This is the accepted version of a paper presented at VRIPHYS.

Citation for the original published paper:

Implicit Incompressible SPH on the GPU.
In: Proceedings of Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)
2015 Lyon, France

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:bth-10997
Implicit Incompressible SPH on the GPU

Prashant Goswami †, André Eliasson‡ & Pontus Franzén‡

Blekinge Institute of Technology, Sweden

Abstract
This paper presents CUDA-based parallelization of implicit incompressible SPH (IISPH) on the GPU. Along with the detailed exposition of our implementation, we analyze various components involved for their costs. We show that our CUDA version achieves near linear scaling with the number of particles and is faster than the multi-core parallelized IISPH on the CPU. We also present a basic comparison of IISPH with the standard SPH on GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing

1. Introduction
Smoothed particle hydrodynamics (SPH) has emerged as a powerful method to simulate fluid behaviour in a number of graphics applications. The possibility of enforcing computationally inexpensive incompressibility has been introduced in predictive-corrective incompressible SPH (PCISPH) [SP09] and improved upon in implicit incompressible SPH (IISPH) [ICS*14], and later in divergence-free SPH (DFSPH) [BK15] in the context of particle simulation to achieve more realistic fluid behaviour in recent years. The ability of the IISPH solver to simulate millions of particles together with employing larger time steps makes it promising for real-time considerations.

Similar to other particle-based methods, a large particle count is desirable in IISPH to achieve high resolution features in the simulation. To this end, sequential and multi-core simulations remain beyond the scope of any real-time purpose for more than a few thousand particles. Most modern graphics applications (games, simulators etc.) make massive use of the on-board computational power of the GPUs, where fluid simulation could be just one of the several components. Therefore, in order to fit efficiently within the existing framework, it is desirable to explore a GPU solution for IISPH.

We propose a completely parallel, GPU-based solution for IISPH using CUDA. The particles are resident on the GPU-memory throughout the simulation and could be used for rendering in the next step on the graphics card itself. We show that real-time to interactive frame rates are achievable using our simple solution, exhibiting linear dependence on the number of particles. We also give an initial estimate of how IISPH fairs in comparison to the standard, compressible SPH solver in terms of performance on the GPU.

2. Related Work
SPH has several benefits over other fluid simulation techniques like simplified boundary handling, obtaining fine-scale effects like splashes and implicit mass conservation. It was first introduced to computer graphics in [DG96] for deformable bodies and later to simulate fluids [MCG03]. Later [BT07] came up with a weakly compressible model for free surface flows.

Traditional SPH formulation needs a large stiffness value (and hence small time steps) to simulate incompressible behaviour, thereby increasing the computational time. The first promising work in this direction in computer graphics was by [SP09] where the density of particles are fixed in a predictive-corrective manner. The method was a significant improvement as it eliminated solving computationally expensive matrices for Poisson equation while still being able to handle large time steps.

Further work improved the state-of-the-art for simulating incompressible fluid behaviour, mostly towards reducing the cost. [RWT11] employ a hybrid approach by solving Poisson...
equation on a coarse grid and transferring the initial pressure estimate to particles for density correction. In [HLL+12] Poisson equation is solved locally and integrated with the predictive-corrective framework. In [MM13] incompressibility is formulated as a positional constraint satisfying problem which enables taking even larger time steps than PCISPH. IISPH [ICS+14] improves the predictive-corrective incompressible SPH method by using a modified projection scheme which improves the convergence rate of the solver. Though multi-core CPU parallelization yields a linear parallel scaling as demonstrated in [TSG14], the frame rates are no longer interactive beyond a few thousand particles.

Several methods improve the efficiency of standard SPH. These include developing hybrid SPH–FLIP model to reduce actual physical particles in memory [CPI+14], using multiple resolutions [SG11], skipping computation on inactive fluid parts [GP11] and employing regional time stepping [GB14]. In this work, we focus our discussion on techniques more closely related to ours and refer the reader to [IOS14] for an in-depth survey of the various fluid simulation methods.

Recent growth in the hardware capabilities has seen an emergence of efficient parallel solutions, particularly on the GPU. [HKK07,ZSP08] were one of the first to propose GPU solutions for SPH which employed shaders for the purpose. [GSSP10] came up with a CUDA-based solution utilizing shared memory, which scores both on the memory usage and efficiency fronts. Continuing along the line, the aim of this work is to explore the parallel GPU porting of IISPH.

