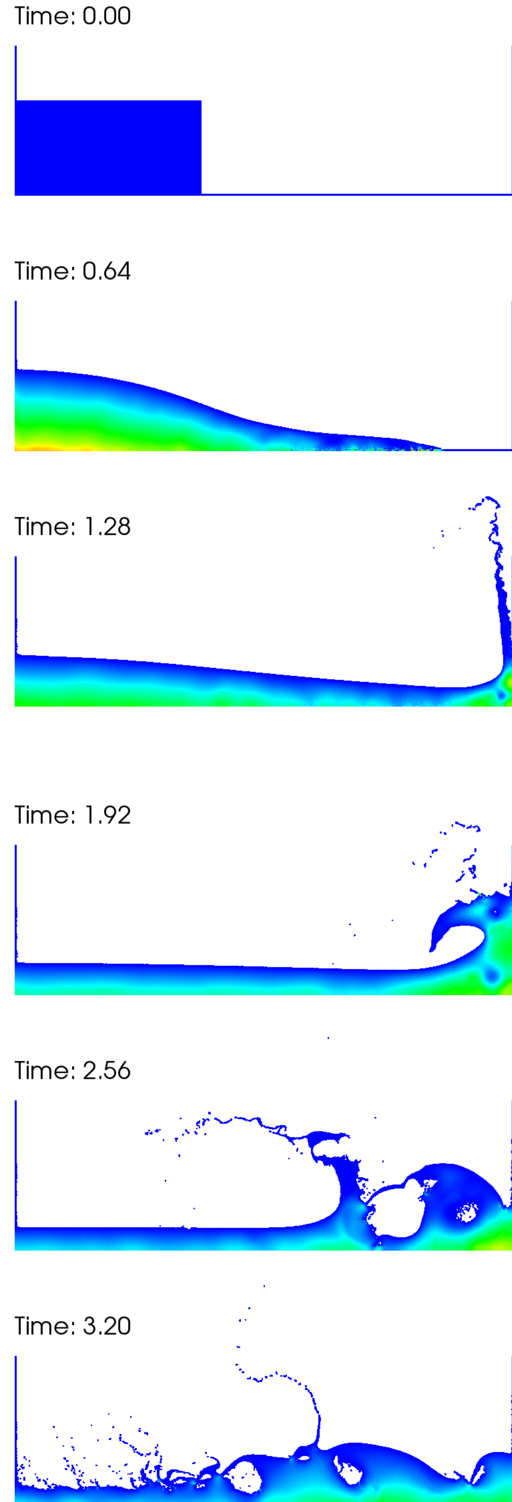

Fig. 1: Dam break test case: evolution the model and of the pressure field

The simulation was run using several particle spacing values and particle number. The performance measurements are listed in Table 6.1 It is possible to verify that the speedup factor with the GPU version of the code is higher than 50 as the size of the problem increases.

The results of the code profiling are listed in Table 2. The code spends most of its time in calculating the particle interaction to update the density and the velocity of each particle; this is the

Tab. 1: GPU and CPU performance on the Dam break test case

Dam break	Case 1	Case 2	Case 3
Particles	6462	22899	85773
Time steps	10000	20000	40000
CPU time	15m 26s	106m 44s	784m
GPU time	19s	1m 21s	8m 34s
Speedup	48	79	91

Tab. 2: GPU profiling for the dam break problem

Kernel	GPU time	Threads	Registers	Occupancy
balanceMassMomentum	63.71%	256	27	50%
updateList	20.8%	256	21	50%
updateParticles	3.54%	256	13	100%
balanceEnergy	0.53%	256	5	100%

core of the problem so it is not a surprise. The neighbor list calculation requires some 20% of the time, all the other kernels take much less computing time. The first two methods require a cycle on all the possible neighbor particles to calculate distances and interactions. All the other methods can be run in parallel without any communication within threads.

These two methods have space for optimization and will be realized in the following of the project. The occupancy of the GPU card is limited to 50% for these methods due to the high number of registers adopted. The card tested is classified as Compute capability 1.3, newer cards, of compute capability 2.0 have an higher number of available registers and would allow an occupancy of 75% for the *balanceMassMomentum* kernel and 94% for the *updateList* kernel. An higher performance is therefore expected on newer cards without any optimization of the code. We will try an optimization reducing the number of register, moving the data on the slower dynamic shared memory, in order to increase the performance also on older hardware.

The radix sort kernel used for for the hash array is surprisingly efficient.

Fig. 2: Pressure wave test case: energy density in the model

Tab. 3: GPU and CPU performance on the Pressure wave test case

Pressure wave	Case 1	Case 2	Case 3
Particles	160801	251001	361201
Time steps	800	1000	1200
CPU time	25m 2s	51m 14s	84m 4s
GPU time	33.5s	55.1s	1m 14s
Speedup	45	53	68

6.2 Pressure wave propagation

As a second example, a pressure wave propagates in a tank filled with mercury, due to an instantaneous temperature rise. The model is a section of the tank. It is square with a size of $10cm$. The energy increase is located at its center and has a Gaussian profile around the axis with a mean radius of $4mm$. The fluid is simulated with a polynomial equation of state, the lateral surfaces are free, the simulation is run for $0.1ms$ to cover the propagation along the domain, including the reflection on the free surfaces.

Figure 2 shows the energy density, and Figure 3 shows the pressure wave propagation at several time steps.

The simulation was run using several particle spacing values and particle number. The results are shown in Table 3. As the number of particle increases, the performance of the GPU is better when compared to the CPU since the effect of memory transfer betwen the computational devices are minimized. The speedup factor in this test case approaches 70.

Fig. 3: Pressure wave test case: pressure wave propagation at 20, 40, 60, 80μs

7 Future developments

In this preliminary report we have presented the performance results of the parallelization of the Armando SPH code on the GPU platform. The results are very encouraging, since a factor of more than 60 on the performance of a CPU core is obtained using an NVIDIA Tesla Card. It appears that GPU acceleration is perfectly suited for the method and on the basis of the profiling analysis it appears that further margin of improvement is possible through a more efficient use of the shared memory of the card and a reduction in the register usage, to maximize the operational throughput of the computational kernels.

In the final report, the code will be further optimized and applied to more complex problems of liquid metal flow and target design.

8 References

1. Gottlieb, S., Shu, C., 1998. Total varation diminishing Runge-Kutta schemes, Mathematics of Computation, Vol. 67, 221, 73-85

2. Green, S., 2010. Particle Simulation using CUDA, CUDA SDK Samples.

3. Liu, G., Liu, M., 2003. Smoothed Particle Hydrodynamics: A Meshfree Particle Method. World Scientific.

4. Massidda, L., 2008. ARMANDO, a SPH code for CERN. Tech. Rep., CERN Geneva.

5. Massidda, L. Kadi, Y., 2010. SPH simulation of liquid metal target dynamics, Nuclear Engineering and Design 240, 940-946.

6. Monaghan, J., Lattanzio, J., 1985. A refined particle method for astrophysical problems. Astronomy and Astrophysics 149, 135–143.

7. Monaghan, J., 1992. Smoothed particle hydrodynamics. Annu. Rev. Astron. Physics 30, 543. 12, 13

8. Monaghan, J., 1994. Simulating free surface flow with SPH. Journal of Computational Physics 110, 399–406.

9. Nyland, L., Harris, M., Prins, J., 2007. Fast N-Body Simulation with CUDA. GPU Gems 3. Addison Wesley, 2007