3. IISPH Basics

Similar to PCISPH, IISPH does not rely on any stiffness parameter that appears in the equation of state (EOS) to compute pressure. However, it uses a discretization of PPE (pressure Poisson equation) which is a variant of earlier introduced incompressible SPH (ISPH) formulation in [ESE07]. The density (and error) prediction is obtained by discretized continuity equation.

IISPH solver comprises three major steps: advection prediction, pressure solving and moving particles, in that order. In the advection stage, density \( \rho \) and velocity \( \mathbf{v} \) of all particles are predicted using external forces like viscosity and gravity. The particles then enter density correction iteration wherein the pressure value for each particle is updated which in turn updates its density. The iterative loop is designed to continue until all the particles have density error below a specified threshold \( \eta \), subjected to a minimum of 2 iterations. Finally the velocity and position of the particles are obtained from the computed forces.

A major difference here with respect to PCISPH is that the density correcting pressure is built implicitly by iteratively solving linear equation for each particle and not directly computed and accumulated from the density error. The implicit formulation improves the convergence of the iterative solver. Similar to previous methods [SP09], [MM13], during a simulation step the computed neighbourhood set is reused through the iterations. The IISPH method is as outlined in Algorithm 1. We refer the reader to [ICS+14] for a more detailed understanding of the method.

```
Algorithm 1: IISPH Method

\begin{algorithm}
\begin{algorithmic}
\Procedure{Predict Advection} {\( \text{particle } i \) do}
\State Compute \( \rho_i(t) = \sum_j m_j W_{ij}(t) \)
\State Predict \( \mathbf{v}_{\text{adv}}^i = \mathbf{v}_i(t) + \Delta t \frac{\rho_{\text{adv}}^i(t)}{\rho_i(t)} \)
\State \( d_i = \Delta t^2 \sum_j - \frac{m_j}{\rho_i(t)} \nabla W_{ij}(t) \)
\EndProcedure

\Procedure{Pressure Solve} {\( l = 0 \) while \( (\rho_{\text{avg}}(\mathbf{v}) > \eta) \lor (l < 2) \) do}
\State \( \sum_j d_j \rho_j^l = \Delta t^2 \sum_j - \frac{m_j}{\rho_j^l(t)} \nabla W_{ij}(t) \)
\EndProcedure

\Procedure{Integration} {\( \text{particle } i \) do}
\State Compute \( \rho_{\text{avg}}^i = \rho_i(t) + \Delta t \sum_j m_j (\nabla \cdot \mathbf{v}_{\text{adv}}^j)^{\text{avg}} \)
\State \( \rho_i(t) = \rho_{\text{avg}}^i \)
\State \( l = l + 1 \)
\EndProcedure
\end{algorithmic}
\end{algorithm}
```

4. IISPH on CUDA

Neighbourhood determination: Like other particle-based methods, the first step in IISPH is to determine neighbourhood set for each particle. To this end, we follow a similar approach as laid out in [GSSP10] with some changes. The simulation domain is divided into a virtual indexing grid along each axis, with block size equal to the support radius for the particles. However, we replace Morton based \( z - \) index with simple linear index for all blocks in the simulation domain (Kernel calcIndex in Algorithm 2). This avoids expensive bitwise operations while still maintaining coherence between adjacent particles. The particles are then sorted based on this index using fast radix sort provided by thrust library (Kernel sortParticles). To benefit from the increased memory on modern GPUs and in order to be able to take maximum advantage of available parallelism, for each particle we explicitly store the indices of all its neighbouring particles. The shared memory technique introduced in [GSSP10] could be still useful to fit higher particle count on a limited GPU memory, though with some penalty on parallelism.
The old particle data like positions and velocities (assigned to CUDA texture) are copied to new sorted index in the global memory array (Kernel sortParticleData). In order to determine the neighbours, the start and end index of each cell in the virtual grid is stored (Kernel findGridCellStartAndEnd). This is accomplished by simple linear scanning of sorted particles which update their index in the respective cell for the start and end. Finally, each particle collects its neighbours from the cell it lies in together with the adjoining 26 cells (Kernel updateNeighbours). The neighbouring indices are stored for each particle and are reused each time neighbours are fetched during subsequent computations in the iteration. The above operations are launched as separate kernels, see also Algorithm 2.

**Predict advection:** The PREDICT ADVECTION procedure comprises two separate kernels. In the first step, density is predicted which is required for the computation of factor \( \mathbf{d}_{\text{adv}} \). \( \mathbf{v}_{\text{adv}} \) is obtained using viscosity and external forces such as gravity. Thereafter, launch of a second kernel is necessitated by the observation that all particles need to finish with \( \mathbf{d}_{\text{adv}} \) calculation before \( \rho_{i}^{\text{adv}} \) because of the relative velocity which appears in the density formulation. Following this, the factor \( a_{ii} \) is determined for each particle.

\[
a_{ii} = \sum_{j} m_{j} (d_{ij} - d_{ji}) \nabla W_{ij}
\]

Whereas the value of \( d_{ij} \) computed from previous kernel is used, \( d_{ji} \) requires an extra computation (but no storage). Since the value of \( a_{ii} \) remains unchanged for an iteration, it is stored in the global memory for each particle and subsequently made a CUDA texture.

**Pressure solve:** The density is resolved through two or more corrective loops on each particle per iteration. For this purpose, the kernels in PRESSURE SOLVE are launched from the standard CPU call for each loop. Kernel computeSumPressureMovement updates \( \sum_{j} d_{ij} p_{j} \), which is required in Kernel computePressure to obtain the updated pressure \( p_{i}^{t+1} \).

\[
p_{i}^{t+1} = (1 - \omega) p_{i}^{t} + \omega \frac{1}{a_{ii}} \left( \rho_{0} - \rho_{i}^{\text{adv}} - \sum_{j} m_{j} \left( \sum_{k} d_{ijk} p_{k} - \sum_{l \neq i} d_{ijk} p_{l} \right) \nabla W_{ij} \right)
\]

The quantity \( \sum_{j} d_{ij} p_{j} \) is supplied as a CUDA texture since it is used first for the computation of \( p_{i}^{t+1} \) and then \( p_{i}^{t+1} \). Furthermore, \( \sum_{k} d_{ijk} p_{k} \) is also derived from it. At the end of each loop, the new density \( \rho_{i}^{t+1} \) (Kernel computePredicted-Density) and hence the error \( \rho_{i}^{\text{err}} \) (Kernel computeDensity-Error) is computed.

\[
\rho_{i}^{t+1} = \rho_{i}^{\text{adv}} + p_{i} \sum_{j} m_{j} (d_{ij} - d_{ji}) \nabla W_{ij} + \\
\sum_{j} m_{j} \left( \sum_{k} d_{ijk} p_{k} - d_{ij} p_{j} - \sum_{l \neq i} d_{ijk} p_{l} \right) \nabla W_{ij}
\]

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Texture Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeDisplacementFactor</td>
<td>( x_{i}, v_{i}, N_{i} )</td>
</tr>
<tr>
<td>computeAdvectionFactor</td>
<td>( x_{i}, v_{i}^{\text{adv}}, d_{ii}, N_{i} )</td>
</tr>
<tr>
<td>computeSumPressureMovement</td>
<td>( x_{i}, \rho_{i}, p_{i}, N_{i} )</td>
</tr>
<tr>
<td>computePressure</td>
<td>( x_{i}, d_{ii}, \sum_{j} d_{ij} p_{j}, \sum_{i} d_{ij} p_{j}, N_{i} )</td>
</tr>
<tr>
<td>computePredictedDensity</td>
<td>( \rho_{i}^{\text{err}} )</td>
</tr>
<tr>
<td>computeDensityError</td>
<td>( v_{i}^{\text{adv}}, \rho_{i}, p_{i}, N_{i} )</td>
</tr>
<tr>
<td>calcIntegration</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Quantities supplied as texture for the various CUDA kernels.

- **CPU:** Intel Core i7-3770 (3.4GHz)
- **GPU:** MSI GeForce GTX 970 Gaming 4G
- **RAM:** DDR3, 1600MHz, 16GB
- **OS:** Windows 7 Ultimate 64-bit

(a) Setup 1

- **CPU:** Intel Xeon E5-1650 (3.2GHz)
- **GPU:** NVIDIA Quadro K4000
- **RAM:** DDR3, 1600MHz, 16GB
- **OS:** Windows 8.1 Enterprise 64-bit

(b) Setup 2

Table 2: Hardware specifications for experiments.

The total number of corrective loops depends on the density error. In order to estimate the global density error, we launch the reduction kernel in CUDA which supports parallel addition and maximum finding operations.

Even though dynamic parallelism is supported in recent versions of CUDA with compute capability 3.5 or higher, the dependence of quantities makes it infeasible to efficiently launch a kernel from within another kernel. This is because we require global synchronization as opposed to synchronization just within a block (for example, \( \rho_{i}^{t+1} \) depends upon the computation of its neighbours’ \( p_{j}^{t+1} \)).

**Integration:** The new velocity is computed for each particle by summing up pressure forces from the neighbours and adding the contribution to the advection component \( \mathbf{v}_{\text{adv}} \). For this the complete neighbour set is made available to each particle as texture memory.

The complete list of quantities supplied as texture memory for the various CUDA kernels is given in Table 1.

5. Results

The proposed approach was implemented in C++ using DirectX and HLSL shaders. We used two different setups for our experiments as given in Table 2.

In all scenarios a fixed time-step of 3.5 ms and particle spacing of 0.09 m was used. The density error threshold \( \eta \) was set to 1% of the rest density \( \rho_{0} \) (= 1000). All the kernels were launched with a maximum of 256 threads per
Algorithm 2: Parallel IISPH on CUDA

```c
procedure NEIGHBOUR DETERMINATION
 Kernel calculateIndex
 Kernel sortParticles
 Kernel sortParticleData
 Kernel findGridCellStartEnd
 Kernel updateNeighbours

procedure PREDICT ADVECTION
 Kernel computeDisplacementFactor
 foreach particle i do
   compute \( \mathbf{d}_i \)
   predict \( \mathbf{v}_i^{adv} \)
   compute \( \mathbf{d}_i \)

 Kernel computeAdvectionFactor
 foreach particle i do
   compute \( \mathbf{p}_i^{adv} \)
   compute \( \mathbf{p}_i^0 \)
   compute \( \mathbf{a}_i \)

procedure PRESSURE SOLVE
 l = 0
 while \( (\mathbf{p}_{avg} - \rho_0 > \eta) \lor (l < 2) \) do
   Kernel computeSumPressureMovement
   foreach particle i do
     compute \( \sum_j \mathbf{d}_{ij} p_j^l \)
   Kernel computePressure
   foreach particle i do
     compute \( p_i^{l+1} \)
     \( p_i(t) = p_i^{l+1} \)
   Kernel computePredictedDensity
   foreach particle i do
     compute \( \rho_i^{l+1} \)
     compute \( \rho_i^{l+1} \)
   Kernel computeDensityError
   compute \( \rho_{avg} \)
   \( l = l + 1 \)

procedure INTEGRATION
 Kernel calculateIntegration
 foreach particle i do
   compute \( \mathbf{v}_i(t + \Delta t) \)
   compute \( \mathbf{x}_i(t + \Delta t) \)
```

Three versions of the algorithm were implemented. The first one was a sequential version running on the CPU. The second one was the proposed parallel version running on the GPU. A simple parallel implementation of the CPU-version was also developed using OpenMP for comparison, though [TSG14] would give a better estimate on this front. The various scene set-ups used are as shown in Figure 1. The CUDA solution is able to achieve higher performance than an OpenMP implementation on the CPU. Setup 1 (Table 3) achieved a speed-up of about 6 times compared to the parallel CPU version. In setup 2 (Table 4) the speed-up (about 2 times) was lower because of a slower graphics card.

The CPU has a steady time per frame regardless of the scene while the CUDA times fluctuate slightly on computer setup 2 but on setup 1 the fluctuation is greater, see also Figure 2. The various scenes differ slightly from each other, they still follow a linear growth when the number of particles increases. A similar linear growth in memory usage was detected in both the CPU- and the GPU-implementation for all test scenes, see Figure 3.

Figure 4 demonstrates the average time spent on each CUDA kernel in our implementation on the faster setup. For larger particle counts, Kernel computeDisplacementFactor, Kernel computePredictedDensity, Kernel computePressure and Kernel computeAdvectionFactor are the most expensive ones (in that order). Further, we notice that the neighborhood computation is not so expensive when compared to the other kernels. Using our implementation, we obtain a total occupancy of about 0.75 in our experiments. To estimate the cost of incompressibility with IISPH, the presented approach was applied to standard, compressible SPH implementation and measured with both setups. The results are as given in Table 5. A low stiffness constant of 1000 was employed and the time steps were computed using CFL condition. The results for both setups in Table 5 follow a linear growth rate just like IISPH. For 175K particles, standard SPH is around 2.3 times faster than IISPH in terms of the number of physics iterations executed per second.

6. Conclusions

In this paper, we presented an efficient, CUDA-based parallel implementation of IISPH method. The proposed technique performs faster than the multi-core CPU-based parallel implementation and achieves near-linear scaling with the number of particles. A related future work would be to compare the GPU version of IISPH with other methods like divergence-free SPH and position-based fluids for efficiency.
Figure 1: Time lapse visualization of three different scenes with varying particle counts.

(a) Simple: 121K particles

(b) Gallery: 99K particles

(c) Two blocks: 116K particles

Table 3: The average time and fps (frames per second) measured on setup 1 for each scene with a calculated speed-up between the GPU and parallel CPU.

<table>
<thead>
<tr>
<th>Particles</th>
<th>Physics - GPU</th>
<th>Physics - CPU (OpenMP)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 600</td>
<td>2.58</td>
<td>0.68</td>
<td>3.89</td>
</tr>
<tr>
<td>20 000</td>
<td>4.06</td>
<td>2.24</td>
<td>45</td>
</tr>
<tr>
<td>54 000</td>
<td>10.21</td>
<td>64.20</td>
<td>6.29</td>
</tr>
<tr>
<td>103 000</td>
<td>21.07</td>
<td>126.80</td>
<td>6.02</td>
</tr>
<tr>
<td>175 000</td>
<td>39.18</td>
<td>221.16</td>
<td>5.64</td>
</tr>
</tbody>
</table>

Table 4: The average time and fps measured on setup 2 for each scene with a calculated speed-up between the GPU and parallel CPU.

<table>
<thead>
<tr>
<th>Particles</th>
<th>Physics - GPU</th>
<th>Physics - CPU (OpenMP)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 600</td>
<td>4.58</td>
<td>5.34</td>
<td>1.17</td>
</tr>
<tr>
<td>20 000</td>
<td>9.45</td>
<td>16.23</td>
<td>2.22</td>
</tr>
<tr>
<td>54 000</td>
<td>23.18</td>
<td>46.23</td>
<td>3.89</td>
</tr>
<tr>
<td>103 000</td>
<td>47.82</td>
<td>92.29</td>
<td>1.19</td>
</tr>
<tr>
<td>175 000</td>
<td>83.06</td>
<td>161.06</td>
<td>1.94</td>
</tr>
</tbody>
</table>

Table 5: Table showing the results of a standard SPH implementation on the two setups for comparison.

<table>
<thead>
<tr>
<th>SPH CUDA</th>
<th>Setup 1</th>
<th>Setup 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particles</td>
<td>time (ms)</td>
<td>fps</td>
</tr>
<tr>
<td>7 600</td>
<td>0.79</td>
<td>1268</td>
</tr>
<tr>
<td>20 000</td>
<td>1.15</td>
<td>329</td>
</tr>
<tr>
<td>54 000</td>
<td>7.42</td>
<td>173</td>
</tr>
<tr>
<td>103 000</td>
<td>16.55</td>
<td>64</td>
</tr>
</tbody>
</table>

7. Acknowledgements

We thank anonymous reviewers for their constructive comments that helped us to improve the paper. We also thank Pierre-Luc Manteaux (INRIA-Grenoble) for the insightful discussions on IISPH.

References


Figure 2: The graph visualizes the computation time of the algorithm per frame on setup 1 (a) and setup 2 (b) for the CUDA-solution compared to OpenMP. All four tests scenes used a time-step of 3.5 ms and a particles spacing of 0.09 m. Measurements was taken over 1 000 frames.

Figure 3: The GPU memory usage in all scenes grows linearly with the number of particles.

Figure 4: Split-up of the timings taken by the CUDA kernels in our GPU implementation.


